University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

Self-organizing system using collective robots

Diploma thesis

Submitted in partial fulfillment of the requirements
for the degree of *Engineer*
in the domain of *Electronics, Telecommunications and Information Technology*
study program *Applied Electronics*

Thesis Advisor(s)
Prof. Dr. Ing. Corneliu Burileanu
As. Univ. Drd. Ing. Ana Neacșu

Student
Teodora-Cătălina Vaman

July 2021

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Program de studiu **ELA**

Anexa 1

TEMA PROIECTULUI DE DIPLOMĂ

a studentului **VAMAN F. Teodora-Cătălina , 442B-ELA**

1. Titlul temei: Sistem de auto-organizare folosind roboți colaborativi

2. Descrierea temei și a contribuției personale a studentului (în afara părții de documentare):
Auto-organizarea este procesul prin care un sistem aflat initial intr-o starelipsita de ordine evolueaza prin interactiuni si feedback pozitiv catre ostare de echilibru. Inspirat de coloniile de furnici si de bancurile de pestidin natura care reusesc sa se auto aranjeze in forme diverse am decis cascopul proiectului sa fie implementarea unui sistem de roboti carecomunica si se organizeaza singuri. Robotii folositi (KiloBots) folosesc vibromotoare pentre deplasare si și își transmit informațiile intre ei prinintermediul undelor infraroșu. Pentru a facilita comunicarea, un controllereste montat deasupra roiului de roboti, astfel toate operatiile, precumincarcarea unui program sau verificarea bateriei, se pot realiza asupragrupului intreg in acelasi timp. Studentul va proiecta si testa un algoritmpentru a muta roiul de roboti din pozitia lor initiala catre o formaprestabilita. Pentru a se organiza in forma robotii trebuie sa se deplasezepe o cale fara coliziuni, fara blocaje si fara coordonare centralizata (fiecarerobot trebuie sa aiba acelasi program incarcat in memorie).Pentru a demonstra abilitatea grupului de roboti sa se organizeze,utilizatorul va putea sa specifice orice forma isi doreste, algoritmul vaprelua forma si o va trimite catre roboti. Dupa aceasta, robotii vor trebuisa comunice unii cu altii si sa isi stabileasca calea catre interiorul formeidorite fara a se ciocni intre ei si fara a crea blocaje

3. Discipline necesare pt. proiect:
AMP, SDA, POO, MC

4. Data înregistrării temei: 2020-11-24 12:49:52

Conducător(i) lucrare,
As. drd. Ing. Ana-Antonia NEACȘU

Prof. Corneliu Burileanu

Student,
VAMAN F. Teodora-Cătălina

Director departament,
Ș.L. dr. ing Bogdan FLOREA

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **eb71754cf3**

Anexa 5

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "*Sistem de auto-organizare folosind roboți colaborativi*", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică și Telecomunicații*, programul de studii *Electronică aplicată* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, 30.06.2021

Absolvent *Teodora-Cătălina VAMAN*

(semnătura în original)

Statement of Academic Honesty

I hereby declare that the thesis *Self-organizing system using collective robots*, submitted to the Faculty of Electronics, Telecommunications and Information Technologies, University POLITEHNICA of Bucharest, in partial fulfillment of the requirements for the degree of *Engineer* in the domain Electronics and Telecommunications, study program *Applied Electronics* is written by myself and was never before submitted to any faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand that plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, June 2021.
Student: Teodora-Cătălina Vaman

Table of Contents

List of figures . . . iii

List of tables . . . v
List of abbreviations . . . vi

1. Introduction . . . 1
1.1. Motivation . . . 1
1.2. Objectives . . . 2
1.3. State of the art . . . 2

2. Theoretical concepts . . . 7
2.1. Swarm Behaviour . . . 7
2.2. Kilobots . . . 9
2.2.1. Movement . . . 10
2.2.2. Communication . . . 10
2.3. Kilobot System . . . 11
2.3.1. Workspace organisation . . . 11
2.3.2. ATmega328p microcontroller . . . 12
2.3.3. Overhead Controller . . . 13
2.3.4. KiloGUI . . . 14
2.3.5. Programming Environment . . . 16
2.3.5.1. AVR-GCC compiler . . . 16
2.3.5.2. Kilolib . . . 16
2.3.6. CoppeliaSim . . . 20
2.3.6.1. Features . . . 20
2.3.6.2. Kilobot Model . . . 23

3. Algorithms . . . 26
3.1. Edge Following . . . 26
3.1.1. Orbiting with one stationary robot . . . 26
3.1.2. Orbiting with multiple stationary robots . . . 27
3.2. Gradient Formation . . . 27
3.3. Shape Formation 1 . . . 30
3.3.1. Matrix Generator . . . 32

3.4. Shape Formation 2 . . . 33
3.4.1. Trilateration . . . 35
3.4.2. Ray-Casting Algorithm . . . 36
3.4.3. Ramer-Douglas–Peucker Algorithm . . . 37

4. Results and comparisons . . . 39
4.1. Results using the real Kilobots . . . 39
4.2. Results using the simulation . . . 41

5. Conclusion . . . 45
5.1. General Conclusions . . . 45
5.2. Personal Contributions . . . 45
5.3. Further developments . . . 46
References . . . 47

Anexa A. Code for Edge Following . . . 50

Anexa B. Code for Edge Detection using gradient . . . 52

Anexa C. Code for shape formation algorithm 1 . . . 56

Anexa D. Code for shape formation algorithm 2 . . . 62

Anexa E. Code for Matrix Generation . . . 68

Anexa F. Code for Ramer-Douglas-Peucker Algorithm . . . 71

List of figures

1.1. Kilobots in the trail avoidance (stigmergy) condition [1] . . . 3
1.2. Swarms of robots forming spots (left) and stripes (right) [2] . . . 4
1.3. Overview of the platform and results of the experiments conducted with the Kilobot Soft Robot.[3] . . . 5
1.4. Different configurations obtained with a Kilobot system [4] . . . 6
2.1. Flock Simulation . . . 8
2.2. Kilobots Components [5] . . . 10
2.3. Image showing the reflection path of robot communication . . . 11
2.4. Picture of the work station, including the overhead controller (A), the group of 10 kilobots (B), the control station (C) and the charger (D) . . . 12
2.5. Picture of the overhead controller [6] . . . 13
2.6. Picture of KiloGUI app and the Calibration Menu [5] . . . 15
2.7. Compilation Process . . . 17
2.8. CoppeliaSim IDE . . . 24
2.9. Kilobot Parts . . . 25
3.1. Flowchart for orbiting algorithm . . . 26
3.2. Kilobots engaged in orbiting behaviour . . . 27
3.3. Schematic representation of the experiments demonstrating that the bicoid gene encodes the morphogen responsible for patterns in Drosophila . . . 28
3.4. Flowchart of Gradient Algorithm . . . 29
3.5. Gradient formation with real Kilobits . . . 29
3.6. Gradient formation with 30 robots in CoppeliaSim simulation . . . 29
3.7. Matrix Formation for a rectangle and a cross-like shape . . . 31
3.8. Representation of robots occupying position in the shape . . . 31
3.9. Two experiments for shape formation using real kilobots and CoppeliaSim . . . 32
3.10. The steps taken to modify a given picture . . . 33
3.11. Image showing the first 57 elements in the shape matrix . . . 34
3.12. The steps of shape formation . . . 34
3.13. Robots demonstrating the shape formation algorithm by forming the letter L . . . 35
3.14. Trialateration process . . . 36
3.15. Example of ray-casting of a polygon . . . 37
3.16. Thirty six robots forming the word "KILO" . . . 37
3.17. The various results of Ramer-Douglas–Peucker algorithm . . . 38

4.1. The results of four test in which the robots organize in a small rectangle using the Matrix Algorithm 39
4.2. The results of four test in which the robots organize in a small rectangle using the Polygon Algorithm 40
4.3. The results of the kilobots forming a rectangle in CoppeliaSim using the Matrix Algorithm 42
4.4. Fifteen robots in the process of forming a 150x150 rectanlge 43
4.6. Two experiments to illustrate the differences between the physics engines 43

List of tables

2.1. Technical specifications of the Kilobot 9
2.2. Motor calibration values 15
4.1. Results for Algorithm 1 simulation in ODE 42
4.2. Results for Algorithm 1 simulation in Bullet 2.83 42
4.3. Results for Algorithm 2 simulation in Bullet 2.83 44
4.4. Results for Algorithm 2 simulation in ODE 44

List of abbreviations

GPS = Global Positioning System
LED = Light-Emitting Diode
IR = Infrared
OHC = Over-Head Controller
ADC = Analog to Digital Converter
PWM = Pulse Width Modulation
RISC = Reduced Instruction Set Computer
TTL = Transistor–transistor logic
API = Application Programming Interface
ROS = Robotic Operating System
ODE = Open Dynamics Engine
IDE = Integrated Development Environment
ELF = Executable and Linkable File
RDP = Ramer–Douglas–Peucker
USB = Universal Serial Bus
ARK = Augmented Reality for Kilobots

Chapter 1
Introduction

1.1 Motivation

The world of robotics is a vast, interesting and very diverse one. Nowadays, robots can be found in an increasingly long list of domains, ranging from household helping robots, such as a smart vacuum cleaner that avoids obstacles and creates a map of the room in memory for an efficient cleaning strategy, to the Perseverance Rover[1] on its exploration mission 54.6 million kilometers away from Earth. From this vast industry, another sub-domain emerged, the field of self-organizing robots, the main purpose of which is to have a group of autonomous entities working together towards a common goal, without external human intervention.

The inspiration came from the natural world were small individuals collaborating to gather food, defend from predators, or build impressive structures are often found. A great example can be seen when looking at bee colonies which are able to divide the workforce into well defined categories, such as the builders who communicate with one another and raise the beehive, or the foragers who search for food and bring information back to the swarm. Observed from the outside, it almost seems as though the group is controlled by a mysterious force that dictates all the rules and activities, a force titled *the spirit of the hive* by the poet Maurice Maeterlinck in his 1927 novel "The life of Bees". We know now there is no external or internal authority that commands the others, it is rather the effect of each individual making the right decisions every time.

The purpose of this paper is to analyse algorithms that make use of this naturally occurring swarm behaviour and implement them using a group of kilobots[2]. The focus will be on algorithms that compel the group to form different kinds of two-dimensional shapes. Firstly, we examine a method that makes use of the distances among neighbouring robots and uses them as an indicator to determine whether or not the robot is correctly placed in the shape. The second method implements a local coordinate system shared between the robots who use it to create the desired shape. Finally, we will take the algorithms and test them by using a simulator and a real group of robots.

[1] https://mars.nasa.gov/mars2020/

[2] https://kilobotics.com

1.2 Objectives

After presenting the main aspects of swarm robotics, this paper aims to implement a system that uses those exact principles. That being said, the main objectives are:

- **Organizing the workspace**:
  In order to properly control the robots, a special arrangement is needed which takes into account all of their proprieties. They need a smooth surface to facilitate their movement and they need the controller to be placed at an appropriate distance so that every robot in the group receives the information concurrently.

- **Implementing and testing simple robot behaviours**:
  All complex algorithms share some basic fundamentals such as movement, communication or decision making. Therefore it is vital to test those fundamentals separately in order to properly check if everything is working accordingly and to correct them if otherwise.

- **Implementing and testing shape formation algorithms**:
  To better demonstrate the collaborative effect and the interactions between kilobots two complex algorithms are to be examined. Another objective is to experiment with the robots using these algorithms and record the result.

- **Comparing the results**:
  After conducting the experiments, the outcomes will be compared depending on their speed and accuracy. The advantages and disadvantages will be presented at the end of the paper.

1.3 State of the art

Even though the kilobots are a relatively new product, a good number of research papers have already been published. The papers tackle all aspects of collective behaviour, from coordinating to form different shapes to transporting objects or mimic aspects from the natural world.

Firstly, a classical task for robot swarms is area coverage and exploration. *Stigmergy* is a form of communication in which the individuals exchange information not directly but through some chemical agents or actions. In the world of ants, communication is established through different pheromones left behind by individuals as they move. The main types are *attractive pheromones* that alert other group members of something valuable such as food and *repellent pheromones* which may be used to notify others of an unprofitable foraging pathway [7]. In the word of robotics, researchers came to the conclusion that this phenomenon can be very useful in the exploration of unfamiliar territory because it helps robots spend less time searching in an area already visited by others. Therefore, they implemented different ways to simulate the chemical pheromones. Stigmergy is also useful as a control mechanism because the robots need little memory or processing power in order to operate in the environment because they only need to analyse the chemicals surrounding them. Digital pheromones can be achieved in different ways: as points on a map shared by all members in the swarm, by using other robots as reference points, using augmented reality to track kilobots in a virtual environment (a project called ARK)[8] or making use of the light sensor mounted on the robots and utilizing projections of light as guidance [9]. An interesting project consisted of designing a unique working space for the robots, called "Kilogrid" [10], composed out of a grid of computing nodes that provide a bidirectional channel between the robots and a

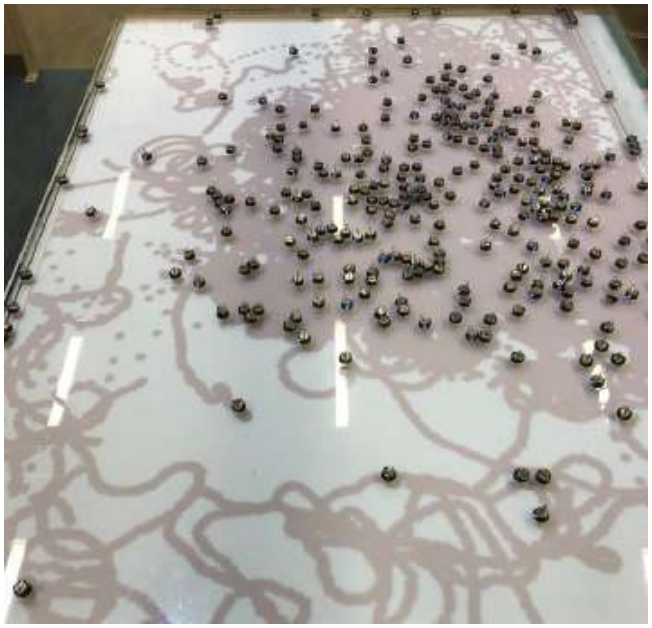


Figure 1.1: Kilobots in the trail avoidance (stigmergy) condition [1]

programming station, which can be used to implement foraging algorithms. A research paper published in 2019 by the Department of Engineering Mathematics from the University of Bristol titled "Testing the limits of pheromone stigmergy in high-density robot swarms"[1] focused extensively on this topic and conducted a compelling experiment. They used a swarm composed out of 324 kilobots, placed inside a 3 x 2 arena and programmed them with a simple final state machine: either engage in a random walk-type movement or if detecting pheromones start an avoidance behaviour. The random walk was implemented by generating a random number which dictates the direction of movement. The avoidance behavior compels the robot to turn either left or right for 0.5 seconds then move forward for 1 second if a pheromone is detected. For identifying the robot position and simulating the pheromones image processing in Matlab 2017b was used. A projector placed over the area creates a dynamic environment by projecting blue light circles onto the pheromone locations. The set-up used in this experiment can also be seen in figure 1.1. The study investigated and recorded the area occupied by groups of kilobots which double in size (ranging from 2 to 400 robots) using classical search algorithms and stigmergic strategies. The conclusion reached was that for normal groups the differences were almost unnoticeable, however, when working with high-density robot swarms more sophisticated exploration algorithms may not bring more advantages than simply using random walk. However, implementing the pheromone trail avoidance algorithm showed an improvement and proved it may be a key factor in other terrain mapping applications.

Secondly, there are a great number of self emerging patterns in nature, such as animal skin or slime mold patterns which can be modeled by reaction diffusion systems. A study issued by the Institute for Perception, Action and Behaviour from the University of Edinburgh and Edinburgh Centre for Robotics [2] took a greater interest in these so-called Turing Patterns and used a robot system to imitate this phenomenon. Introduced by Alan Turing in his paper titled "The Chemical Basis of Morphogenesis" [11], the Turing Patterns describe the way patterns such as spots or stripes appear spontaneously from a uniform and stable state. The reaction diffusion system uses two agents: an activator and an inhibitor with different diffusion coefficients, and it measures the changes in concentration in the participating elements, taking into account the interactions amongst them and the diffusion. The patterns will emerge if the inhibitor is disappearing faster than the activator. The paper considered

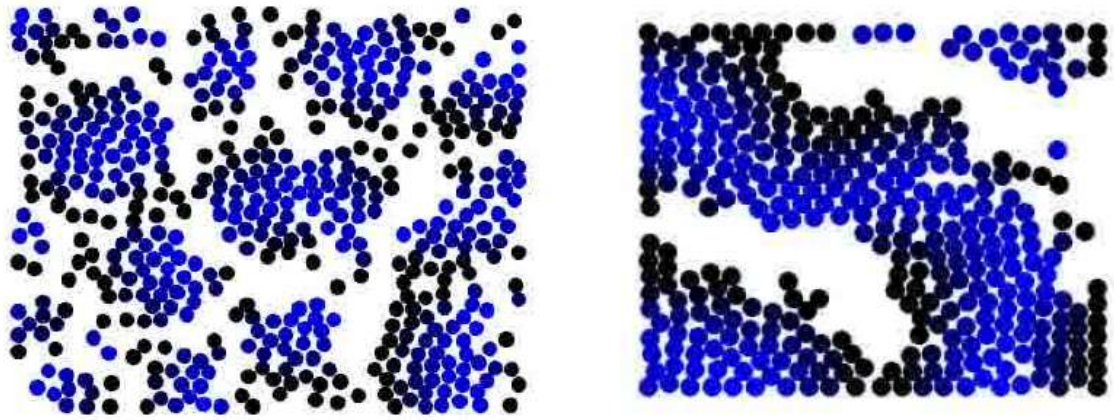


Figure 1.2: Swarms of robots forming spots (left) and stripes (right) [2]

two approaches, a stationary reaction diffusion system, where the robots were arranged in a grid and the concentration value affected the LED light and a non-stationary one, in which the concentration also affected their movement. In figure 1.2 one can observe the experiment conducted in that research paper. A simulation of 500 agents was used to display the formation of spot patterns, and a smaller swarm, of only 300 agents, but using a higher diffusivity constant, led to the emergence of strip like patterns. Although the agents modify their values over time, it was observed that the spot pattern deviates very little, compared to the stripes, which are less stable and require the agents to be more tightly clustered, for a longer period of time. These models are the stepping stone for many useful applications. Only by modifying the system parameters it is possible to have the swarm create any model or pattern, such as a triangle, quadangle or hexagon [12], and also switch between patterns after a specified amount of time. Moreover, using the concentration value and the way it diffuses over time, specific roles can be given to each individual, and even more complex algorithms can be developed.

Thirdly, a vital element in the robotics industry is the interconnectivity between all branches and fields. Keeping this in mind, swarm robotics, although interesting on its own, when used in conjunction with other domains can lead to the creation of more unique and fascinating projects. One such project, titled "A Soft-Bodied Modular Reconfigurable Robotic System Composed of Interconnected Kilobots", was presented in 2019 at the International Symposium on Multi-Robot and Multi-Agent Systems and featured a new concept that merges the notion of swarm robotics with the one of soft robotics [3]. Soft robotics is a research field most active in the last decade, which focuses on the study of biologically inspired machines constructed out of flexible and easily deformable materials, such as elastomers, gels or fluids. They present abilities not found in common rigid robots, for instance squeezing through a narrow opening, growing in dimensions or adapting their shape to fit the environmental conditions. The new concept introduced by the research is a soft body system that is both modular and reconfigurable, and able to change its size and shape to cope with unknown situations [3]. The system is composed out of a group of kilobots organized in a grid whose rows and columns, and therefore the number of modules used, can be manually changed before each experiment. Each kilobot is equipped with a 3D printed holding structure from which small springs of approximately 3.1 cm length can be attached in order to connect all the modules in the grid. This is done for simplifying the control of the system and making it more fault tolerant. Each module senses the distance from its neighbours and has predetermined knowledge of the dimensions of the grid. After using this information to calculate its relative position, the robot will start to estimate local deformation in the desired shape, which can be

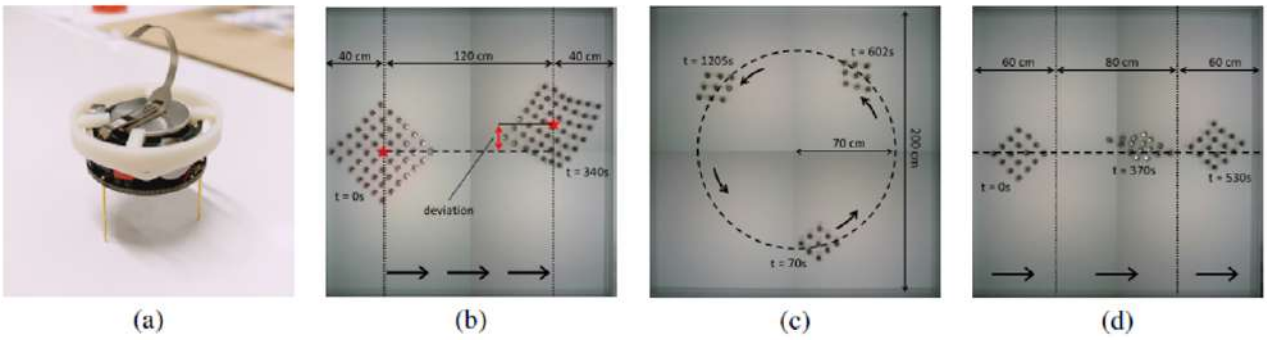


Figure 1.3: Overview of the platform and results of the experiments conducted with the Kilobot Soft Robot.[3]

a normal, extended, or shrunk lattice, and based on that adjust its direction of movement. A series of experiments were conducted to test the Kilobot Soft Robot's abilities, experiments also shown in figure 1.3. In addition to the soft body systems, the experiments also worked with ARK technology [3], for real time position tracker and feedback. The first experiment (b) had the purpose to determine the relationship between the swarm's size and the accuracy of movement. The robot was tasked to move in a straight line, without external coordination, while also trying to maintain the given shape. It was observed that as the number of modules increases, also did the accuracy, the large number of individuals compensating for the inaccuracies. The second experiment (c) tested the ability to follow a predefined curve trajectory. The Kilobot Soft Robot did not store the path in memory, the robot located in the front did however receive feedback from the ARK system, which indicated if the path headed towards left or right. The last experiment examined the system's performance to change its shape from a normal to a shrunk grid while also moving in a straight line. The results established the soft-bodied robot can alter and return to the original shape with success. The Kilobot Soft Robot is an interesting study which combines two different robotics fields and brings into question the advantages of soft links versus rigid connections.

Lastly, as mentioned before, in swarm robotics the conventional goal is to use local interactions between robots and program them to self-organize and form different predefined shapes [13]. All the knowledge about the shape and the desired positions is stored inside the memory of the robot. Nonetheless, another approach exists. With colloidal particles in mind, which convert energy from their environment such as light, heat, electric or magnetic fields into propulsion forces, a team from the Department of Chemical Engineering from the University of Michigan proposed a new concept in which the information about the desired outcome is stored in the design of the system's structure [4], more specifically, its the physical relations between elements that control the behaviour. In order to imitate the moving particles, the researchers used robots linked together into closed loops, limiting the robot's movement to only the 2 dimensional plane of the loop. These chains of robots can be programmed to morph into different configurations by sequencing the motor's orientation. There are six parameters that affect the mechanical interactions and allow programmability and also ensure the system's scalability: the loop's size, motor orientation in relation to the loop, relative strengths of each motor's propulsion force, relative motor sizes, the stochastic forces that act on the motors and internal pressure within the loop.

Several experiments using Kilobots were conducted to demonstrate this morphing behaviour

[3] Augmented Reality for Kilobots [8]

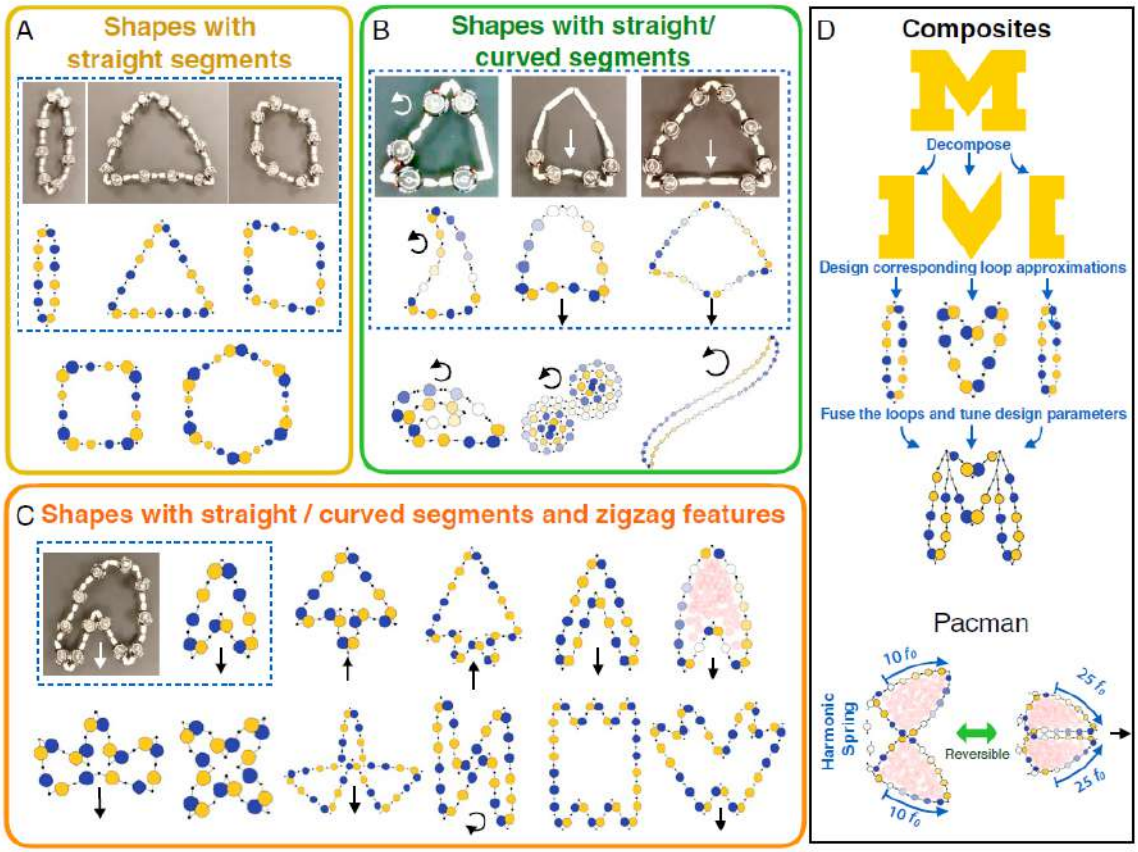


Figure 1.4: Different configurations obtained with a Kilobot system [4]

and in addition to that, demonstrate the system's capability of dynamic behavior, proved by the formation of a closing and opening griper, and complex static behaviour, determined by the formation of the letter "M" out of three different loops. The most simple design which imitates a colloidal motor has the robots connected with wooden sticks, one between every two kilobots and each kilobot programmed to propel forward. In this design, the motor can exist in two states of orientation. When two robots linked together face opposite orientations, the propulsion forces stretch the shape into a straight chain. This also called straightening force can be used to create forms with straight segments such as triangles, rectangles, or hexagons. Furthermore, one of the robots in the kilobot - hinge - kilobot formation can be replaced with a wooden stick, meaning now the motors can exist in one more additional stage, a passive stage. This has the effect of introducing a new curving force whose value affects the degree of curvature of the edge and creates curved shapes. Another force used in creating the shapes is called Notching Force and appears when neighbouring segments attempt to fold beyond what is physically permissible, leading to foldings and zigzag features, such seen on arrow heads or stars. Moreover, all these loops can also be linked together to form even more complex structures. The griper was formed by joining two triangle shapes at one of the vertices. By switching the motors between the passive and active states the effect of griping and tearing was obtained. The letter "M" was formed by fusing two straight segments with a loop obtained with the use of notching force. Some examples of the shapes generated in this research can be seen in figure 1.4.

In conclusion, there are a plethora of studies interested in swarm robotics, self organization and the way individuals manage to rise above their capabilities and perform complex tasks. We have discussed several of them but we have only scratched the surface of this vast domain. It is to be expected that as technology evolves, so will the prospects of using collective systems in real life applications.

Chapter 2
Theoretical concepts

2.1 Swarm Behaviour

Hive intelligence is a scientific branch with roots deeply implanted into the way biological systems, made out of seemingly unimportant entities, emerge in nature, evolve and behave as if they were a much bigger organism, and perform tasks that were previously impossible for them. For instance, we look at fish, bees or ants, small and vulnerable living things that have a very small chance of survival if they live alone, but when organized in groups they are able to gather food efficiently, build impressive and safe shelters to protect themselves from external conditions and in some cases even attack predators much bigger than them. This is also the case of robotic systems, formed of a large number of autonomous robots that have a minimal set of capabilities such as limited communication or freedom of movement, but which are highly useful in applications such as collaborative search [14], transportation [15], unknown terrain exploration and mapping [16], pattern formation [17, 18] and many more.

This fascination for nature lead many researchers to try and recreate biological behaviors. A well-known example is Craig Reynolds's computer model for coordinated animal motion created in 1986. In that model, *boids* (a generic name given for the entities in the system) start from an apparently random movement and after a short period of time start to act in a coordinated manner, comparable with a flock of birds. Each boid has a sense and a direction and can also react to neighbours in close vicinity (a circle with a fixed radius). The flocking action is generated from three steering behaviours that describe the way each individual chooses his direction, based on the position and velocities of nearby boids. These principles are *separation*, in which the boid steers away from local neighbours in order to avoid collision, *cohesion*, where the boid steers towards the average position of local flockmates and *alignment*, where the boids steers towards the average heading of adjoined boids. A simulation of this model compiled in `C++` is presented in figure 2.1, where a flock also tries to avoid an obstacle in the middle.

Eventually, a consensus about the principles of swarm intelligence was reached. In order for a system to exhibit collective behaviour it must implement at leas two of the following functions: coordination, cooperation, deliberation and collaboration [19].

Coordination can also be called organisation and refers to the spatio-temporal distribution of individuals in the group. The bees swarms are a clear example of this, where the interaction

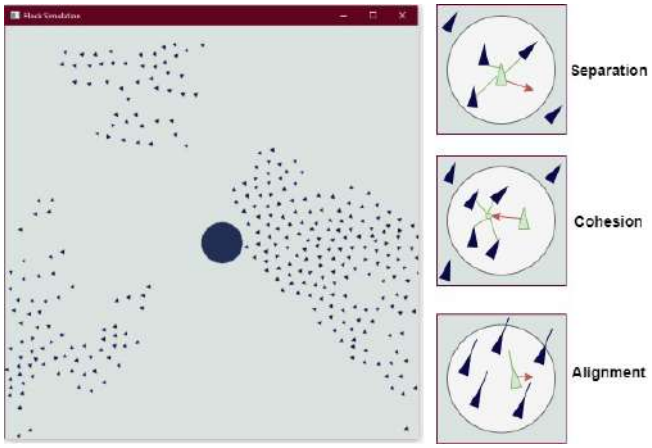


Figure 2.1: Flock Simulation

between individuals create synchronized (temporal organisation) and oriented (spatial organisation) movement toward a specific goal. Another example can be found in the way ants forage for food. They leave pheromone trails from the nest to the food source and back in order to help other ants reach the same source and transport the food faster. These pheromone networks can be seen as spatial organisation. In a similar case, in robotic swarms we can observe this principle in the way robots use local communication to create their own coordinate system.

Cooperation occurs when the group manages to complete a task that was impossible for a single individual to achieve. The entities involved combine their efforts and strength for the greater good of the colony. This can be seen in prey retrieval, when the individual is too weak to transport the item, but with the help of others the task is accomplished. This behaviour can be observed in many species, but the most obvious one are ants, especially the Pheidologeton diversus. It was reported that, in this species, the ants engaged in cooperative transport can lift weights at least ten times heavier than did solitary transporters [19]. Robots can also display this behavior, for instance they work together to form an image or a pattern or can also transport different kind of objects.

Deliberation ensues when the swarm has to make a decision. When faced with multiple opportunities, the swarm must deliberate and collectively decide upon the best one. Particularly, the honeybees decide to gather pollen from the most productive flower parcels due to the dance performed by forager bees returning from different parcels. Deliberation can be found in path-finding or terrain exploring applications.

Collaboration is when individuals are working simultaneously but perform different activities. Leaf-cutter ants are divided based on their size into four main castes that have different jobs. The ants with a head size of approximately 1.6 millimeters are the only ones that can cut leaves that are used to grow a special kind of mushroom, used as food for the group. Ants with a head size smaller than 0.5 millimeters are in charge of cultivating that mushroom [7]. Another example is the Indian paper wasp. They all look the same, however, they do not perform the same task, some of them are charged with finding food, others are builders for the nest and others work as protectors against invaders.

Most of the collective behaviours found in nature are a combination of this four main functions. All the examples were taken from the world of social insects, however, even

though they are the main source of inspiration for many swarm intelligence algorithms, one must not forget that other biological systems also manifest this sort of behaviours and meet the aforementioned functions, systems such as colonies of bacteria or amoeba [20], capuchin monkeys [21], fish schools [22] or even crowds of human beings [23]. Nevertheless, the fundamental objective of swarm intelligence is to create a system that thinks and acts as a bigger entity but is composed out of limited components.

2.2 Kilobots

A `kilobot` is a low-cost robot designed by the Self-Organizing Systems Research Group at Harvard University in 2012 to help the researchers test collective algorithms on a large scale. The name comes from the idea of having "kilos" of robots working together for a common goal.

Technically speaking, they are microbots with a 33 millimeter diameter, an `ATmega 328p` microprocessor that allows them to run different programs, a rechargeable battery, an IR transceiver for sensing and transmitting messages to other kilobots or to the controller, and a movement system made out of two vibro motors. Their specifications are also presented in table 2.1. Compared to other robots used in swarm robotics such as Swarmanoids [24], SAGA robots [25], Colias robots [26] or Khepera robot [27], they do not move with considerable speed, have little knowledge about their environment, and can only communicate with others at a close range. However, the greatest advantage is the cost, since for a smaller price a greater group can be purchased.

Processor	ATmega328p (8 MHZ)
Memory	32KB Flash for user program and bootloader, 1KB EEPROM for storing calibration data and other non-volatile values and 2KB SRAM
Battery	Rechargeable Li-Ion 3.7V
Autonomy	3 - 10 hours continuously and up tp 3 months in sleep mode
Communication	Infrared communication, up to 32kb/s and 1kbyte/s with 25 robots
Sensing	IR and light intensity
Movement	2 vibration motors with 255 power levels, which allow for forward movement at 1cm/s and rotation at 45deg/s
Light	one RGB LED pointed upward
Software	Kilobot Controller software for controling the robot
Programming	C language with WinAVR compiler
Dimensions	diameter: 33 mm, height 34 mm

Table 2.1: Technical specifications of the Kilobot

Figure 2.2 depicts a 3D model of the kilobot taken from the official site. The component labeled 1 is the 3.7 Volt battery. Component 2 is the jumper used to turn ON and OFF the robot. Part 3 is the two motors used for motion, part 4 is an RBG LED, which can be used to signal different states of the robot, element 5 is the light sensor that can be utilized in light-based algorithms. Component 6 is the serial output header that can be connected to the overhead controller and used to send data to the computer for debugging purposes. Part 7 is a direct programming socket for loading firmware to the microprocessor unit of the robot. Part 8 is a charging tab that can be remove if wanted. Element 9 is the infrared LED transmitter

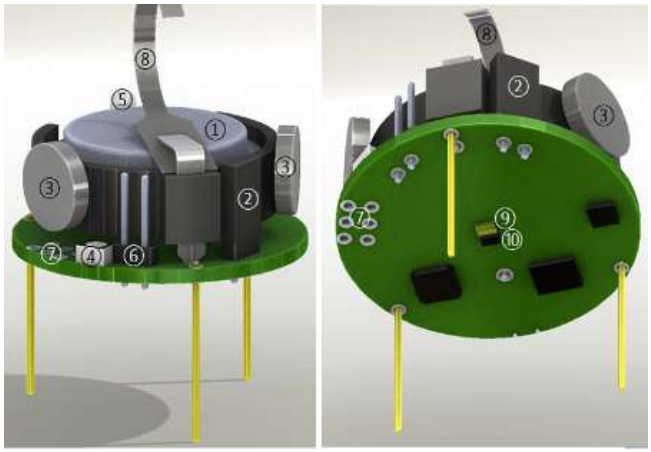


Figure 2.2: Kilobots Components [5]

for sending IR signals and 10 is the infrared photo diode receiver, that receives messages from other robots or from the controller.

2.2.1 Movement

The movement of the robots is based on the slip-stick phenomenon. Stick-slip can be described as the alteration between two surfaces sticking one to another or sliding over each other. At the contact surface between two objects, there is a friction force resisting the relative motion between them. There are two types of friction: static and kinetic. Static friction is caused by the molecular bonding that occurs when entities are in contact, whereas the kinetic one arises because of the roughness of the surface which impedes the motion. The dynamic friction coefficient is relatively constant, not depending on the velocity. The static coefficient, however, increases with the passing of time (the longer the objects are in connection the higher the coefficient will be). This is why the stick-slip phenomenon occurs. In order to set an object in motion, a force greater than the static friction must act upon it, but considering the fact that the static coefficient is bigger than the kinetic one when the object starts to transition to movement, a sudden jump in velocity appears. The object we want to move is the kilobot and the force acting on it is the force generated by the vibrating motors. The motors are placed centered on the sides of the robot as seen in figure 2.2, therefore just using one motor leads the robot to turn around his vertical axis in a direction depending on which motor was activated. If both motors are switched on, the robot has a forward motion. The motors can be individually and manually controlled and have 255 different power levels. This enables the kilobot to move approximately 1 cm/s and rotate approximately 45 $°/s$ [28]. A big disadvantage to this movement system is that there is no way to estimate change in position over time and thus, moving precisely for long periods of time is difficult. Another limitation is the environments in which they can operate. In order to defeat friction they need a smooth and flat surface, for instance a white board.

2.2.2 Communication

In many collective behavior algorithms, robot-to-robot communication is of paramount importance. Without the ability to sense one's neighbours or measure the distances between them there is no way to imitate the interactions amidst entities found in nature. Located in the center of the kilobot and pointing directly downward are an IR transmitter and receiver. This allows the robot to pick up signals equally from all directions. Moreover, both the receiver

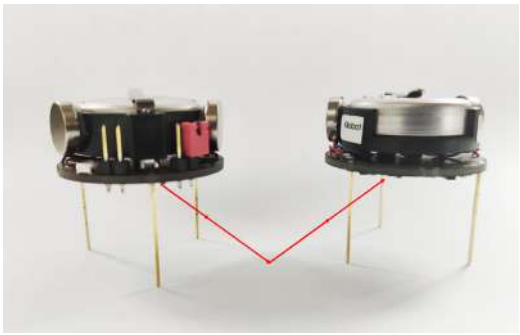


Figure 2.3: Image showing the reflection path of robot communication

and transmitter are wide-angle, with an angle of 60° from the robot's vertical axis [28]. As seen in figure 2.3, the information sent by the robot is reflected on the working surface and can be caught by any robot in the proximity. Messages are conveyed by pulsating the IR LED in accordance with the standard line coding technique. Using this simple communication principle, robots can communicate at rates of 30 kb/s with robots up to 10 cm away [28].

However, using the same channel for communication means that there is a big possibility that at a given time more than one robot will want to transmit a message. A solution to this problem is using the standard carrier sense multiple access with collision avoidance (CSMA/CA) method. The main principle of carrier sense multiple access is the requirement that each robot first checks the state of the medium before sending a message. If the state is idle, then the robot can begin to send data into the channel, otherwise, it waits until the channel becomes idle. However, due to propagation delay, there is a chance that two or more robots begin the communication at the same time. For example, robot 1 checks the state of the channel, finds it idle, and stars transmitting. However, the transmission is not instantaneously, and by the time the first bit of data is sent another robot found the medium idle and started to send data. This is called a collision between two nodes. A method through which CSMA/CA avoids collision is interframe space. That means that when a robot detects the medium to be idle it does not begin sending data immediately, but waits a period of time (interframe space or IFS). After this interval, the robot inspects the medium again and only if it is still idle it starts the communication.

During communication, the receiving robot also measures the intensity of the IR light. Infrared light is electromagnetic radiation with wavelength between 0.74 — 10^3 micrometers and a frequency between 300 GHz – 430 THz. Its intensity also depends on the distance: the farther away the transmitter is from the receiver, the more the intensity of the light will decrease. This indicates that by storing a table that links lengths to intensity of infrared radiation in memory the robots can be calibrated to estimate the distance from one another. In reality, the intensity of light can also be affected by noise or measuring errors which manifest in differences of about 2 millimeters between the estimated distance and the real one.

2.3 Kilobot System

2.3.1 Workspace organisation

The system used for this thesis is composed out of 10 Kilobots, an overhead controller, the charger for the robots and the programming environment presented in figure 2.4. The controller is placed at approximately 30 cm above the group of kilobots and connected to the

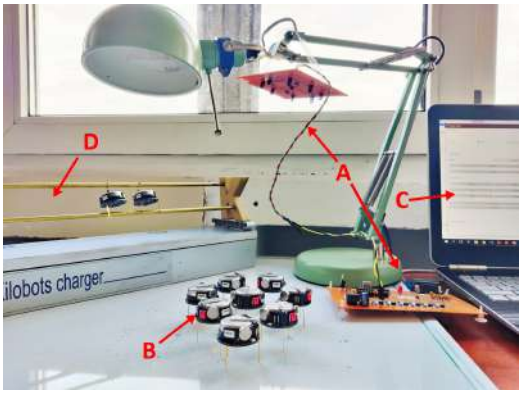


Figure 2.4: Picture of the work station, including the overhead controller (A), the group of 10 kilobots (B), the control station (C) and the charger (D)

computer through a USB to RS-232 adapter. The kilobots are placed on a 85 cm x 65 cm white erase board because of its smooth surface and reflective proprieties which will help with the robot movement. As stated before, the robots communicate using infrared light, and the communication is dependent on the distance between the transmitter and the receiver. As the distance increases, the power of the signal decreases. Therefore, because the kilobots need only an impulse to recognize the message received, it is recommended to use an external light source, so that the power of the signal exceeds the threshold of the photodiode. In order to receive as much light as possible they are placed near a window and during night time one can also turn on the lamp.

Each robot has a 3.4V 160 mAh lithium-ion battery. This battery can power the robot for 3 and up to 24 hours of functioning, depending on the activity level. When the battery is depleted it can be recharged by connecting the legs of the robot to a positive 6V voltage and the charging tab to the ground. To simplify the process and to minimize the risk of inverting the polarity and damaging the robots a new charger was designed [6]. Made with a 12V Voltage Adapter and an adjustable step-down power supply module, it maintains a stable 6V voltage and offers short circuit protection; the 3D printed parts offer stability and allow up to ten kilobots to be charged simultaneously from 0 to 100% battery level in about 4 hours.

2.3.2 ATmega328p microcontroller

ATmega328p is a microcontroller chip part of the megaAVR family created by Atmel. It is a high-performance integrated circuit based on AVR enhanced RISC architecture. As features, the microcontroller has a 32KB Flash memory with read-while-write ability, 1KB of EEPROM memory, 2KB SRAM, 28 pins, 32 general purpose working registers, internal and external interrupts and a maximum working frequency of 20Mhz [29]. Moreover, the chip has an instruction set of 131 instructions, and by executing them in a single clock cycle, the device is able to achieve up to 20 MIPS (million instructions per second) throughput at 20MHz. The most useful elements of a microcontroller must be the peripheral features with the help of which various different applications can be implemented. Some of them are [29]:

- **Two 8-bit Timers with separate prescaler and compare mode**
- **One 16-bit Timer with prescaler, compare and capture mode**
- **Six PWM Channels**

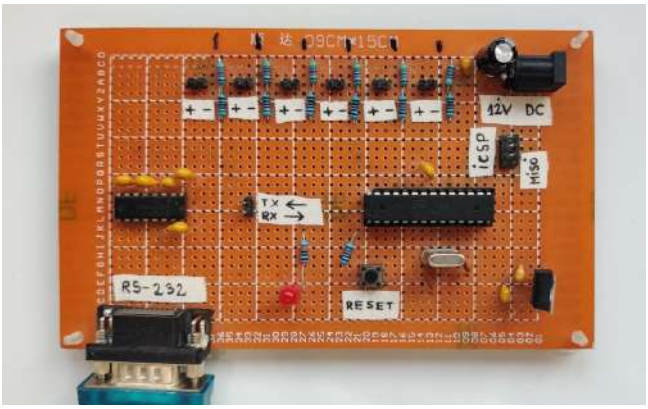


Figure 2.5: Picture of the overhead controller [6]

- **One 8-channel 10-bit Analog to Digital Converter**
- **UART, SPI and I²P Serial Interfaces**
- **Programmable Watchdog Timer with separate On-chip Oscillator**

As a special aspect, the ATmega328p can be placed in five different sleep modes in order to save energy: Idle, ADC Noise Reduction, Power-save, Standby and Extended Standby. At 1MHz, in active mode the device consumes 0.2mA and 0.1µA in power-down mode. Also, it can operate between -40 and 80 Celsius degrees and at a voltage between 1.8 and 5.5V. ATmega328p is used both on the kilobots and the overhead controller as the central unit of processing.

2.3.3 Overhead Controller

In regards to updating or programming the kilobots, using a cable for each bot is not only time consuming, but also very ineffective, especially for large swarms. Therefore, the robots can be programmed via the IR communication channel. This is done using a controller that converts the information received from a computer through a USB - RS232 cable into IR instructions. Placing the over head controller at a height of about one meter ensures the fact that all robots inside a one meter diameter circle, beneath the controller will be able to receive commands. The OHC will program all the robots in the group at a fixed rate, independent of the number of robots. To interface the over head controller with the computer, a graphical user interface app, provided by the Kilobotics team, called KiloGUI is used. Although the team that markets the robots also offers the controller, it was considered not to be an essential purchase. Instead, the controller used for this project was designed and manufactured by a recent graduate of the Faculty of Electronics, Telecommunications and Information Technology [6]. The components used for the Overhead Controller are: the ATmega328p microcontroller, the MAX232 integrated circuit, for converting the signals used by the RS232 standard into TTL logic levels used by the microcontroller, L78S05CV voltage regulator from 12V to 5V, a reset button, a LED, used for checking if everything is working accordingly and a header that can be used for serial transmission. For sending the commands to the robots, the message must be converted to a binary sequence the will be sent to a board of infrared LEDs. The board has eight LEDs that will turn on if the symbol received from the overhead controller has a value of "1" and turn off for a value of "0". Although one board is able to send messages on its own, for a better coverage, six identical boards were assembled.

2.3.4 KiloGUI

`KiloGUI` is a free software component for using the overhead controller to upload new programs on the robots and execute basic operations such as checking the voltage level, switching the LED on and off, or placing the robots in sleep mode. To demonstrate the process we take the simple case of making the robots blink red and blue consecutively. Firstly, we compile our program, which results in a `.hex` file; we then choose it in the Program box. Secondly, the robots must jump to their bootloader and be ready to receive new instructions. This is done by clicking the Bootload button and waiting until the robots start blinking their LED blue. The next step is to click Upload and wait for the program to be fully transmitted, the moment in which the robot will start to blink green twice per second, meaning the operation was successful. To run the program we just click the Run button and watch the robots blink red and blue. All the functionalities of KiloGUI are:

1. `FTDI/Serial` : For selecting the interface to connect the overhead controller to the PC, for our experiments we used Serial Communication.

2. `Program - [select file]` : For choosing the ".hex" file (produced after compiling a C program for the kilobots) that will be sent to the robots.

3. `Bootload` : As explained before, this command makes the robots jump to their bootloader in order to receive new instructions. Clicked once leads the robots to turn their LED blue almost immediately, clicked twice and the robots start blinking, indicating they are ready for the upload step.

4. `Upload` : Send a new program to robots in bootloader mode. While they receive the commands, the robots alternate between a green and a blue light. Once the upload is finished, the led will blink green indicating the success of the operation, otherwise, it will maintain a blue light.

5. `Reset` : Causes the robots to jump to the starting point of the program, reset all flags and states and remain idle until further instructions.

6. `Run` : Begin the execution of user program.

7. `Pause` : Places the robots in a idle state, but preserves the variables of the program so that when they run again the program can resume from where it was interrupted.

8. `Sleep` : Switches the robots to a low-power state called sleep mode. The led will flash white once every 8 seconds. Note that the robots must be placed in sleep or pause mode for them to charge.

9. `Voltage` : Helps to check the level of voltage by turning the LED blue or green if the battery is fully charged and yellow or red if the battery needs to recharge. It is to be taken into consideration that this mode only displays an approximation. It was observed through experiments that if the robot has a low battery it starts to behave poorly and it affects its decision-making abilities, this being a better indication that the robot should be recharged.

10. `LedToggle` : Toggle the LEDs of the controller, its usage is for verifying that the connection between the PC and controller is working accordingly.

11. `Serial Input` : By connecting the kilobot with a 2-wire serial cable to the controller it is possible to display messages from the robot. It is helpful in debugging.

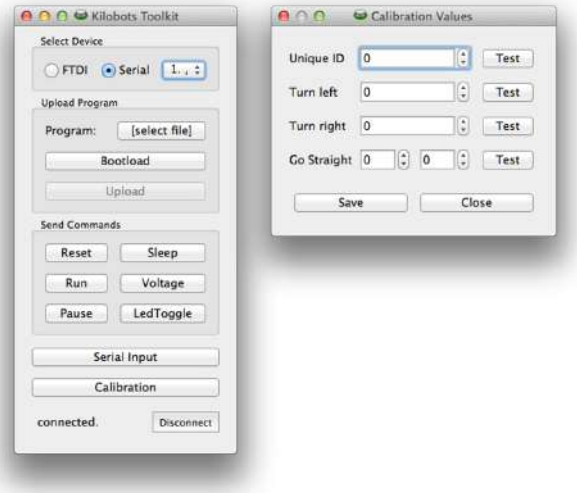


Figure 2.6: Picture of KiloGUI app and the Calibration Menu [5]

12. `Calibration` : Because not all robots are manufactured the same and some differences may appear and also because the working surfaces may vary from application to application, the PWM value needed for controlling the motors cannot be the same for all robots. That being the case, the robots can be independently calibrated using this command. Furthermore, a unique identifier (UID) can be allocated to each robot for better awareness of an individual in the swarm.

Calibration is extremely important for the movement to happen smoothly. Especially in the forward motion, when both motors have to be activated at the same time, if one of them is not as powerful, the tendency will be to move not in a straight line but rather in a circular manner. Moreover, considering the different terrains on which the robots can operate, the same values that allow a robot to move effortlessly on a surface will make it move almost chaotically or even not at all on another one. To avoid that, the robots need fine-tuning. In `EEPROM` memory there are four 8-bit values that describe the duty-cycle that can be used for the motors when the robot is moving: left, right, straight-left and straight-right. These values can be calibrated using KiloGUI. In table 2.2 are presented the values we used in the experiments. We also gave each robot an ID. One can observe the fact that the values have almost no correlation between each other, they were chosen through testing and analyzing to see what fits best for every robot.

Unique ID	**Turn left**	**Turn right**	**Go straight**
1	72	63	65 - 60
2	75	68	67 - 66
3	71	65	62 - 58
4	73	70	63 - 61
5	70	64	58 - 60
6	75	68	66 - 58
7	81	72	72 - 60
8	72	67	65 - 71
9	75	68	66 - 64
10	75	71	62 - 58

Table 2.2: Motor calibration values

2.3.5 Programming Environment

The kilobots are programmed using embedded `C` and for an easier control a special library, provided by the same team that manufactured the robots is used, called Kilolib. A simple way to start experimenting with the robots is using the Web-based Editor from the kilobotics website. It uses the Amazon serves for compilation and Dropbox for storing the files, so anyone with a Dropbox account can write and compile programs. It also comes with an example file `united.c` with the basic loop and set up definitions. However, the web editor comes with a few disadvantages such as dependence on a connection to internet, the compilation time or the small programming window. It is not recommended for programming the kilobots on a regular basis.

For this project, all the code was written using Visual Studio Code and compiled locally using the `avr-gcc` compiler.

2.3.5.1 AVR-GCC compiler

In general, when we write code for different projects we do not take into consideration the type of computer that we use. This is because we work with high level programming languages, such as `C`, `C++` or `Java`, that are more or less independent from the instruction set the machine on which the program was written uses. However, when executing the program or uploading it to a microcontroller all those details become important. Therefore, we need a compiler to convert the high level code into instructions the machine can understand. AVR-GCC does exactly that, it takes code written in `C` language and creates a binary source file that can be uploaded into an AVR microcontroller.

The compilation process can be divided into four parts: Preprocessing, Compilation, Assemble and Linking. Pre-processing is when all the headers and libraries (`#include`) are added to the source code and also, when all the macros (`#define`) are expanded. The compiler then takes the file and turns it into assembly code for a specific processor that will also be later transformed into machine code in the Assemble part. Finally, the linker will link the object code resulting from the previous steps with the library code to produce an executable file. To upload the file to a microcontroller, an additional step is needed: converting the ELF program (Executable and Linkable File) into a hex file. The upload can be done using **avrdude** (AVR Downloader Uploader), an easy to use program for reading or writing all chip memory types, like EEPROM or flash, of Atmel's AVR microcontrollers. For this project we encapsulated the compilation process into a *makefile* and we used KiloGui to send the resulting file to the robots. A diagram for the compilation process is also presented in 2.7.

2.3.5.2 Kilolib

Kilolib is an essential component for programming the robots. It is an open-source `C` library, also created by the kilobotics team, that facilitates robot control by encapsulating all operations that directly access the microcontroller's registers and resources into separate functions. This is a tremendous help for the user, for it is no longer needed to understand low-level code to start programming and, also, the resulting project will be much easier to understand. Documentation where all the functions and their usage is presented is also available online on the kilobotics

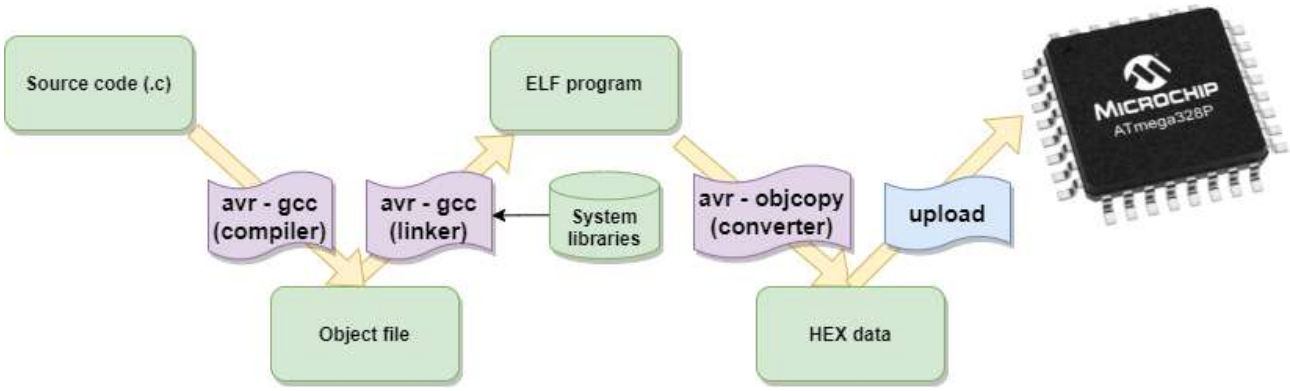


Figure 2.7: Compilation Process

website [1]. The library has fourteen functions, nine variables and two data structures presented as follows:
The functions:

- `void delay (uint16_t ms)` :
  Pauses the program for a certain period of time. As an argument, it receives a 16-bit unsigned integer value that represents the number of milliseconds in which the program is paused. During this time, the processor cannot complete other tasks, therefore, instead of using the delay function it is better to use timers (`kilo_ticks`) .

- `uint8_t estimate_distance(const distance_measurement_t * d)` :
  Returns the distance in millimeters based on the strength of the received message. The only parameter is a pointer to an object of type `distance_measurement_t` in which signal strength measurements taken during message reception are saved. Using a distance calibration table, stored internally on the EEPROM memory of the kilobots, the function estimates the distance between itself and the robot that sent the message.

- `uint8_t rand_hard() ; uint8_t rand_soft(); void rand_seed(uint8_t seed)`
  All three of these functions can be used to generate a random 8-bit number. `rand_hard()` is a hardware number generator. The microcontroller of the robot has a circuit inside called an analog-to-digital converter or ADC. The function takes the robot's own battery voltage level, uses it as input for the ADC, and extracts a seemingly random number from the least significant bit by applying the Von Neumann's fair-coin algorithm. The algorithm is based on tosing a coin two times. If it lands on head (associated with the binary value of 1) and then tails (binary value of 0) the output will be 1. If it lands on tails and then heads the output will be 0. Otherwise, (heads-heads or tails-tails) the process will be repeated. By its nature, this function is slow. For a faster solution `rand_soft()` can be used. This function uses rand_seed() to generate a seed and implements a linear-shift-register to generate a pseudo-random-number.

- `int16_t get_ambientlight()`
  This function returns a 10-bit measurement that represents the amount of ambient light detected by the photodiode from the kilobot. Considering the fact that all the measurements in the kilobot are performed using the same analog-to-digital converter and the ADC requires a certain amount of time to change the source of the input, it can

[1] https://kilobotics.com/docs/index.html

happen to receive a message at the same time as the light is measured. Therefore it is possible to get either an eronated distance estimation or a wrong light value.

- `int16_t get_voltage()`
  Reads the battery voltage level. Int16_t is a variable type that is always 16-bit, regardless of the board used. However, this function (and `get_ambientlight()` and `get_temperature()`) only use 10 bits to convey the information. Therefore, the resulting values will be between 0 and 1023. When the ADC is unavailable the function will return -1.

- `int16_t get_temperature()`
  The function returns a value between 0 and 1023 representing the temperature of the kilobot. It is worth mentioning that the sensor only captures drastic changes in the environment, in the order of 2 Celsius degrees or more.

- `void set_motors(uint8_t left, uint8_t right)`
  This function is used to set the power of the motors. Each motor can be controlled using pulse width modulation or PWM. The function has two positive 8-bit integer values (0 – 255) that represent the duty-cycle of the PWM signal. Strictly speaking, setting a motor to 0% duty cycle turns the motor is off and setting the motor at 100% duty cycles means it will run at full power. To avoid having to guess the perfect values for which the robot will have a good enough movement it is recommended to used as arguments the calibrated values: kilo_turn_left, kilo_turn_right, kilo_straight_left and kilo_straight_right. Also, it is important not to have motor running at full speed (100% duty cycle – 255) more than 2 seconds for it can lead to a permanent damage of the motor. The regular operation values should be between 50 and 90, meaning a between a duty cycle of 20% and 35%.

- `void spinup_motors()`
  When a robot transitions from being stationary (0% duty cycle) to being mobile (more than 10% duty cycle for one or both motors) it must defeat the static friction forces. For that, the motors should by turned on for 15 milliseconds at full power using this function. However, the same effect can be obtained by using the `set_motors()` function in association with `delay(15)`.

- `void set_color (uint8_t color)`
  Used to set the output of the RBG led mounted on the kilobot. The function uses the 8-bit value received as a parameter to set each color channel individually. We have three colors: REG, GREEN, and BLUE with a 2-bit resolution, therefore, four-level of brightness can be set: from 0 (led is turned off) to 3 (the led is at full-brightness). In order to simplify the process of setting the bits manually, the library provides a special data structure called RGB. For example, using RGB(0,0,0) will turn off the LEDs, and RGB(3,0,0) will only switch on the red channel, causing the led to turn on a bright red.

- `void kilo_init()`
  This function initializes kilobot hardware and ensured everything works acordingly. It deals with setting the hardware timers, calibrating the hardware oscillator, configuring the ports, setting up the digital to analog converter, registering system interrupts and initializing the messaging subsystem. It is advisable to call this function as early as possible inside the main function of the program.

- `void kilo_start (void(*setup)(void), void(*loop)(void))`
  This is the function that begins the kilobot functionality. The kilobot's program is divided into two parts: the *set-up* that will run only once and the *loop*, where all the code will

run repeatedly. `Kilo_start` receives as parameters the two functions. The loop event can be interrupted by certain triggering events such as program start, resume, pause or restart, sent by the overhead controller. The set-up must contain data initialisation or anything that can be calculated only one time at the beginning of the program. A very simple example of a code that can run on the kilobots is:

```
uint32_t counter;
void setup() {
   counter = 0;
}

void loop() {
   counter++;
}

int main() {
  kilo_init();
  kilo_start(setup, loop);
  return 0;
}
```

- `uint16_t message_crc (const message_t *msg)`
  Is used to compute the cyclic redundancy check or CRC before transmitting a message. Cyclic redundancy check is a method of detecting errors in received data. This is done by adding a redundant value to the message (a value with no use for the information conveyed). The value is used to determine if the information is the same as the one sent by the transmitter. The CRC algorithm is based on considering a string of bits as polynomial representation ($1011 = x^3 + x^1 + x^0$). The redundant value added at the and of the message represents another polynomial that is divisible module 2 with a control polynomial chosen before. This function must be called before any transmission otherwise the received message will be considered corrupt and will be discarded.

In addition to these functions, the Kilolib library also provides several helpful variables. Some of these variables are declared with the keyword extern and volatile. Extern ensures the variables can be accessed anywhere in any of the files of the program with the condition that those files contain a definition for the variable. Volatile is a special keyword that announces the compiler that the value of the variable may change unexpectedly, at any time, even if the program does not interact with it. The variables are:

- `extern volatile uint32_t kilo_ticks`
  This variable stores the internal Kilobot clock. It is a 32-bit positive integer that is initialized to zero every time the robot's program is reset and it is incremented once every 30 milliseconds, roughly 32 times per second. This variable is preferable to be used instead of the delay() function when programming certain events. Because this variable depends on the internal clock and not on the program it is declared volatile.

- `extern uint16_t kilo_uid`
  This variable stores a 16-bit unsigned integer used as a unique identifier for the kilobots. The identifier can be chosen during the calibration process. Considering the fact that we have 16 bits, that means a maximum of 65535 robots can have unique IDs in a swarm.

- `extern uint8_t kilo_turn_left`
  The variable stores an 8-bit unsigned integer which is the calibrated value for the duty-cycle of the signal applied to the right motor needed for a good left turn.

- `extern uint8_t kilo_turn_right`
  The variable stores an 8-bit unsigned integer which is the calibrated value for the duty-cycle of the signal applied to the left motor needed for a good right turn.

- `extern uint8_t kilo_straight_left`
  This variable stores an 8-bit unsigned integer which is the calibrated value for the duty-cycle of the signal applied to the left motor needed for moving straight. This is different from the turning variable because a lower power is needed for the forward movement.

- `extern uint8_t kilo_straight_right`
  This variable stores an 8-bit unsigned integer which is the calibrated value for the duty-cycle of the signal applied to the right motor needed for moving straight. In order for the robot to move forward, the set_motors function has to receive both this and `kilo_straight_left` variable.

- `struct message_t`
  This is a structure used to store the message. It is composed out of the message payload, which is a 8-bit integer array called data, the message type, which is a value between 0 and 127 for user messages and the message CRC, calculated using the `message_crc()` function.

The Kilolib library is indispensable when working with the robots. It provides an easy-to-understand interface over the tasks that are rather abstruse for a high-level programmer, a task such as sending and receiving a message or controlling the motors. Moreover, the library is open source, posted on github, so that anyone can modify it, improve it, or just consult it to better understand the functionalities.

2.3.6 CoppeliaSim

2.3.6.1 Features

Swarm behavior algorithms have much more impressive results if the number of robots is greater, hence the origin of the name "kilobots". But having "kilos" of robots working in unison can be quite expensive. Furthermore, in the implementation stage, some unforeseen errors can occur, for example, an imperfect algorithm can lead to the collision of two or more robots and can cause irreparable damage, or using a poorly chosen variable for the motor control can affect them permanently. In order to minimize the risk of this happening and to test the algorithms on a larger scale, the best solution is to use a simulator.

There are many robotics simulators, some examples being Microsoft Robotics, Webots, or Gazebo, but for this application, it was decided that the best one is CoppeliaSim, for its great versatily and also because it already had the kilobot model created. CoppeliaSim, previously known as V-REP, is a free simulator with many useful functionalities and tools used by big companies in the industry, including Audi, Kuka, Amazon Robotics and Google [30]. A big advantage is the fact that CoppeliaSim is cross-platform, meaning all content is portable between all platforms (Linux, Windows, Mac), scalable, and easy to preserve. A single portable file contains not only the scene, but all the models used and the control code. Another advantage of this simulator is its highly customizable nature. Through the use of an elaborate API, or Application Programming Interface, every aspect of a simulation can be personalized. The program supports six different coding approaches, all compatible with each other and which can be used separate or at the same time. Thus, one can control a model or a scene using an **embedded script** (scripts written in the Lua programming language; a very easy and flexible method compatible with all CoppeliaSim installations), an **add-on** or the **sandbox script** (add-ons also written in Lua that can start automatically or be called as functions), a **plugin** (plugins are used in combination with the first method, used for

adding special functionalities to the simulation, for example, a fast calculation capability that if written inside a script would take to much time), a **remote API client**, a **ROS node** (this method allows an external program, situated on a real robot or machine, to connect to CoppeliaSim through the Robotic Operating System) or a node talking **TCP/IP** (allows an external application to connect to CoppeliaSim through distinct communication means). For our experiments we selected the first method, writing control scrips using the Lua language.

Lua language is a scripting language developed by the Pontifical Catholic University of Rio de Janeiro in Brazil with the reputation of being one of the fastest scripting languages according to several benchmarks [31]. Moreover, Lua has been largely used in the industry, some notorious projects using the language for their embedded systems include Adobe's Photoshop Lightroom, the classical MMORPG Word of Warcraft and the popular game Enigma. Its strength comes from combining procedural syntax with powerful data description constructs based on associative arrays and extensible semantics and also from its automatic memory management with incremental garbage collection [31]. Another advantage is its well detailed and clear online documentation where one can find not only information about the language itself but also a large variety of tutorials. Considering the syntax, Lua is quite easy to learn. It has a total of 20 defined keywords, multiple commands can be written on the same line (if delimited by semicolon) and data types can be converted at any point in the script, meaning Lua is dynamically typed. The data types accepted are: table, userdata, function, thread, nil, boolean, string, and number; table being the only structured data type. Function blocks are also delimited by keywords, they start with *goto* or *then* and end with *end*, *elseif* or *else*. As a disadvantage, Lua works exclusively as an embedded part of a host application, not having its own main program. In a few words, Lua is compact, fast in execution, easy to learn and free to use, making it a perfect choice for the robotics simulator.

A simulation generated in CoppeliaSim has two indispensable parts: scene objects and calculation modules. Scene objects are visible elements in the scene's hierarchy and view used for building the scene. It is recommended for the objects to be arranged in a tree-like hierarchy. Calculation modules are what make the simulation "come to life" by directly operating on one or several scene objects. The scene objects found in CoppeliaSim are:

- **Shapes** : Shapes are rigid meshes composed of triangular faces, used for body simulation and visualization. They can be optimized for fast collision responses and can be grouped together to form other objects.

- **Joints** : The equivalent of joints in reality are actuators. Joints are used to link scene objects together with one or three degrees of freedom. The types supported by the simulation are revolute joints (that act as a hinge), prismatic joints (for a linear sliding movement), screws, and spherical joints (that offer three degrees of freedom). The joints are used in conjunction with calculation modules such as the inverse kinematics mode or the force mode.

- **Proximty Sensors** : Are used to calculate the minimum distance between the sensor and an object situated in a field of view. The volume of the field of view can be modified into a pyramid, a cone, a cylinder, or a disk.

- **Vision Sensors** : Are camera-type sensors, reacting to light, colors and depth and used to extract information from the simulation scene. They also have an internal filtering and image processing function.

- **Force Sensors** : Are objects that are usually placed as links between other shapes and have the ability to record and react to the forces and torques applied to them. An interesting aspect is the fact that these sensors can also break if the force exceeds a certain threshold.

- **Graphs** : Used to record and display simulation data. Data can be visualized directly (time graph) or combined with each other to display X/Y graph or 3D curves.

- **Cameras** : When associated with a viewpoint allows the user to look at the simulation under different angles.

- **Lights** : Are objects that allow illumination of the simulation scene or individual objects. They also directly influence cameras and vision sensors.

- **Paths** : Are a succession of points in the given space used to define trajectories and complex movements. Paths can also include rotations and pauses in order to better describe a robot's course.

- **Mills** : Represent convex figures that can be customized and used to simulate surface cutting operations such as laser cutting or milling.

- **Dummies** : A dummy is a point with orientation, mainly used as a reference frame.

Some of the objects presented have attributes that can be modified in order to let other shapes or functions interact with them. The object can be set to be collidable, measurable, detectable, rendable and viewable. But without the calculation modules we can only use the program to display different shapes and objects, not to reproduce reality. The main calculation modules are:

- **Kinematics module** : Is used for kinematics calculations, both inverse and forward kinematics, for any type of mechanism. Kinematics, also called "the geometry of motion", is the branch of mechanics that describes the motion of points, objects and systems, without reference to the causes of motion (forces).

- **Dynamics module** : Is responsible for the real-time interaction between rigid bodies and the way objects operate inside the simulation. CoppeliaSim offers four alternatives for the physics engine that can be used: Bullet Physics, ODE, Newton and Vortex Dynamics. All of them have different parameters that can be consulted inside the application and the recommendation is to use and test all four of them to determine which one is best suited for each individual project.

- **Collision detection module** : Is utilized for a fast checking of impacts between shapes or collection of shapes. Considering that the collisions are a critical part of simulations, the calculation must be done rapidly, therefore this module is independent of the dynamics module. In order to accelerate the process, the module uses as data structure a binary tree out of bounding boxes. Also, more optimization is achieved with a temporal coherence caching technique.

- **Path/motion planning module** : Supported by the OMPL library (Open Motion Planning Library) this module handles path planning tasks. The library includes implementations for a large number of planning algorithms such as a derivation of Rapidly-exploring Random Tree (RRT) algorithm

- **Dynamic particles** : Another module supported by CoppeliaSim is dynamic particles that can be used to simulate air flow, water jets, jet engines or propellers. This is important for drone or quadricopter simulations or can be used in concomitance with other robots to simulate a bigger scene, for example, a robot hand painting a fence or creating an image using dye.

The simulator has a very intuitive and easy to use IDE (Integrated Development Environment) shown in figure 2.8. In that figure some of the most important elements of the IDE are highlighted.

1. This is the **workspace**. When the simulator first opens it is blank, but it can be modified to look like a real surface. Also, every model added to the simulation appears in the middle of the workspace

2. Some **objets** that can be used inside CoppeliaSim. We have three robots that exist in reality: NAO, the Quadcopter, (where the dynamic particles used to mimic the air jet can also be observed) and Kuka youBot. All three of them can operate just as they do in reality. The other object shown is created by the user by linking a sphere to a cuboid. When an object is selected their XYZ axis appears, with red indicating the X axis, green for the Y axis and blue for Z. An object can be moved or rotated in reference to any axis. All the objects can be selected and added to the scene from the **model browser**, flagged with number **7** in figure 2.8.

3. This is the simulation **Control Toolset**. From here one can play, pause and stop the simulation. In the picture, the simulation is paused. Four simulation modes can be selected: Real-time Mode, Speed-up Mode, Slow-down Mode and Visualization Mode, where the user can see what the vision sensor sees if one is included in the simulation.

4. Here is an important toolbar. From left to right we have the *Camera Pan* button, which moves the workspace vertically or horizontally, *Camera Rotate* button, *Camera Shift* button, which acts as a zoom option, moving the camera away or near an object, *Fit-to-view* that positions the camera to fit the selected object, *Object/item shift*, used to move an object either to an absolute position or relative to a desired point, and *Object/item rotate*.

5. This is the **hierarchical model display**. Placing an object in the scene automatically adds it to the hierarchical structure. The user can create its own hierarchies, for example, the Cuboid and the Sphere are connected, the sphere being a child to the cuboid object, and acts as a separate object.

6. The left side toolbar is very important to simulations. From here one can modify the physical attributes of an object, like height or weight. By selecting the *Scene object properties* button, one can alter the kinematics and dynamic parameters, and also add scripts to control the objects. The other buttons in the toolbar are used for customization.

2.3.6.2 Kilobot Model

A model for the Kilobot was provided by the K-Team, the team from where the robots were purchased, therefore it is safe to assume the model was tested and used in many other applications. Fortunately, the model is open source, meaning we are allowed to tinker with it, analyze it and understand how it was constructed. In figure 2.9 one can observe the initial

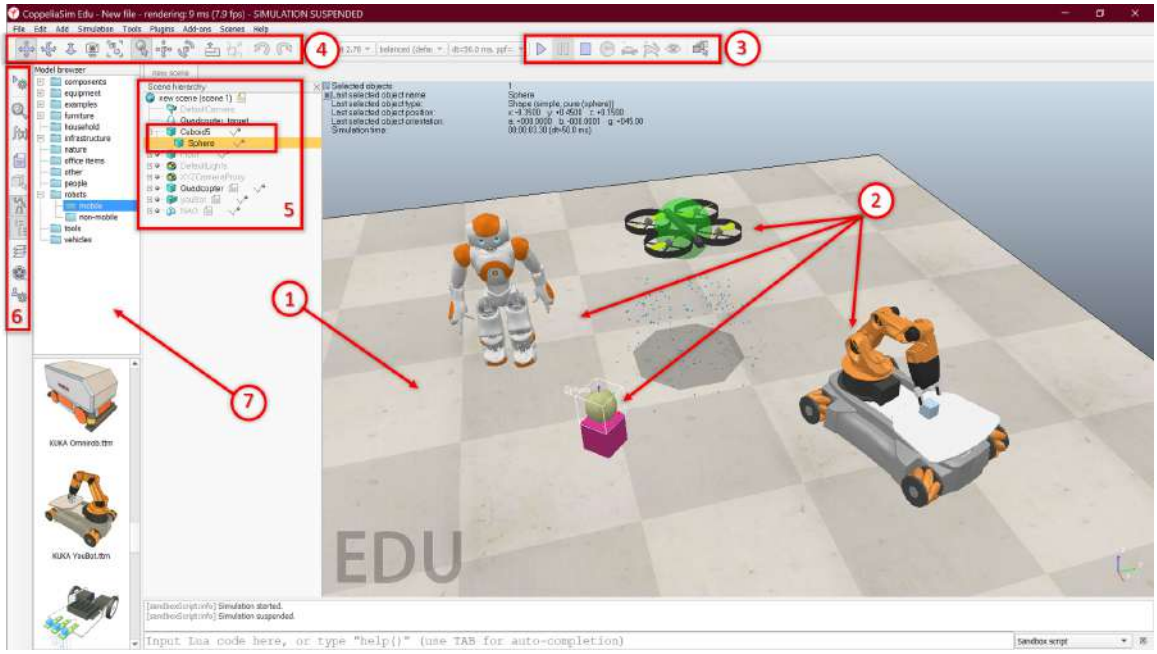


Figure 2.8: CoppeliaSim IDE

Kilobot model, the deconstructed one with all parts categorized and also the object's hierarchy.

The body of the kilobot is made out of seven shapes: the BatHold, short for battery holder, the Base, the equivalent of the robot's PCB, the two motors and three legs. The base, battery holder and motors are only for design purposes, to make the model resemble the real robot. To simulate the LED mounted on the kilobot, the BatHold can modify its color using special functions inside the program. In order for the model to act like its real counterpart, some sensors must be added.

Firstly, we look at the proximity sensor. It is a special object supported by CoppeliaSim, used for calculating the distance between itself and other objects. This particular proximity sensor is a sphere with a radius of 5 millimeters, situated in the middle of the kilobot. It is set to be an Infrared sensor and to detect only other Kilobots. Also, it has a default range of 84 millimeters that can be modified to better suit experiment conditions. The range can be observed in figure 2.9 as the pink circular ring surrounding the components.

Secondly, we have the message sensor, which is a dummy object, a point with orientation used as a reference frame. The way the model simulates messaging between robots is included in the robot's script. Inside the script is a particular array, called `message_tx` that can be used to send data, and an array called `message_rx` used for storing received data and the distance estimation from the sending robot. Every 0.2 seconds the robot updates `message_tx` and activates a special flag, signaling the fact that the message is valid. At every step it also checks for incoming messages, meaning it checks if other MessageSensor dummies are in a 7 millimeter range and if they have valid data. If the conditions are met, the robot updates `message_rx`.

Thirdly, we must consider the movement system. As previously stated, the two motors are only for design and play no part in the way the model moves. The legs too are only for support. If one looks closely, one can observe that above the contact point of the legs with the floor, there are two joints, one for the left leg and one for the right one. Joints are special elements, used to link the leg with the contact point and to operate in a force/torque mode.

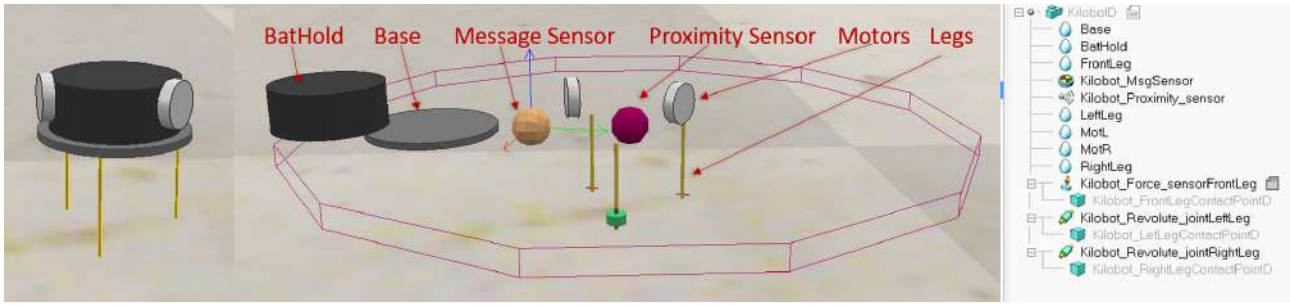


Figure 2.9: Kilobot Parts

This means that by applying a velocity to the joint, the leg, and therefore the robot, will start to move. On the front leg, one can find a Force Sensor that reacts to the forces of the two legs and helps with a smoother motion. The velocity applied to the left and right leg is not random, in fact, the user only chooses a value between 0 and 255 (just as for the real robot) to signify the desired power, and the script takes that value and multiplies it with a predefined motor ration in order to produce an acceptable effect.

Chapter 3
Algorithms

3.1 Edge Following

3.1.1 Orbiting with one stationary robot

Orbiting is a classic example to start and verify that everything is working accordingly. Its objective is to have a robot moving at a fixed distance around a stationary robot. The stationary robot shall be called a ***star*** and the moving robot a ***planet***. This can later be extended to edge following if there is more than one stationary robot.

The star will be emitting a constant message used by the planet robot to estimate the distance between them. The communication range is between 33 mm when the robots are colliding, and 110 mm; so we chose the *orbiting distance* to be 50 mm. If the planet robot senses that its current distance is less than the desired distance it will start turning left until it leaves the orbit; at that moment it will start rotating right. This alternate movement between left and right will cause the robot to move in a clockwise direction around the star. For a smoother movement when the distance is exactly 50 mm the robot will move forward. One can observe the diagram for this code in figure 3.1

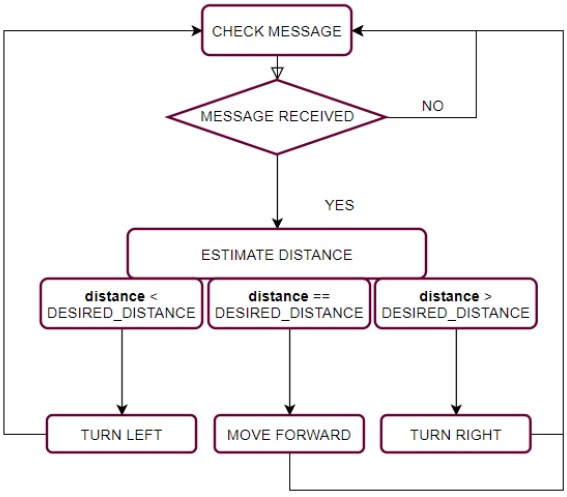
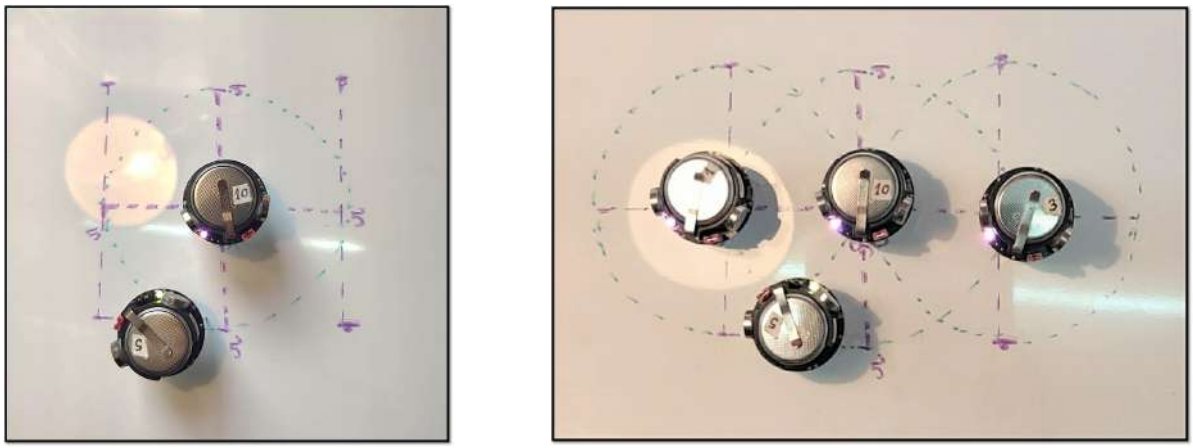


Figure 3.1: Flowchart for orbiting algorithm



(a) Kilobot orbiting a star (b) Kilobot orbiting multiple stars

Figure 3.2: Kilobots engaged in orbiting behaviour

The algorithm was tested with two real robots, the setup is presented in figure 3.2a (The video for this demonstration is also available as a link in the title of the figure). The planet is robot labeled 5 and its orbiting robot 10 at a radius of about 5 cm. The robot took almost 2 minutes to complete a full circle. After numerous experiments, it was also observed that if the planet does not receive a message in time, due to perturbations or errors, the orbit will be compromised, either by the robot moving too far to the left and no longer being able to get back to the orbit or by the robot moving right too much and colliding with the star. The solution was to wait for at least four messages, estimate the distance for every message received, remember the minimum one, and only then make a decision regarding the direction. This leads the robot to maintain a finer orbit but the trade-off was the time, now 3 minutes for a full circle.

3.1.2 Orbiting with multiple stationary robots

The Orbit algorithm can be extended to edge following if more than one stationary robot is used. However, using the previous algorithm unaltered can lead to faults such as the planet colliding with one of the stars. For the new version, each kilobot will have an array to to keep track of all its neighbours. The array will be updated with the current distance every time a message will be received. The robot shall wait for four successful messages, check for the minimum distance in the array, and make a decision accordingly. A demonstration is presented in figure 3.2b, in which a robot orbits three stars for two full rotations, in eleven minutes and thirty nine seconds. If one of the star robots is moved, the planet will adjust its trajectory to better suit the new arrangement. The code for this algorithm is presented in annex A.

3.2 Gradient Formation

An interesting study commonly found in nature is pattern formation due to morphogen gradients. A clasic example is the case of the Bicoid gene that divides the Drosophila (fruit fly) embryo into head, thorax, and abdominal regions. The protein inside the gene is transferred between the cells of the organism and after a period of time the fate of the cell is determined based on the concentration inside the nuclei [32].

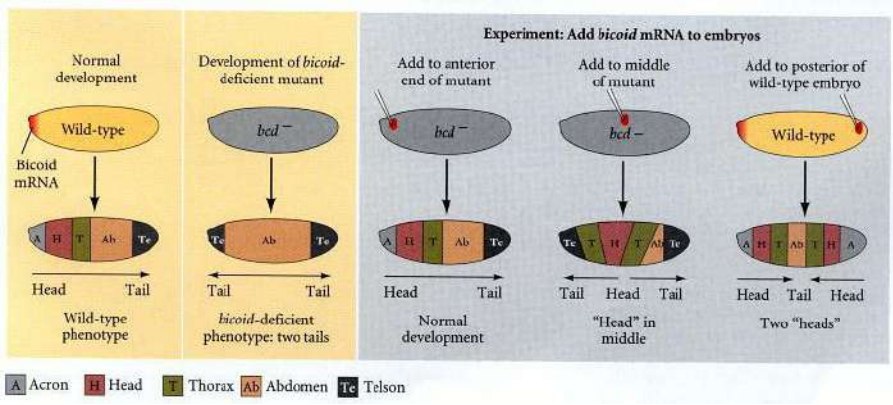


Figure 3.3: Schematic representation of the experiments demonstrating that the bicoid gene encodes the morphogen responsible for patterns in Drosophila

To demonstrate this process an experiment was conducted in which bicoid RNA was injected into bicoid-deficient embryos. It was observed that any location where the gene was injected became the head. Inserted into one side of the organism lead to a normal anterior-posterior evolution, but injecting into the center caused the head to appear in the middle of the embryo and the regions on either side of it to become thorax structures. Also, when injected into both sides of the entity, two heads emerged, one at either end, as presented in figure 3.3 [33]. In general, in morphogen gradients one cell is the source of the chemical that propagates over a surface of cells that record the value of the gradient and use that to determine what part of the pattern to become.

This algorithm's goal is to mimic that behavior. For that, we first chose a robot to be a source and to continuously transmit its gradient value of 0. Any other robot will be listening for messages, record the minimum value they received and set their gradient to be that minimum value plus one. After that, they too will be transmitting their gradient to other robots in their vicinity, creating the gradient effect. The diagram for the algorithm is presented in figure 3.4. In figures 3.5 and 3.6, to visualise the gradient value we associated it with a different color for the LED. Green is used for the source with the value of 0, yellow is for the gradient of 1, light blue is for 2, red for 3, pink for 4 and any robot with a gradient equal or greater than 5 is colored with dark blue. In figure 3.6, just as in the double bicoid experiment, we used two source robots to see how the gradient propagates.

The gradient value can also be used to approximate the distance between two robots. The process comes from networking where the number of intermediate devices such as routers through which a given piece of data must pass in a transmission path, also called the hop-count, gives an estimation of the distance between two given hosts. In our case, the gradient represents the number of times the message was passed from the source to a particular robot. For example, a robot with a gradient value of 5 means the message was passed through at least 4 robots before arriving here, and by knowing the average communication range to be 65 mm we can assume that this robot is at about 65×4 mm distance from the source. This algorithm can be extended to edge detection by recording the maximum value of gradient received from the neighbours for a period of time (the code can be found in annex B).

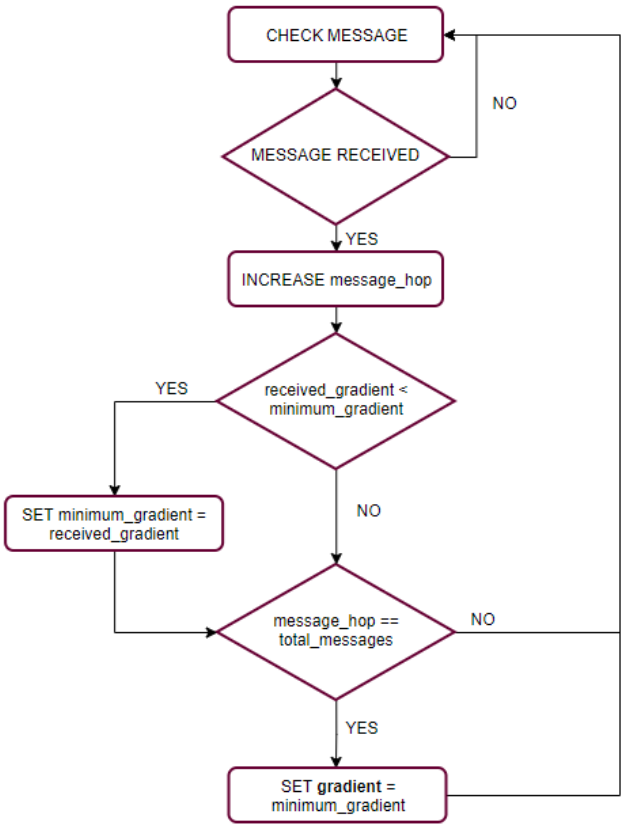


Figure 3.4: Flowchart of Gradient Algorithm

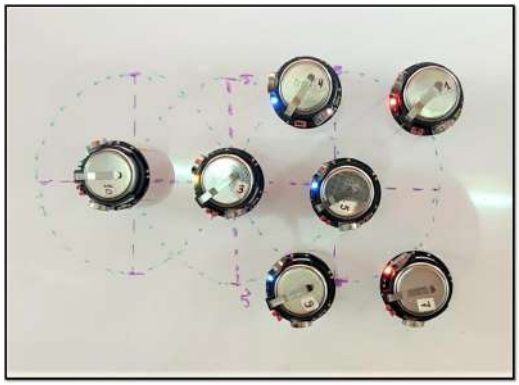
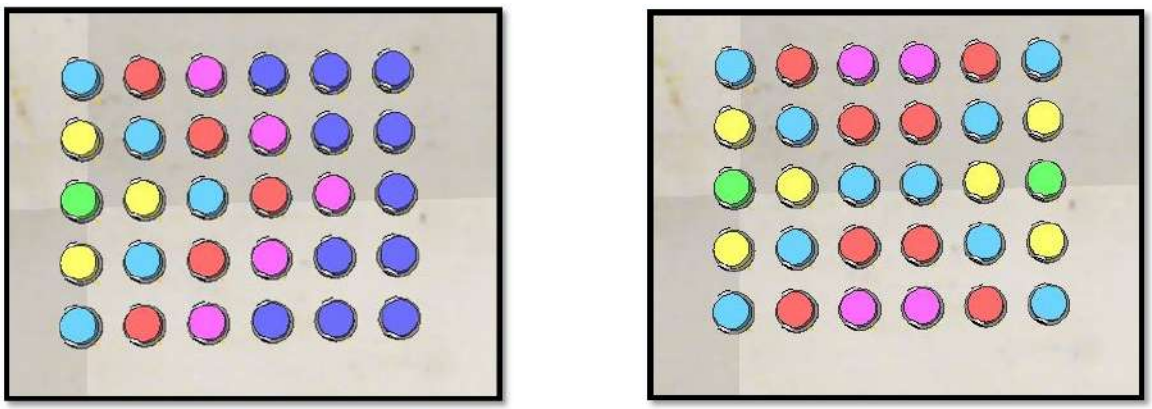


Figure 3.5: Gradient formation with real Kilobits



(a) One gradient source (green robot) (b) Two gradient sources

Figure 3.6: Gradient formation with 30 robots in CoppeliaSim simulation

3.3 Shape Formation 1

Shape formation algorithms must ensure that a group of robots with limited capabilities and local communication can work together to form a predefined shape without external control. This first algorithm combines the previously discussed concepts of *edge following*, where a robot orbits around a larger group by measuring the distances from the robots in its proximity and *gradient formation*, where a source robot sends a gradient value that propagates through the swarm, giving each robot a sense of spatial distance from the source. The main focus of this algorithm is the robot's ability to measure the distance from other robots and based on that, determine if its position is valid (part of the shape) or not. The algorithm was adapted from a previous research project published in 2019 [34]. Each kilobot has stored in memory a matrix that describes the desired shape.

The matrix is constructed as follows (also shown in figure 3.7) :

1. We divide the image into circles with a radius of 3 cm and we assign each circle a unique id in the order that we want the robots to enter the shape

2. We chose 3 robots that will start in the shape, they are called seed-robots and will have the ids 0, 1 and 2, the gradient of 0 and the status *stable*

3. For the remaining robots we chose 2 neighbours and store their index and the distance from them into a structure such as this:

$$(\quad Neighbour_1 \qquad Desired\_distance_1 \qquad Neighbour_2 \qquad Desired\_distance_2 \quad)$$

   We considered a generic distance of 1 between robots, this will later be multiplied with a constant depending on how dispersed we want the robots to be.

4. Then we place all the structures into a matrix; the index of the rows will be the index of the robot plus the value of three. For example, on the first row will be the instruction for the robot number three.

The robots start arranged in a random shape, without any knowledge about their position and with a default id of -1. After they calculate their gradient value the robot with the highest gradient starts orbiting the group (maximum gradient means the robot is on the edge). The orbiting robot is moving until he finds a valid and not occupied point in the shape matrix. At that moment he stops, sets its id and becomes a beacon for other incoming robots. For instance, in figure 3.8, the robot represented with yellow is moving around the seed robots, constantly checking the distance from them and traversing the shape matrix. When it reaches a length of 50 mm from from robot 1 and 2 that means it is on the position indexed 3 in the matrix, therefore the robot changes its id from -1 to 3 and sets its status as stable. When the next kilobot will orbit the group and reach the distance of 70.7 mm (50 * $\sqrt{2}$) from robot 3 and 50 mm from robot 2 it will stop and update its id as 4. This process continues until all the robots have valid ids and the shape is completed.

Thus, the robots have four distinct states. They start in the *wait* stage, where they compute their gradient and check if they are clear for moving, which happens if they do not receive a gradient with a value higher than their own and if a certain amount of time has passed. After that, they transition into the *search* stage where they start orbiting the group.

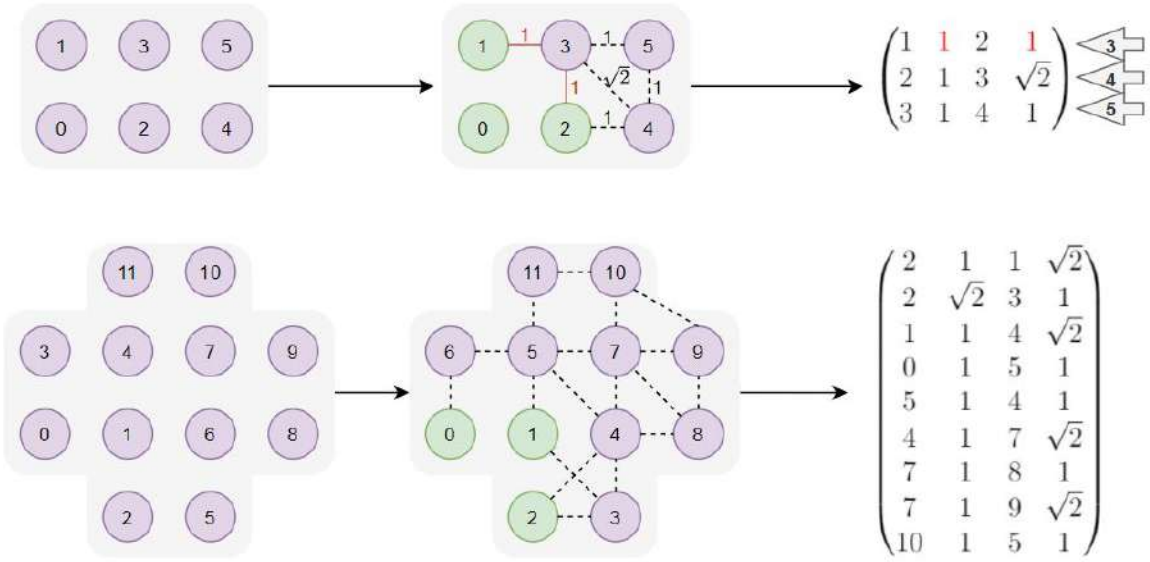


Figure 3.7: Matrix Formation for a rectangle and a cross-like shape

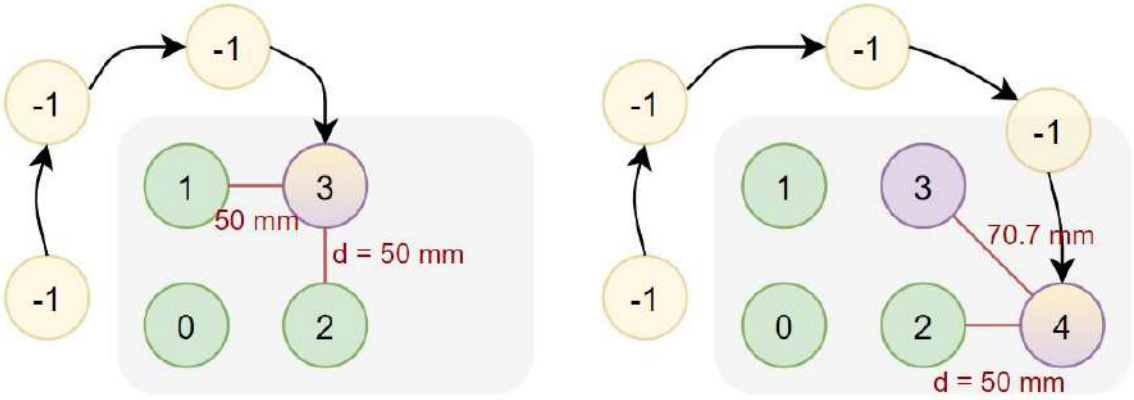
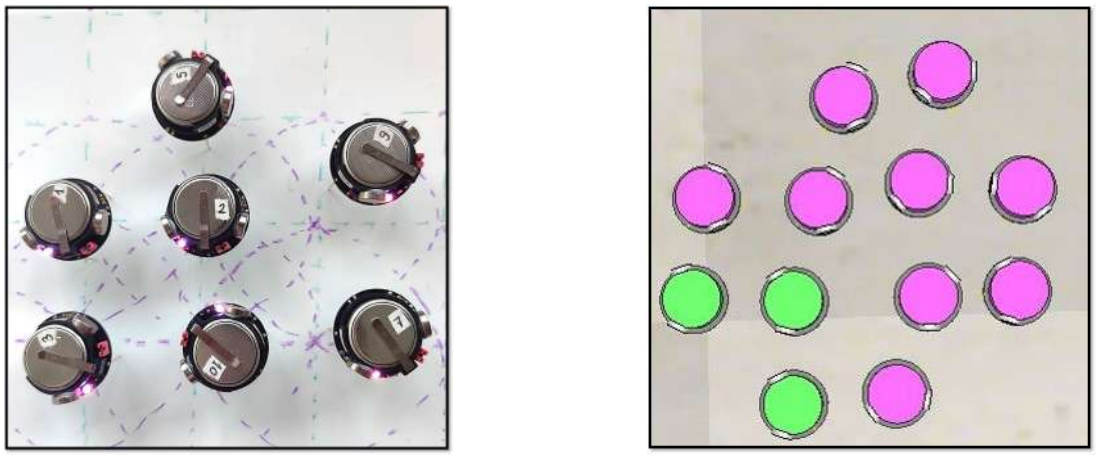


Figure 3.8: Representation of robots occupying position in the shape

The robots keep an array in memory with all their neighbours, including the distance from them, their gradient value, and whether or not they are localised inside the shape. The array is refreshed continuously and any neighbours from which no messages have been received for a predetermined period of time are discarded from the array. They also have stored the matrix containing the instructions for the desired shape and at every step, they check if the distances from their neighbours match any index in the matrix. If that condition is met, they change their status to *local*, stopping the movement and letting other robots know they took that place. For the time being the algorithm uses special colors to differentiate between each state. The LED is off while the robots are waiting, in the orbiting phase the robots switch between yellow if they are turning left, blue for right and red for forward and when they find a stable position, meaning they will have the local status, the LED will have a magenta color. For future developments, another coloring system can be used in which each index in the shape can be associated with a different color, allowing for the formation of even more interesting shapes.

Two experiments are presented in figures 3.9a and 3.9b, with the videos available as a link in the title and the code presented in annex C. In the first experiment, which took thirteen minutes and thirty five seconds the robots arranged themselves in a simplified house shape, composed out of a 2 by 3 rectangle and a robot on top which represents the roof. In the



(a) Kilobts forming a simplified house shape (b) Kilobts forming a cross-like shape

Figure 3.9: Two experiments for shape formation using real kilobots and CoppeliaSim

second experiment, done using CoppeliaSim, the twelve kilobots had the goal to form a cross. The simulation took twelve minutes and forty four minutes, but only because it was run at a higher speed, in real time the simulation would take forty nine minutes. It is noticeable that the shapes are not perfect due to errors in estimating the distance, but they are close to the desired outcome.

Another problem is matrix generation. For easier shapes, the matrix can be calculated by hand, however for complex ones the process is far too complicated and time consuming. That being the case, an algorithm for generating the data structure from an existing picture was developed.

3.3.1 Matrix Generator

First of all, some image processing is needed. An image with a size of approximately 100 by 100 pixels is taken as input. The next step is removing the background. This is usually a difficult task that involves using either complex algorithms or even neural networks. However, for this application, pictures with a simple background, of a completely different color from the object of interest, for example, a plane or star in the sky, were selected. Therefore, it is easy to create a mask by extracting the color of the background from the image. The current image is in RGB format, meaning, it is a data structure with 3 channels, one for each main color: Red, Green, and Blue. For the next steps these channels are irrelevant so, by using the formula

$$Gray\_Image = 0.299 \times Red\_Channel + 0.587 \times Green\_Channel + 0.114 \times Blue\_Channel$$

we convert the RGB image into grayscale, a 2D array where all the pixels have values between 0 and 255. Sometimes, for complex images, it is necessary to apply a median filter to smoothen the picture. The next step is binarizing the image. The process is simple, if a pixel has a value greater than a threshold, the pixel will take the value of 1 and if it is smaller it will take the value of 0. The resulting array will be used for the next part of the algorithm. All the steps taken to adjust the image were written in Matlab and are also presented in figure 3.10.

Second of all, the user must specify the starting point by choosing three points from the

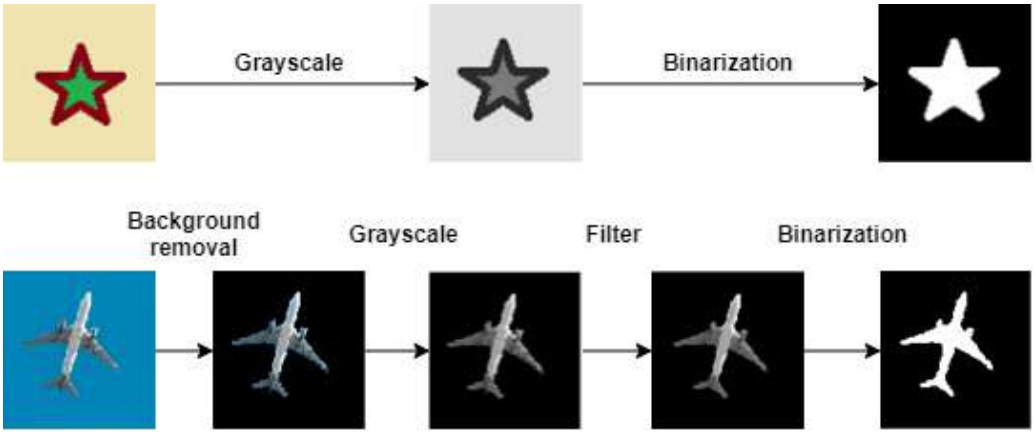


Figure 3.10: The steps taken to modify a given picture

shape that will be localized from the beginning. The essential idea is the fact that each pixel in the shape represents a robot and must have a distinct index. The algorithm uses three main data structures:

- `map` :the initial matrix, received as input from the original image
- `auxiliary_matrix` : of the same size with the `map` array but which has, in the beginning, only the user selected nodes, numbered 1, 2 and 3 and the rest of the elements having a value of 0
- `shape_index` matrix: which is the matrix that will be sent to the robots before starting the shape formation process; its columns will have the form:
  $(Neighbour_1 \quad Desired\,distance_1 \quad Neighbour_2 \quad Desired\,distance_2)$
  and will have as many rows as not null values in the initial matrix.

While there are valid nodes in the initial matrix, one of the nodes is chosen and inserted into the auxiliary array where the distance from exiting neighbours are recorded and inserted into the shape_index array along with a new id, the number of order in which the robot enters the shape. In this manner the gradual generation of the data structure is assured, meaning a robot will not depend on another robot with a greater index, which should get in the shape after itself. As illustrated in figure 3.11 the shape of the plane was converted into numbers representing the ids of robots. A portion of the tail was highlighted in which we can see the starting point, the robots numbered 1, 2 and 3. Also, in the right are the first 57 elements of the shape_index matrix that can be sent to the kilobots to start the formation algorithm. The code for this part was written in `C++` and can be found in annex E.

3.4 Shape Formation 2

The previous algorithm had the disadvantage that every robot had a predefined spot therefore, errors propagate with ease and can alter the resulting shape. On that account, we took a different approach. We defined the shape not as a matrix but as a polygon in the euclidean space. Now, the position of the robots is no longer defined by the number of localized neighbours but by their own coordinates, leaving some space for errors. This idea was first introduced by Michael Rubenstein, the leader of the team responsible for the invention of

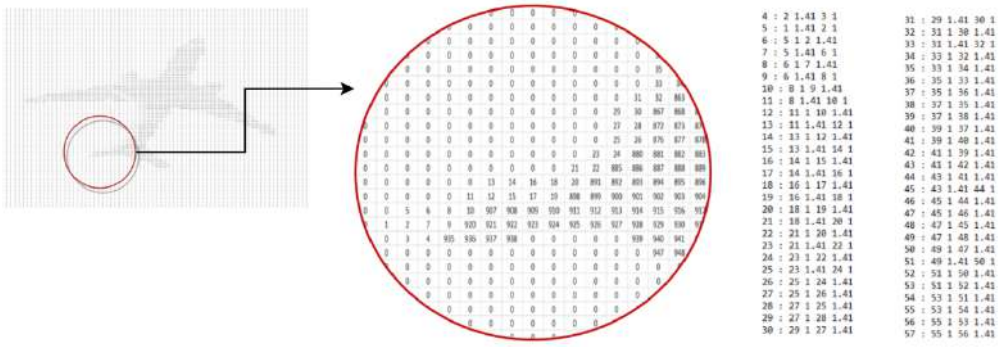


Figure 3.11: Image showing the first 57 elements in the shape matrix

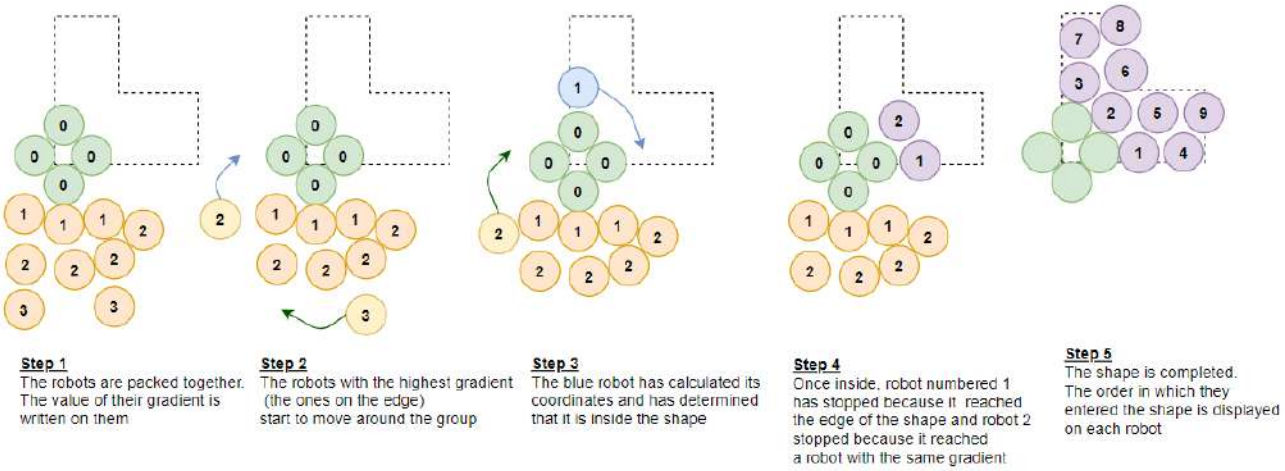


Figure 3.12: The steps of shape formation

kilobots, in his scientific paper "Programmable self-assembly in a thousand-robot swarm" [13].

At first, all the robots are packed together in a group without any knowledge about their location. Four of these robots are chosen to be seed robots, hence they will form the origin of the image and be the source of a gradient that will traverse the entire group. The kilobot with the highest gradient value (meaning the kilobot found furthest away from the seed) will start edge following the group. To avoid the blockage that can occur if all robots with a maximum gradient start moving at the same time some randomness is involved. The kilobots are unaware of the global positions, they can only communicate with nearby neighbours, however, they can estimate the distance from each other and can use that distance to determine their position and construct a local coordinate system with a basic implementation of trilateration. The localized robots continuously transmit a message containing their (x, y) position in the coordinate system. A new robot listens and measures distance from its localised neighbours, and when three stationaries are found, it uses this information to compute its own (x, y) position [13]. Once a robot has found its location it can determine if its inside the shape or not. If it is not, it continues to orbit around the group. Once it enters the shape, it will stop moving when it reaches the edge of the polygon or if it reaches a robot with the same gradient value. This process leads the shape to be formed successively in layers.

The steps of the algorithm are also presented in figure 3.12 and the code can be found in annex D. The seed robots are represented with green and the robots waiting in the group are colored yellow. The numbers represent their gradient values at the start of the algorithm.

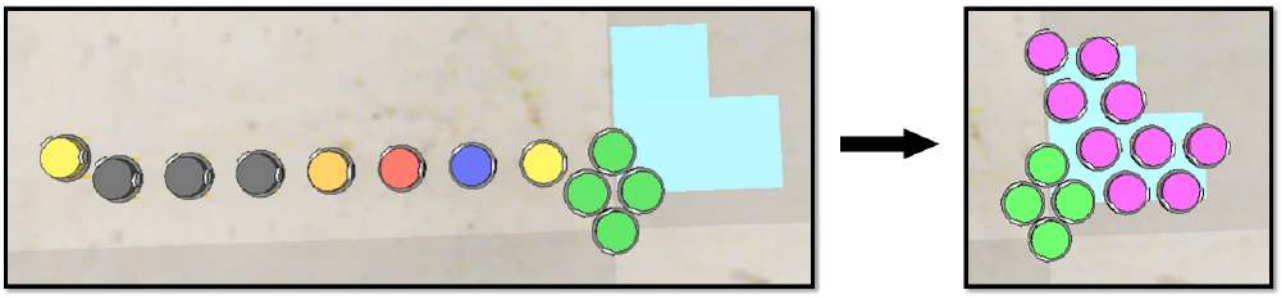


Figure 3.13: Robots demonstrating the shape formation algorithm by forming the letter L

The first to start and orbit the group will be the robots with the gradient value of 3, but the robots continuously calculate new values. Once inside the shape, the robot which changed its color to a light blue will stop at the edge of the shape. The following robot will stop when is too close to robot numbered 1. In the last image, the numbers represent the order in which the robots join the shape. An example for this algorithm in which the kilobots try to organize themselves in the shape represented in blue was simulated and is presented in figure 3.13. The robots participating in the formation are placed in a straight line to facilitate the gradient formation phase. The robots are colored in relation to their gradient, the seed robots with the gradient of 0 are colored green, yellow, blue red and purple are used to differentiate between the gradient values smaller than five. One can observe the robot from end of the queue already started the movement phase. The video for this simulation is also available as a link in the title.

For simple polygons like rectangles or squares is easy to determine if a point is inside the shape or not by comparing their coordinates to the edges of the shape. However, this algorithm can receive as input any polygon, with a non-limited number of vertices. For a fast verification of the location of the point, we used a variation of the well-known algorithm of Ray-Casting. Moreover, although the images can have an unlimited number of edges this will slow down the process, and considering the fact the robots are circles with a 33 mm diameter, the resulting shape will not have a very good resolution. Therefore, we proposed an algorithm to reduce the number of points inside a polygon but still maintaining the integrity of the shape.

3.4.1 Trilateration

Trilateration is an algorithm for finding the coordinates of a point in space. In GPS (Global Positioning System) it is used to localize the position of a Transmitter/Receiver station in a 2D plane, using the positional knowledge of three nodes. For example, it can be used to determine the position of a mobile phone located within the range of three radio transmitting towers [35]. Trilateration works with the length between points. With only one reference node we know that a point situated at a fixed range must be on the circumference of the circle with a radius equal to that range. By adding another reference node we narrow down the location of the desired point to two alternatives: the intersection of the circles. The true location is determined by using three reference nodes (figure 3.14). In our case, the reference nodes represent the seed robots, which have a predefined knowledge of their (x,y) position, and the node we want to find the location for is the orbiting robot. If we name the reference points: $\mathsf{P}_1(x_1, y_1)$, $\mathsf{P}_1(x_1, y_1)$ and $\mathsf{P}_3(x_3, y_3)$ and consider the equation of the circle we get the following system of equations:

$$\begin{cases} (x - x_1)^2 + (y - y_1)^2 = r_1^2 \\ (x - x_2)^2 + (y - y_2)^2 = r_2^2 \\ (x - x_3)^2 + (y - y_3)^2 = r_3^2 \end{cases} \tag{3.1}$$

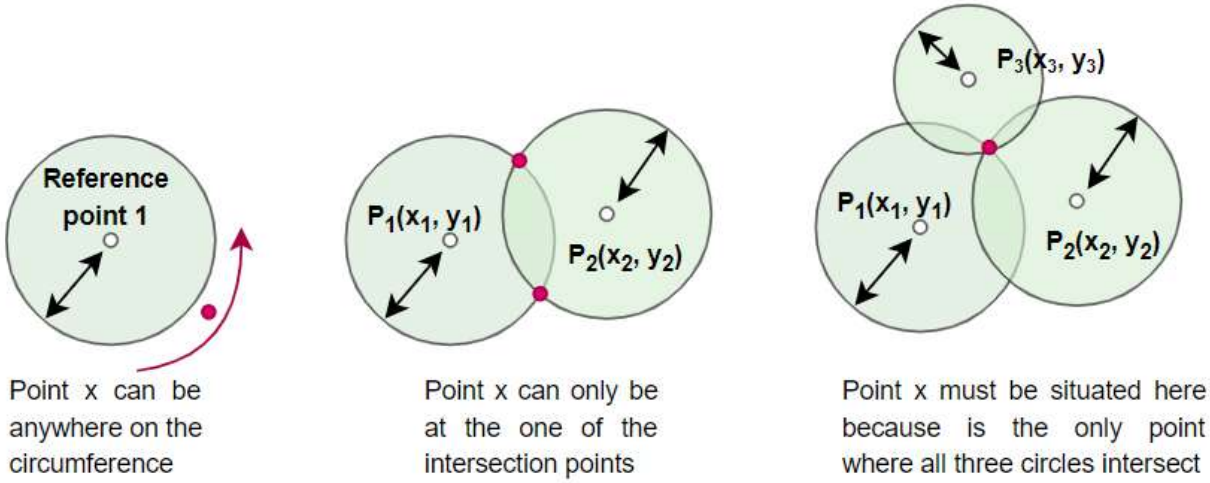


Figure 3.14: Trialateration process

We subtract the second equation from the first one and the third from the second one and we have the result:

$$\begin{aligned} x\left(-2x_1+2x_2\right)+y\left(-2y_1+2y_2\right) &= r_1^2-r_2^2-x_1^2+x_2^2-y_1^2+y_2^2 \\ x\left(-2x_2+2x_3\right)+y\left(-2y_2+2y_3\right) &= r_2^2-r_3^2-x_2^2+x_3^2-y_2^2+y_3^2 \end{aligned} \tag{3.2}$$

After we name each parenthesis with a letter we can easily find the result:

$$\begin{cases} Ax+By=C \\ Dx+Ey=F \end{cases} \Rightarrow \boldsymbol{x=\frac{CE-FB}{EA-BD}} \quad ; \quad \boldsymbol{y=\frac{CD-FA}{DB-AE}} \tag{3.3}$$

3.4.2 Ray-Casting Algorithm

In order to quickly check if a point is inside a shape, a basic implementation for the Ray-Casting algorithm was used. This is one of the basic algorithms used in computer graphics for determining what is visible inside a field of view. For each pixel, light rays are "cast" from the focal point of a virtual camera towards a 3D scene and based on what they encounter in their path the intensity of the pixel is modified. For our application we used this idea only to check the placement of points in the 2D space. The premise is simple: we have a point and we want to determine if it is inside a polygon or not, therefore we draw a line extending from it to infinity. Then we count the number of times that line intersects the polygon. If the number is even that means that the point is outside the region, if it is odd then the point is inside. As one can see in the example presented in figure 3.15, there are lines stretching across the shape from different points. The points marked with red have intersected the polygon in four or two points, meaning they are outside the area. Lines stretching from the points marked with green intersect the shape in an odd number of times, three or one, and are therefore inside the polygon.

In order to checks if this algorithm works on our system, we programmed the kilobots to spell some words, as each letter can be drawn as a polygon. We have 36 robots organized in a 6x6 grid, at a distance of 50 mm from each other and they also know from the start their position in the grid. They receive as input a string, and every six seconds they display a letter from that string. This is possible because in their memory each letter is stored as a polygon, and they continuously calculate, using ray-casting, to see if they are inside the polygon, in which case the LED turns purple, or outside the polygon, which results in the LED turning

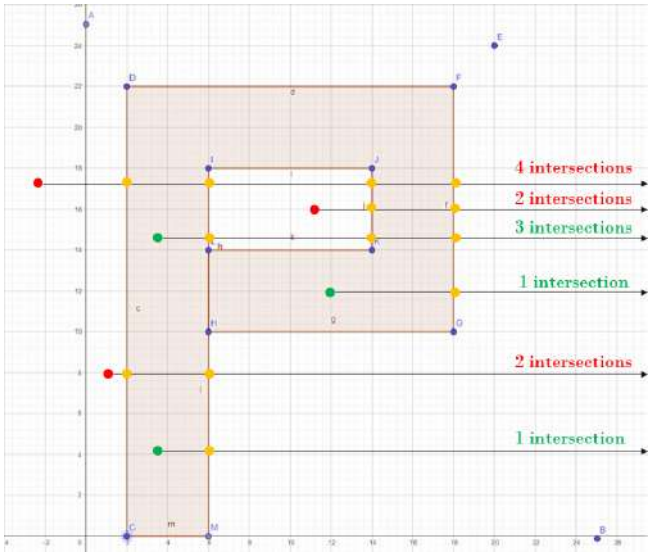


Figure 3.15: Example of ray-casting of a polygon

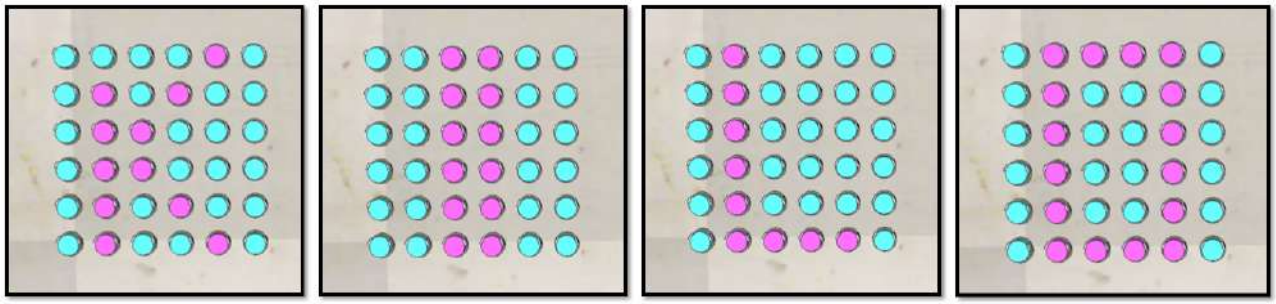
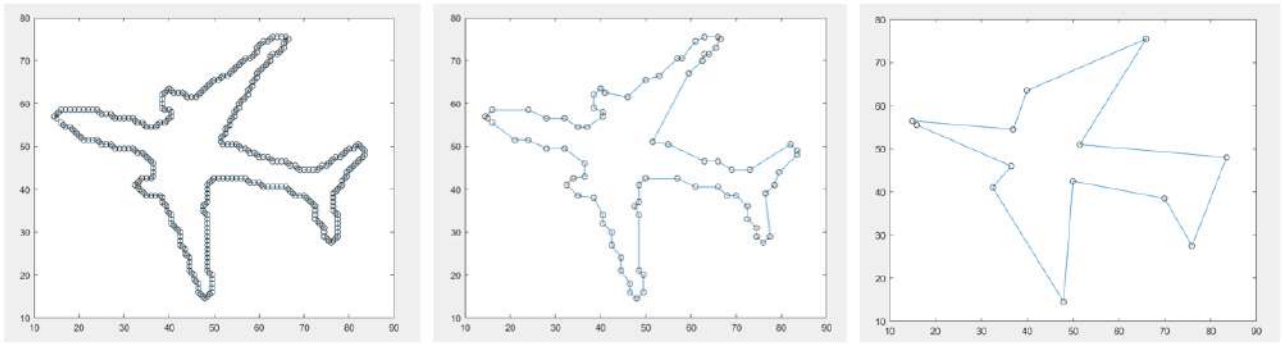


Figure 3.16: Thirty six robots forming the word "KILO"

light blue. In figure 3.16 we sent the word "**KILO**" to the robots and observed the results. The video for this experiment is also available in the title.

3.4.3 Ramer-Douglas–Peucker Algorithm

Ramer–Douglas–Peucker is a well known algorithm that is used for line simplification or to lower the resolution of a given set of points. The also called iterative end-point fit algorithm was presented by D. Douglas and T. Peucker in 1973 but it is very popular even to this day due to its speed and efficiency in areas ranging from image processing and computer graphics to cartographic generalization or even simplifying ship trajectories [36]. The algorithm implements a recursive split-and-merge strategy. The input is represented by a list of points in the 2-dimensional space, which together form a curved composed out of line segments, and a distance threshold called epsilon ε. Initially, the first and last points from the list are marked to be preserved in the final polyline. Then, using perpendicular distance, the furthest away point from the line segment stretching from the first to the last point is selected. If the distance is smaller than ε, the point can be discarded for it does not influence the final approximated curve. However, if the distance is greater than ε it means the point must be kept. Now, the algorithm will divide the point list into two halves, one from the starting point to the furthest away point and the other from that point to the last one, and repeat the process until no points remain unchecked. The threshold is the only user defined parameter and it directly influences the final result, an increase in the threshold value decreases the accuracy of the data but offers a small resulting set, at the same time, decreasing the epsilon offers a more precise



a. Setul initial b. Epislon = 0,5 c. Epislon = 5

Figure 3.17: The various results of Ramer-Douglas–Peucker algorithm

representation but a list with a greater number of elements.

For example, in figure 3.17, the starting point list had 361 elements. After applying the algorithm with a small value of 0.5 for ε the list shrinks to more than a half: 79 unique elements and the picture had practically no change. However, using a value of 5 for ε results in a rough approximation of the initial image with only 13 elements in the list. The algorithm was tested in Matlab and all the code can be found in annex F. In order to create the data set, the same binary picture as in section 3.3 was used and with the help of Matlab's pre-existing function `imcontour` we extracted the contour. Compared to other smoothing and compression algorithms, moving-average filters for example, the RDP algorithm has the advantage that the simplified shape is created from the original points, through a quite fast manner, making it a good choice to use for the kilobot application.

Chapter 4
Results and comparisons

4.1 Results using the real Kilobots

In order to test the first algorithm, it was decided to try and form a 2 x 3 rectangle. The seed robots, with the ids 1, 2 and 3, were placed on the work surface, on an auxiliary grid to facilitate the measurements. The other robots, starting with the id of -1, were placed left of the origin, in a straight line. The algorithm starts with the edge detection part. Each robot calculates its gradient and if no other robot has a greater value it will consider its position to be on the edge and start moving. While it is moving, the robot will keep constantly storing in memory the distance from its neighbours and checking if it reached a valid spot in the shape. If this happens, the robot will stop and start signaling other robots his position. This exact algorithm was tested seven times, out of which 4 times the robots successfully managed to arrange themselves in approximately 12 minutes. Twice during the failed experiments, due to errors in communication or movement, one of the robots positioned itself wrongly and deranged the whole system and a third time robot numbered two ran out of battery, thus other robots could no longer use it for coordination. The results for the successful experiments are presented in picture 4.1. For the first test we observed the robots managed to find valid spots in the rectangle, but the resulting shape was a little bended to the left, so it was decided to increase the `message_hop`, the number of states in which the robot should only listen for the neighbouring messages, from four to six. An improvement can be noticed, however the robots still consider to be stable before they reach the ideal place. Therefore, we introduced a delay between the moment the robot realises it has a valid position and the moment it stops

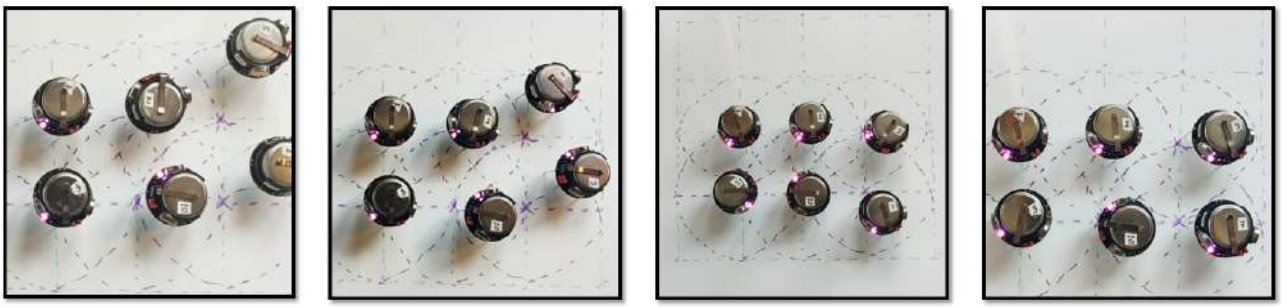


Figure 4.1: The results of four test in which the robots organize in a small rectangle using the Matrix Algorithm

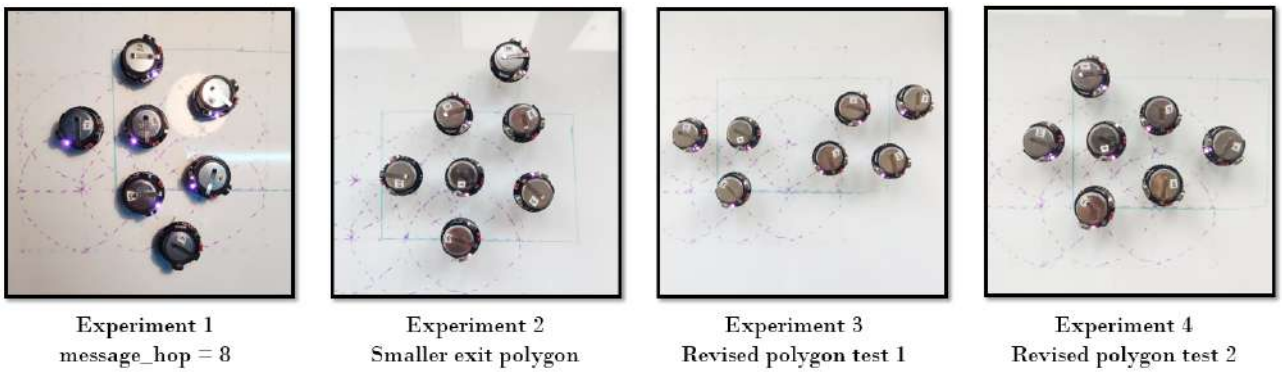


Figure 4.2: The results of four test in which the robots organize in a small rectangle using the Polygon Algorithm

the movement by making the robot orbit for two extra steps. In experiment number three the delay was to big and the robot overshoot their position but by using only one additional step the outcome was satisfactorily.

An important thing on which the proper functioning of the system depends is the calibration of robots. If they are not well calibrated they can get stuck on their way and lead to the formation of the so called "traffic jams". Also, if they have a low battery level, the motors will no longer function accordingly and can lead to the same effect. A disadvantage would be the fact that the kilobots can only arrange themselves in a grid with a even spacing of 50 millimeters. Nevertheless, this algorithm proved to be quite efficient and although with a large swarm more complex shapes can be formed, the algorithm has good results even with a smaller group.

The next step is testing shape algorithm number two, in which the kilobots calculate their position in a local coordinate system and determine if they are inside a shape or not. We decided to have a common goal as before, forming a rectangle so we can better compare the two solutions. This time we used robots labeled 10, 4 and 3 to have a predetermined knowledge of their position and placed them on top of the drawn grid, to form the origin of the system. Also, using a marker on the white board we delimited the area in which the kilobots were supposed to gather. We ran the tests several times and recorded the results. The first thing we had to modify was the `message_hop`, from 6 to 8, to allow the robots more time to listen for their neighbours and the distances from them, to minimize the errors in estimation that could interfere with the trilateration part and cause the robots to incorrectly determine their location. However, we observed that even though the robots correctly estimate the moment they enter the shape (observed by their led turning a light blue), they realize they left the shape far to late. Therefore, we introduce a new plan. The kilobots will no longer use the same polygon to check if they are leaving the shape, but will use scaled down version of it. For instance, in the first test, the polygon given to the robots was a rectangle with the length of 175 millimeters and with of 100 millimeters, and after determining valid coordinates the robot would check if it is inside the polygon or not. If inside the shape, the robot would continue to move and calculate its location and when it determines it is outside of the perimeter it will stop. The new approach considers that once inside the polygon, a scaled down version of the original polygon is used, in this example, a rectangle with the length of 142 millimeters (142 = 175 - the radius of a kilobot) and a width of 77 millimeters. It was observed that the solution had the expected outcome, however, this time the robots position themselves way before the edge of the shape and also robot seven had an approximation error and stopped outside the area.

So, we used another rectangle 160 millimeters long and 90 millimeters wide, which produced a more suitable result. In experiment three, robot labeled nine had a problem in movement and deviated from the orbiting path, but still managed to approximate its position and stabilized itself. This caused the following robots to also change their trajectories. However, all four of them stabilized themselves in the region of the rectangle. We then repeated the experiment and managed to reach a satisfying result, all robots stopped inside the shape. One can observe the formation is created gradually from the bottom to the top. This happens because, while inside the shape, one of the stopping conditions is whether a robot reaches a certain distance from another robot with the same gradient value. In our tests we used 50 millimeters as that distance. All the tests took an average 30 minutes to complete and are presented in figure 4.2.

The main disadvantage of this algorithm is the dependence on the communication range. For a robot to correctly localize itself it needs a stable connection with at least three other robots. If that connection is unstable and the distance is estimated with errors, the robot will fail to determine its position. Moreover, if even just one robot assumes a wrong position in the shape, the errors can propagate and affect other robots as well. One can be sure of the fact that if a bigger swarm is used for the experiments, the resulting shape will have a better resolution, but for smaller groups of eight to ten robots, the shapes do not have a very well defined contour.

4.2 Results using the simulation

The algorithms were tested in the simulator as well. As mentioned before, CoppeliaSim offers 4 options for its physics engine: Bullet Library, Open Dynamics Engine, Vortex Studio and Newton Dynamics. Both Bullet library and ODE are open source physics engines focused on rigid body dynamics and collision detection and are mainly used for visual effects or for video games physics. Newton Dynamics implements a deterministic solver, which is not based on classic iterative methods [37], making it better suited for real-time physics simulations according to the official CoppeliaSim website. Vortex Studio is a commercial physics engine, that works with real world parameters and it is mainly used in research projects that require a higher precision, however, Vortex Studio is not available to test for free and was not used during the test. Moreover, after numerous tests, it was found that Newton Dynamics engine does not work with the provided kilobot robot model because it causes unnatural spins or blocks in the movement of the robot. Considering this facts, the algorithms were tested using Bullet Library and Open Dynamics Engine. The simulations were performed on a machine running Windows 10 Education with 8GB Memory and 4 cores with an Intel i7-6700HQ 2.6GHz processor.

Firstly, we start with the algorithm based on the distances between neighbours and we propose the same experiment as the one conducted with the real kilobots, constructing a 2 by 3 rectangle. The test simulated with ODE took five minutes and thirty-eight seconds and resulted in all the robots correctly finding their spots. Using CoppeliaSim's features we are able to acquire their exact positions, and compare them with the ideal ones. We are then able to calculate the percentage error with the following formula:

$$\begin{aligned} \Delta\varepsilon &= |x_{\text{real}} - x_{\text{ideal}}| \\ \varepsilon_{r\%} &= \frac{\Delta\varepsilon}{x_{\text{real}}} \times 100\% \end{aligned} \tag{4.1}$$

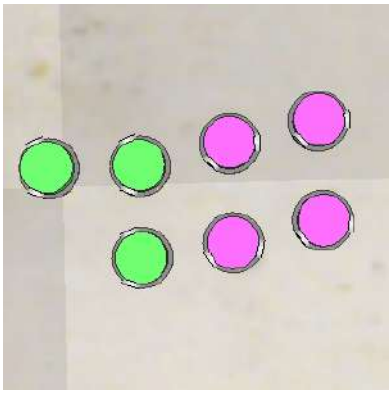


Figure 4.3: The results of the kilobots forming a rectangle in CoppeliaSim using the Matrix Algorithm

For the second engine, the simulation took nine minutes and twenty seconds, almost double the time as ODE, but the results were not so different, unperceivable to the naked eye. The calculated error was 5.66% and it was higher than the 4.23% which resulted before, but not by much. The positions and calculation were all cataloged in tables 4.1 and 4.2. One can observe from the data that the first robot managed to place itself accurately enough, however, because every robot depends on the penultimate participant who entered the shape, the following robots have an increasingly higher error in their position. The Mean Error is the average of all percentage values. On account of the result being so similar, we included only a picture of the final result, in figure 4.3.

Open Dynamics Engine						
x_real	y_real	x_ideal	y_ideal	Relative Error on Ox	Relative Error on Oy	Mean Error
94,6	-34,3	95	-40	0,42	1,2	4.23%
94,2	20	95	10	0,85	10,4	
143,3	-23,4	145	-40	1,19	17	
142,9	30,8	145	10	1,9	21,4	

Table 4.1: Results for Algorithm 1 simulation in ODE

Bullet 2.83 Engine						
x_real	y_real	x_ideal	y_ideal	Relative Error on Ox	Relative Error on Oy	Mean Error
94,8	-38,8	95	-40	0,2	1.2	5.67%
94,2	20,4	95	10	0,8	10,4	
143,81	-23	145	1,19	0,83	17	
143,1	31,4	145	10	1,9	21,4	

Table 4.2: Results for Algorithm 1 simulation in Bullet 2.83

Secondly, we tested the shape formation algorithm with the local coordinate system. The goal was similar, have four robots cover the surface of a square with the edge length of 100 mm. The ODE simulation took four minutes and five seconds and the one using Bullet 2.83 took five minutes and twenty seconds, this time, however, the results were not as similar. The final results are presented in figure 4.6. For analysing the results we also used CoppeliaSim's feature

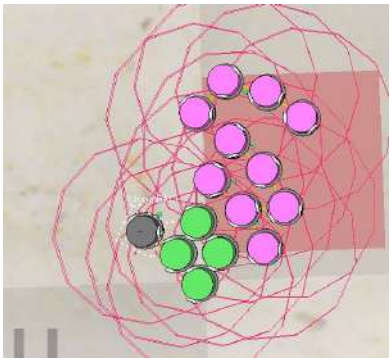
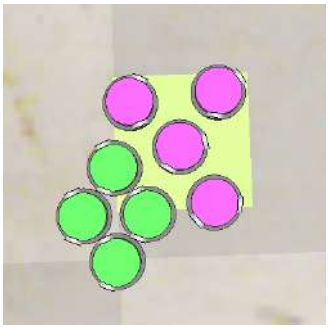
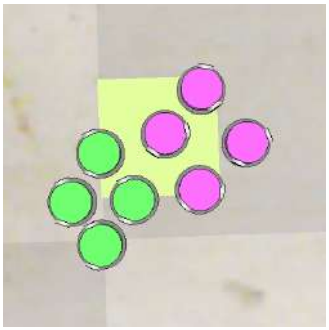


Figure 4.4: Fifteen robots in the process of forming a 150x150 rectanlge



(a) Bullet 2.83



(b) ODE

Figure 4.6: Two experiments to illustrate the differences between the physics engines

to determine the coordinates of the robots and we compared them to the coordinates computed by each robot after using the trilateration algorithm. The information was introduced in tables 4.3 and 4.4. From the data, we conclude that the robots are able to calculated their coordinates with an acceptable margin of error. As expected, the first robot to enter the shape has the most accurate knowledge of its position due to the fact that it only communicates with the seed robots, which have predetermined coordinates. Other kilobots, who follow next, take into consideration all their localized neighbours, that means any differences from reality in previous calculated points propagate throughout the system, thus explaining the fact that the last point in the tables has the biggest error.

Another experiment we tried was with a 150 x 150 square. Unfortunately, due to the limitations of the machine on which the program is working, the simulation freezes after adding fifteen robots. We can observe however the fact that the algorithm works as expected. The robots have three important states represent with the following colors: yellow for the orbiting part, a light-blue color when they think they entered the desired shape and magenta when they consider to be stable and stop the movement. It can be observed the fact that besides the first robot, others stop before reaching the edge of the square. This is because they calculate their gradient based on a fixed range of communication that is bigger than the length of two adjacent robots (60 mm), and they are conditioned to stop when they are too close to a robot with the same gradient. The video for this demonstration is available as a link in the title of picture 4.4. Also, all the videos presented in this thesis can be accessed on Google Drive[1]

[1] https://drive.google.com/drive/folders/1weGDtCmvJIWS3-TwIypSJ4E6DED8Scq_?usp=sharing

Bullet 2.83 Engine						
x_real	y_real	x_sim	y_sim	Relative Error on Ox	Relative Error on Oy	Mean Error
58,6	32,4	58,58	32,7	0,02	0,3	5.06%
40,2	66,79	39,38	66,55	0,82	0,24	
10,6	93,68	10,603	94,026	0,003	0,346	
48,69	105,4	63,39	98,5	14,7	6,9	

Table 4.3: Results for Algorithm 2 simulation in Bullet 2.83

ODE Engine						
x_real	y_real	x_sim	y_sim	Relative Error on Ox	Relative Error on Oy	Mean Error
66,82	28,35	67,2	32,66	0,38	4,31	8.12%
44,25	70,1	45,07	71	0,82	0,9	
101,12	62,42	100,01	63,18	1,11	0,76	
50,9	100,4	70,8	95,7	19,9	4,7	

Table 4.4: Results for Algorithm 2 simulation in ODE

Chapter 5
Conclusion

5.1 General Conclusions

Shape formation is one of the essential problems swarm robotics is keen on solving. The task is usually framed as such: starting from a group of robots with random locations and no knowledge of their environment, have the group use local interactions and arrange themselves in any given form without any collisions. This thesis started from this idea and presented a similar process in which a group of robots communicates with one another and exchange information, all for the common goal to create different shapes.

The Kilobots represent a good system for testing these sort of algorithms. They have a simple moving principle based on vibration and they use infrared light for sending and receiving information, and because of that they come at a relatively affordable price. Moreover, within this project a virtual simulation was also used in order to test the algorithms before implementing them. The algorithms proved to be successful and the results of the observations were also documented in this paper. The first completed algorithm was based on the distances between each neighbouring robots. Each robot had a matrix in which all the valid points inside a shape were specified. The goal of the robot was to identify such a point and occupy that position. The shape created gradually as more robots found empty positions. Another algorithm created a local coordinate system by placing three seed robots in the origin of the shape. Other robots use the seed to orientate in the coordinate system and determine if they are inside a polygon or not. Out of the two algorithms the first one proved to have better results, but with a greater swarm the results might differ.

5.2 Personal Contributions

Within this project, several proprieties of swarm robotics were discussed and analyzed. That being said, my personal contributions were:

- **review of the current state-of-the-art in swarm robotics studies** : Taking into consideration the fast-changing nature of this domain, we presented some of the most recent research papers available, the most recent being published in 2020, and we investigated some of the most interesting aspects of their results.
- **workspace setup organization** : Considering the kilobot's proprieties, we proposed a reflective and smooth surface to ensure correct movement, proper lighting to maximize

the communication range, and placed the controller at the appropriate height to assure all the information can be received by the entire group of robots at once.

- **suggested a simulation program for working with kilbots** : In order to test the algorithms on a greater number of units, and also to check if everything is working accordingly before uploading the instructions to the robots we present the use of CoppeliaSim.
- **implemented and tested shape formation algorithms** After implementing some basic applications such as edge following, gradient formation, and edge detection, we used those principles and combined them into more complex algorithms in which the group of robots self organizes into any given shape.
- **compared the algorithms both in real time and in the simulator** : After having an adequate implementation for the algorithm, we ran experiments for multiple times and discussed the different situations which may occur. Also, we presented a comparison between CoppeliaSim's physics engines

5.3 Further developments

The project described in this paper is not an encapsulated one, it can be scaled and combined with other concepts to create something even more complex. Some of the improvements that can be added to this system include:

- using **augmented reality** to better record the movements of the kilobots, and also, use it to develop systems that require a feedback loop from the kilobots to the programming center
- **acquiring a greater number of kilobots** to test the algorithms in bigger scenarios
- **testing or even implementing other simulations** to find the best solution for simulating the robots

An interesting project that can be implemented in the future is having the robots search for targets in a virtual environment. Using cameras mounted above the workspace and system similar to ARK [8], one can combine reality with simulated objects, for example, create the desired targets, a nest for the robots to return to when the target is found and even walls or obstacles to observe the way the swarm will search for the most efficient solution. Also, one can have the swarm traverse a labyrinth and search for the exit. Another project that can be implemented in the future can use the kilobots to mimic real time drawing. The robots arranged in a grid, will be associated with the pixels of a canvas on which a user can create pictures with every color and shape. Every few seconds the robots will be updated with the new picture, creating the impression of real time drawing. Furthermore, with a greater number of robots, an entire ecosystem could be simulated. Some kilobots could be delegated to be gatherers of food, others explorers, and some even predators. This experiment can also be scaled up by having more than one swarm in the system and compel them to compete for resources.

In conclusion, the kilobots can be used in a variety of projects in all sorts of domains. It is my presumption that as the technology evolves, microrobots will become not only "smarter" with the addition of more sensors and computational power but even smaller in size, even at molecular levels, so that they could have a real impact in the world of medicine and non-invasive treatments and investigations.

References

[1] Edmund R Hunt, Simon Jones, and Sabine Hauert. Testing the limits of pheromone stigmergy in high-density robot swarms. *Royal Society open science*, 6(11):190225, 2019.

[2] Calum Imrie and J Michael Herrmann. Self-organised transitions in swarms with turing patterns.

[3] Federico Pratissoli, Andreagiovanni Reina, Yuri Kaszubowski Lopes, Lorenzo Sabattini, and Roderich Groβ. A soft-bodied modular reconfigurable robotic system composed of interconnected kilobots. In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 50–52. IEEE, 2019.

[4] Mayank Agrawal and Sharon C Glotzer. Scale-free, programmable design of morphable chain loops of kilobots and colloidal motors. *Proceedings of the National Academy of Sciences*, 117(16):8700–8710, 2020.

[5] Kilobotics Website. https://kilobotics.com/documentation.

[6] Andrei-Daniel Dedu. Sistem inteligent bazat pe robot, i colaborativi. In *Diploma thesis*, pages –. UPB, 2020.

[7] Duncan E Jackson and Francis LW Ratnieks. Communication in ants. *Current biology*, 16(15):R570–R574, 2006.

[8] Andreagiovanni Reina, Alex J Cope, Eleftherios Nikolaidis, James AR Marshall, and Chelsea Sabo. Ark: Augmented reality for kilobots. *IEEE Robotics and Automation letters*, 2(3):1755–1761, 2017.

[9] Simon Garnier, Faben Tache, Maud Combe, Anne Grimal, and Guy Theraulaz. Alice in pheromone land: An experimental setup for the study of ant-like robots. In *2007 IEEE swarm intelligence symposium*, pages 37–44. IEEE, 2007.

[10] Gabriele Valentini, Anthony Antoun, Marco Trabattoni, Bernát Wiandt, Yasumasa Tamura, Etienne Hocquard, Vito Trianni, and Marco Dorigo. Kilogrid: a novel experimental environment for the kilobot robot. *Swarm Intelligence*, 12(3):245–266, 2018.

[11] Alan Turing. The chemical basis of morphogenesis (1952). *B. Jack Copeland*, page 519.

[12] Yusuke Ikemoto, Yasuhisa Hasegawa, Toshio Fukuda, and Kazuhiko Matsuda. Gradual spatial pattern formation of homogeneous robot group. *Information Sciences*, 171(4):431–445, 2005.

[13] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[14] J. Pugh and A. Martinoli. Inspiring and modeling multi-robot search with particle swarm optimization. In *Proceedings of the 2007 IEEE Swarm Intelligence Symposium*, page 332–339, USA, 2007. IEEE Computer Society.

[15] Michael Rubenstein, Adrian Cabrera, Justin Werfel, Golnaz Habibi, James McLurkin, and Radhika Nagpal. Collective transport of complex objects by simple robots: Theory and experiments. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS '13, page 47–54, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[16] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. A real-time algorithm for mobile robot mapping with applications to multi-robot and 3d mapping. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 1, pages 321–328. IEEE, 2000.

[17] Huaxing Xu, Haibing Guan, Alei Liang, and Xinan Yan. A multi-robot pattern formation algorithm based on distributed swarm intelligence. In *2010 Second International Conference on Computer Engineering and Applications*, volume 1, pages 71–75. IEEE, 2010.

[18] Mario Coppola, Jian Guo, Eberhard Gill, and Guido CHE de Croon. Provable self-organizing pattern formation by a swarm of robots with limited knowledge. *Swarm Intelligence*, 13(1):59–94, 2019.

[19] Simon Garnier, Jacques Gautrais, and Guy Theraulaz. The biological principles of swarm intelligence. *Swarm intelligence*, 1(1):3–31, 2007.

[20] Eshel Ben-Jacob, Ofer Schochet, Adam Tenenbaum, Inon Cohen, Andras Czirok, and Tamas Vicsek. Generic modelling of cooperative growth patterns in bacterial colonies. *Nature*, 368(6466):46–49, 1994.

[21] Odile Petit, Jacques Gautrais, J-B Leca, Guy Theraulaz, and J-L Deneubourg. Collective decision-making in white-faced capuchin monkeys. *Proceedings of the Royal Society B: Biological Sciences*, 276(1672):3495–3503, 2009.

[22] Steven V Viscido, Julia K Parrish, and Daniel Grünbaum. The effect of population size and number of influential neighbors on the emergent properties of fish schools. *Ecological modelling*, 183(2-3):347–363, 2005.

[23] Dirk Helbing, Péter Molnár, Illés J Farkas, and Kai Bolay. Self-organizing pedestrian movement. *Environment and planning B: planning and design*, 28(3):361–383, 2001.

[24] Marco Dorigo, Dario Floreano, Luca Maria Gambardella, Francesco Mondada, Stefano Nolfi, Tarek Baaboura, Mauro Birattari, Michael Bonani, Manuele Brambilla, Arne Brutschy, et al. Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine*, 20(4):60–71, 2013.

[25] Vito Trianni, Joris IJsselmuiden, and Ramon Haken. The saga concept: Swarm robotics for agricultural applications. Technical report, Technical Report. 2016. Available online: http://laral. istc. cnr. it/saga . . . , 2016.

[26] Farshad Arvin, John Murray, Chun Zhang, and Shigang Yue. Colias: An autonomous micro robot for swarm robotic applications. *International Journal of Advanced Robotic Systems*, 11(7):113, 2014.

[27] Francesco Mondada, Edoardo Franzi, and Andre Guignard. The development of khepera. In *Experiments with the Mini-Robot Khepera, Proceedings of the First International Khepera Workshop*, number CONF, pages 7–14, 1999.

[28] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *2012 IEEE International Conference on Robotics and Automation*, pages 3293–3298, 2012.

[29] Microchip Technology. *megaAVR Data Sheet*, 5 2019.

[30] CoppeliaSim customers. *https://www.coppeliarobotics.com/customers*.

[31] Lua Official Page. *http://www.lua.org/about.html*.

[32] Jan L Christian. Morphogen gradients in development: from form to function. *Wiley Interdisciplinary Reviews: Developmental Biology*, 1(1):3–15, 2012.

[33] Scott F Gilbert. Axis formation in amphibians: the phenomenon of the organizer. In *Developmental Biology. 6th edition*. Sinauer Associates, 2000.

[34] Sudhakar Kumar Anurag, Ponala Venkata Eswara Srisai and Kishan Chouhan. Systems and control engineering laboratory (sc 626) kilobotics. page 44, 2019.

[35] Joseph Thomas. A novel trilateration algorithm for localization of a transmitter/receiver station in a 2d plane using analytical geometry. 09 2014.

[36] Liangbin Zhao and Guoyou Shi. A method for simplifying ship trajectory based on improved douglas–peucker algorithm. *Ocean Engineering*, 166:37–46, 2018.

[37] CoppeliaSim's dynamics modules. *coppeliarobotics.com/helpFiles/en/dynamicsModule.htm*.

Anexa A
Code for Edge Following

```
#include "kilolib.h"

#define STOP 0
#define FORWARD 1
#define LEFT 2
#define RIGHT 3

#define TOO_CLOSE 30
#define DESIRED_DISTANCE 50

message_t message;
int message_hop = 0;
int total_msg_hop = 6;

int new_message;
int new_dist;
int dist_min;

// Function to handle motion.
void set_motion(int new_motion)
{
    if (new_motion == STOP)
    {
        set_motors(0, 0);
    }
    else if (new_motion == FORWARD)
    {
        spinup_motors();
        set_motors(kilo_straight_left, kilo_straight_right);
    }
    else if (new_motion == LEFT)
    {
        set_motors(255,0);
        delay(15);
        //spinup_motors(); //this function starts the both motors for 15ms
        set_motors(kilo_turn_left, 0);
        delay(500);
        set_motors(0, 0);
    }
    else if (new_motion == RIGHT)
    {
        set_motors(0,255);
        delay(15);
        //spinup_motors();
        set_motors(0, kilo_turn_right);
        delay(500);
        set_motors(0, 0);
    }
}

//function for deciding the direction of orbiting
void orbit(const int d)
{
```

```
54      if(d < TOO_CLOSE)
55      {
56          set_color(RGB(1,0,0));
57          set_motion(STOP);
58      }
59
60      else if( d > DESIRED_DISTANCE )
61      {
62          set_color(RGB(1,1,0));
63          set_motion(RIGHT);
64      }
65      else if ( d < DESIRED_DISTANCE )
66      {
67          set_color(RGB(0,0,1));
68          set_motion(LEFT);
69      }
70      else
71      {
72          set_color(RGB(1,0,0));
73          set_motion(FORWARD);
74      }
75  }
76
77  //setup function that runs only once
78  //used for inisialisation
79  void setup()
80  {
81    dist_min = 200;
82    new_dist = 0;
83    new_message = 0;
84    message_hop = 0;
85  }
86
87  //the main code - will run repeatedly
88  void loop()
89  {
90    //check if a new message arrived
91      if (new_message == 1)
92      {
93          new_message = 0;
94          message_hop++;
95
96      //remember the minimum distance
97          if (new_dist < dist_min)
98              dist_min = new_dist;
99
100     //when the total number of messages has been reached
101     //begin the orbiting process
102         if(message_hop == total_msg_hop)
103         {
104             message_hop = 0;
105             orbit(dist_min);
106             dist_min = 200;
107         }
108     }
109 }
110
111 //function for receiveing messages
112 void message_rx(message_t *m, distance_measurement_t *d)
113 {
114     new_message = 1;
115     new_dist = estimate_distance(d);
116 }
117
118 int main() {
119     kilo_init();
120     kilo_message_rx = message_rx;
121     kilo_start(setup, loop);
122     return 0;
123 }
```

Anexa B
Code for Edge Detection using gradient

```
#include "kilolib.h"

#define STOP 0
#define FORWARD 1
#define LEFT 2
#define RIGHT 3

#define TOO_CLOSE 30
#define DESIRED_DISTANCE 50
#define NEIGHBOUR_DIST 60

#define seedID 0

#define WAIT 0
#define MOVING 1

int current_motion = STOP;

int message_hop = 0;
int total_message_hop = 8;

int new_message = 0;
float new_dist = 0;
int dist_min = 200;

message_t message;
uint32_t last_update = 0;

int my_id = 7;

int myGradient = 255;
int received_gradient = 0;
int minGradient = 255;
int maxGradient = 0;

float dist = 9000;

int status = WAIT;

int edge_wait = 0;

// Function to handle motion.
void set_motion(int new_motion)
{

  current_motion = new_motion;

  if (current_motion == STOP)
  {
    set_motors(0, 0);
  }
  else if (current_motion == FORWARD)
  {
```

```
54      spinup_motors();
55      set_motors(kilo_straight_left, kilo_straight_right);
56    }
57    else if (current_motion == LEFT)
58    {
59      set_motors(255,0);
60      delay(15);
61      //spinup_motors();
62      set_motors(kilo_turn_left, 0);
63      delay(500);
64      set_motors(0, 0);
65    }
66    else if (current_motion == RIGHT)
67    {
68      set_motors(0,255);
69      delay(15);
70      //spinup_motors();
71      set_motors(0, kilo_turn_right);
72      delay(500);
73      set_motors(0, 0);
74    }
75
76  }
77
78  //function for deciding the direction of orbiting
79  void orbit(const int d)
80  {
81    if(d < TOO_CLOSE)
82    {
83      set_color(RGB(1,0,0));
84      set_motion(FORWARD);
85    }
86
87    else if( d > DESIRED_DISTANCE )
88    {
89      set_color(RGB(1,1,0));
90      set_motion(RIGHT);
91    }
92    else if ( d < DESIRED_DISTANCE )
93    {
94      set_color(RGB(0,0,1));
95      set_motion(LEFT);
96    }
97    else
98    {
99      set_color(RGB(1,0,0));
100     set_motion(FORWARD);
101   }
102 }
103
104 //setup function that runs only once
105 //used for inisialisation
106 void setup() {
107   message.type = NORMAL;
108   message.data[1] = 0;
109   message.crc = message_crc(&message);
110
111   if( my_id == seedID )
112   {
113     myGradient = 0;
114   }
115 }
116
117 //the main code - will run repeatedly
118 void loop()
119 {
120   //check for incoming messages
121     if (new_message == 1 && my_id != seedID)
122     {
123     new_message = 0;
124     message_hop++;
125     last_update = kilo_ticks;
126
127     //use the received information to determine the minimum and maximum gradient value
128     //and also the minimum distance
129     if(received_gradient < minGradient && new_dist <= NEIGHBOUR_DIST)
```

```
130     {
131       minGradient = received_gradient;
132     }
133
134     if(received_gradient > maxGradient && new_dist <= NEIGHBOUR_DIST)
135     {
136       maxGradient = received_gradient;
137     }
138
139     if (new_dist < dist_min)
140             dist_min = new_dist;
141
142     //check if the number of total messages has been reahced
143     if(message_hop == total_message_hop)
144     {
145       message_hop = 0;
146
147       myGradient = minGradient + 1;
148       message.type = NORMAL;
149       message.data[1] = myGradient;
150       message.crc = message_crc(&message);
151
152       if( myGradient > maxGradient)
153       {
154         edge_wait++;
155       }
156
157       minGradient = 255;
158
159       if(status == MOVING)
160         orbit(dist_min);
161             dist_min = 200;
162     }
163   }
164
165   //if the robot had the maximum gradient in its region for one second it will enter the
        MOVING state
166   if( edge_wait > 32 && status == WAIT )
167     status = MOVING;
168
169   //color coding depending on the gradient
170   if( myGradient == 0 )
171   {
172     set_color(RGB(0,1,0));
173   }
174   else if (myGradient == 1)
175   {
176     set_color(RGB(1,1,0));
177   }
178   else if (myGradient == 2)
179   {
180     set_color(RGB(0,0,1));
181   }
182   else if (myGradient == 3)
183   {
184     set_color(RGB(1,0,0));
185   }
186   else
187   {
188     set_color(RGB(0,0,0));
189   }
190
191   //if no more messages were received in the last two seconds the robot will stop any movement
        and increase its gradient
192   if( kilo_ticks > (last_update + 64) && myGradient < 255 && my_id != seedID)
193   {
194     set_motion(STOP);
195     set_color(RGB(0,0,0));
196     myGradient = myGradient + 1;
197   }
198
199 }
200
201 //function for transmiting a message
202 message_t* message_tx()
203 {
```

```
204   return &message;
205 }
206
207 //function for receiving messages
208 void message_rx(message_t *m, distance_measurement_t *d)
209 {
210     new_message = 1;
211     new_dist = estimate_distance(d);
212     received_gradient = m->data[1];
213
214 }
215
216
217 int main() {
218     kilo_init();
219     kilo_message_rx = message_rx;
220     kilo_message_tx = message_tx;
221
222     kilo_start(setup, loop);
223
224     return 0;
225 }
```

Anexa C
Code for shape formation algorithm 1

```
#include "kilolib.h"
#include<math.h>
#include<stdbool.h>

#define MAXIM 11
#define INF 10000

#define STOP 0
#define FORWARD 1
#define LEFT 2
#define RIGHT 3

#define TOO_CLOSE 40
#define DESIRED_DISTANCE 50
#define NEIGHBOUR_DIST 50

#define seedID 0

#define WAIT 0
#define MOVING 1
#define INSIDE_SHAPE 2
#define LOCAL 3

//declaring the variables used

message_t message;
uint32_t last_update = 0;
uint32_t new_message;
uint16_t message_hop = 0;
uint16_t total_message_hop = 6;

int status;
int current_motion = STOP;
int edge_wait = 0;

int dist;
int gradient;
int id;
int localised_flag;
int shape_index;

int received_dist;
int received_id;
int received_timestamp;
int received_gradient;
int received_shape_index;
int received_local_flag;

//data structure to encapsulate all the information from the neighbouring robots
struct Neighbour
{
  int dist;
  int gradient;
```

```
54    int id;
55    int timestamp;
56    int localised_flag;
57    uint8_t flag;
58    int shape_index;
59  };
60
61  struct Neighbour neighbours[MAXIM];
62
63  //the matrix used for creating a house shape
64  int rowsShape = 5;
65  int colsShape = 4;
66  float shape_array[5][4] = { {3,1,2,1}, {4,1,2,1.4142},
67                {2,1,5,1}, {6,1.4142,4,1}, {2, 1, 6, 1.41}
68               };
69
70
71  float distance_array[MAXIM] = {-1, -1, -1, -1, -1, -1, -1, -1,-1,-1,-1};
72
73  //function to handle the motion
74  void set_motion(int new_motion)
75  {
76    if (new_motion == STOP)
77    {
78      set_motors(0, 0);
79    }
80    else if (new_motion == FORWARD)
81    {
82      spinup_motors();
83      set_motors(kilo_straight_left, kilo_straight_right);
84    }
85    else if (new_motion == LEFT)
86    {
87      set_motors(255,0);
88      delay(15);
89      //spinup_motors();
90      set_motors(kilo_turn_left, 0);
91    }
92    else if (new_motion == RIGHT)
93    {
94      set_motors(0,255);
95      delay(15);
96      //spinup_motors();
97      set_motors(0, kilo_turn_right);
98    }
99
100   delay(500);
101   set_motors(0, 0);
102 }
103
104 //function for deciding the direction of orbiting
105 void orbit(const int d)
106 {
107   if(d < 30)
108   {
109     set_color(RGB(1,0,0));
110     set_motion(FORWARD);
111   }
112
113   else if( d > ( DESIRED_DISTANCE ) )
114   {
115     set_color(RGB(1,1,0));
116     set_motion(RIGHT);
117   }
118   else if ( d < (DESIRED_DISTANCE))
119   {
120     set_color(RGB(0,0,1));
121     set_motion(LEFT);
122   }
123   else
124   {
125     set_color(RGB(1,0,0));
126     set_motion(FORWARD);
127   }
128 }
129
```

```
130 //helper function to set the message before transmitting
131 void set_message()
132 {
133   if(status == LOCAL)
134     localised_flag = 1;
135   else
136     localised_flag = -1;
137   message.type = NORMAL;
138   message.data[0] = kilo_uid;  //id
139   //message.data[1] = kilo_ticks;   //timestamp
140   message.data[2] = gradient;
141   message.data[3] = shape_index;
142   message.data[5] = localised_flag;
143 
144   message.crc = message_crc(&message);
145 }
146 
147 //helper function to set the elements in the neighbours array
148 void set_neighbour()
149 {
150   neighbours[received_id].dist = received_dist;
151   neighbours[received_id].flag = 1;
152   neighbours[received_id].gradient = received_gradient;
153   neighbours[received_id].localised_flag = received_local_flag;
154   neighbours[received_id].timestamp = kilo_ticks;
155   neighbours[received_id].shape_index = received_shape_index;
156 }
157 
158 //helper function to discard any robot with whom he could not communicate in the last 3
      seconds
159 void eliminateNeighbour()
160 {
161   for(uint8_t i = 0; i < MAXIM; i++)
162   {
163     if(kilo_ticks > (neighbours[i].timestamp + 96) )
164     {
165       neighbours[i].dist = -1;
166       neighbours[i].flag = -1;
167       neighbours[i].gradient = -1;
168       neighbours[i].localised_flag = -1;
169       neighbours[i].timestamp = -1;
170       neighbours[i].shape_index = 1000;
171     }
172   }
173 }
174 
175 //data structure to encapsulate the concept of a point in space
176 struct point
177 {
178   float x;
179   float y;
180   int dist;
181 };
182 
183 void setup()
184 {
185   for(uint8_t i = 0; i < MAXIM; i++)
186   {
187     neighbours[i].flag = -1;
188     neighbours[i].dist = -1;
189     neighbours[i].timestamp = -1;
190     neighbours[i].gradient = -1;
191     neighbours[i].shape_index = 1000;
192   }
193 
194   status = WAIT;
195   if(kilo_uid <= 3)
196   {
197     gradient = 0;
198     localised_flag = 1;
199     status = LOCAL;
200   }
201   else
202   {
203     gradient = 255;
204     localised_flag = 0;
```

```
205   }
206
207   switch (kilo_uid)
208   {
209   case 0:
210     shape_index = 0;
211     break;
212   case 1:
213     shape_index = 1;
214     break;
215   case 2:
216     shape_index = 2;
217     break;
218   case 3:
219     shape_index = 3;
220     break;
221
222   default:
223     shape_index = 1000;
224   }
225
226   set_message();
227 }
228
229 void loop()
230 {
231   //check for incoming messages and update the neighbours array
232   if(new_message == 1)
233   {
234     new_message = 0;
235     last_update = kilo_ticks;
236     set_neighbour();
237     if(received_local_flag == 1)
238     {
239       distance_array[received_shape_index] = received_dist;
240     }
241     message_hop++;
242   }
243
244   //check if the number of total messages has been reahced
245   if(message_hop == total_message_hop)
246   {
247     message_hop = 0;
248     int dist_min = 200;
249     int gradient_min = 254;
250     int gradient_max = 0;
251
252     int valid = 0;
253     struct point aux[4];
254     int index = 0;
255     for(uint8_t i = 0; i < MAXIM; i++)
256     {
257       if(neighbours[i].dist != -1 && neighbours[i].dist < dist_min)
258         dist_min = neighbours[i].dist;
259       if(neighbours[i].dist < 55 && neighbours[i].gradient != -1 && neighbours[i].gradient <
      gradient_min)
260           gradient_min = neighbours[i].gradient;
261       if(neighbours[i].dist < 55 && neighbours[i].gradient != -1 && neighbours[i].gradient >
      gradient_max)
262           gradient_max = neighbours[i].gradient;
263     }
264
265     //update the gradient
266     if(kilo_uid > 3)
267       gradient = gradient_min + 1;
268
269     //edge detection part
270     if(kilo_uid > 3 && gradient > gradient_max)
271       edge_wait++;
272     else if(kilo_uid > 3)
273       edge_wait = 0;
274
275     if(edge_wait > 32 && status == WAIT)
276       status = MOVING;
277
278     //update the message that will be sent to other robots
```

```
279     set_message();
280
281     // switch (gradient)
282     // {
283     // case 0:
284     //  set_color(RGB(0,1,1));
285     //  break;
286     // case 1:
287     //  set_color(RGB(1,1,0));
288     //  break;
289     // case 2:
290     //  set_color(RGB(0,0,1));
291     //  break;
292     // case 3:
293     //  set_color(RGB(1,0,0));
294     //  break;
295     // case 4:
296     //  set_color(RGB(1,0,1));
297     //  break;
298     // case 5:
299     //  set_color(RGB(0,1,0));
300     //  break;
301
302     // default:
303     //  set_color(RGB(0,0,0));
304     //  break;
305     // }
306
307     //depending on the status of the robot either orbit around the group or chenge the color
      of the led
308     if(kilo_uid >= 3 && dist_min < 200 && (status == MOVING || status == INSIDE_SHAPE))
309       orbit(dist_min);
310     else
311       set_motion(STOP);
312
313     if(status == LOCAL)
314       set_color(RGB(3,0,3));
315
316     if(status == INSIDE_SHAPE)
317       set_color(RGB(0,3,3));
318
319     //check for a valid position in the shape
320     if (kilo_uid > 3 && status != LOCAL)
321     {
322       float aux_dist;
323       int index;
324       for(int i = 0; i < rowsShape; i++)
325       {
326         char ok = 1;
327         for(int j = 0; j < colsShape; j = j + 2)
328         {
329           aux_dist = shape_array[i][j + 1] * 50;
330           index = shape_array[i][j];
331           if( (distance_array[index] < ( aux_dist - 10 )) || (distance_array[index] > (
      aux_dist + 10 )))
332             ok = 0;
333         }
334
335         if(ok == 1)
336         {
337           shape_index = i + 4;
338           status = LOCAL;
339           localised_flag = 1;
340           orbit(dist_min);
341           set_message();
342
343           break;
344         }
345       }
346     }
347   }
348   eliminateNeighbour();
349 }
350
351 //function for sending messages
352 message_t* message_tx()
```

```
353 {
354   return &message;
355 }
356
357 //function for receiving messages
358 void message_rx(message_t *m, distance_measurement_t *d)
359 {
360     new_message = 1;
361     received_dist = estimate_distance(d);
362     received_id = m->data[0];
363   //received_timestamp = m->data[1];
364   received_gradient = m->data[2];
365   received_shape_index = m->data[3];
366   received_local_flag = m->data[5];
367 }
368
369 int main()
370 {
371     kilo_init();
372     kilo_message_rx = message_rx;
373     kilo_message_tx = message_tx;
374
375     kilo_start(setup, loop);
376
377     return 0;
378 }
```

Anexa D
Code for shape formation algorithm 2

```
#include "kilolib.h"
#include<math.h>
#include<stdbool.h>

#define MAXIM 11
#define INF 10000

#define STOP 0
#define FORWARD 1
#define LEFT 2
#define RIGHT 3

#define TOO_CLOSE 40
#define DESIRED_DISTANCE 50
#define NEIGHBOUR_DIST 50

#define seedID 0

#define WAIT 0
#define MOVING 1
#define INSIDE_SHAPE 2
#define LOCAL 3

//declaring the variables used

message_t message;
uint32_t last_update = 0;
uint32_t new_message;
uint16_t message_hop = 0;
uint16_t total_message_hop = 6;

int status;
int current_motion = STOP;
int edge_wait = 0;

int dist;
int gradient;
int id;
int localised_flag;
int shape_index;

int received_dist;
int received_id;
int received_timestamp;
int received_gradient;
int received_shape_index;
int received_local_flag;

//data structure to encapsulate all the information from the neighbouring robots
struct Neighbour
{
  int dist;
  int gradient;
```

```
54    int id;
55    int timestamp;
56    int localised_flag;
57    uint8_t flag;
58    int shape_index;
59  };
60
61  struct Neighbour neighbours[MAXIM];
62
63  //the matrix used for creating a house shape
64  int rowsShape = 5;
65  int colsShape = 4;
66  float shape_array[5][4] = { {3,1,2,1}, {4,1,2,1.4142},
67                {2,1,5,1}, {6,1.4142,4,1}, {2, 1, 6, 1.41}
68               };
69
70
71  float distance_array[MAXIM] = {-1, -1, -1, -1, -1, -1, -1, -1,-1,-1,-1};
72
73  //function to handle the motion
74  void set_motion(int new_motion)
75  {
76    if (new_motion == STOP)
77    {
78      set_motors(0, 0);
79    }
80    else if (new_motion == FORWARD)
81    {
82      spinup_motors();
83      set_motors(kilo_straight_left, kilo_straight_right);
84    }
85    else if (new_motion == LEFT)
86    {
87      set_motors(255,0);
88      delay(15);
89      //spinup_motors();
90      set_motors(kilo_turn_left, 0);
91    }
92    else if (new_motion == RIGHT)
93    {
94      set_motors(0,255);
95      delay(15);
96      //spinup_motors();
97      set_motors(0, kilo_turn_right);
98    }
99
100   delay(500);
101   set_motors(0, 0);
102 }
103
104 //function for deciding the direction of orbiting
105 void orbit(const int d)
106 {
107   if(d < 30)
108   {
109     set_color(RGB(1,0,0));
110     set_motion(FORWARD);
111   }
112
113   else if( d > ( DESIRED_DISTANCE ) )
114   {
115     set_color(RGB(1,1,0));
116     set_motion(RIGHT);
117   }
118   else if ( d < (DESIRED_DISTANCE))
119   {
120     set_color(RGB(0,0,1));
121     set_motion(LEFT);
122   }
123   else
124   {
125     set_color(RGB(1,0,0));
126     set_motion(FORWARD);
127   }
128 }
129
```

```
130 //helper function to set the message before transmitting
131 void set_message()
132 {
133   if(status == LOCAL)
134     localised_flag = 1;
135   else
136     localised_flag = -1;
137   message.type = NORMAL;
138   message.data[0] = kilo_uid;  //id
139   //message.data[1] = kilo_ticks;   //timestamp
140   message.data[2] = gradient;
141   message.data[3] = shape_index;
142   message.data[5] = localised_flag;
143 
144   message.crc = message_crc(&message);
145 }
146 
147 //helper function to set the elements in the neighbours array
148 void set_neighbour()
149 {
150   neighbours[received_id].dist = received_dist;
151   neighbours[received_id].flag = 1;
152   neighbours[received_id].gradient = received_gradient;
153   neighbours[received_id].localised_flag = received_local_flag;
154   neighbours[received_id].timestamp = kilo_ticks;
155   neighbours[received_id].shape_index = received_shape_index;
156 }
157 
158 //helper function to discard any robot with whom he could not communicate in the last 3
      seconds
159 void eliminateNeighbour()
160 {
161   for(uint8_t i = 0; i < MAXIM; i++)
162   {
163     if(kilo_ticks > (neighbours[i].timestamp + 96) )
164     {
165       neighbours[i].dist = -1;
166       neighbours[i].flag = -1;
167       neighbours[i].gradient = -1;
168       neighbours[i].localised_flag = -1;
169       neighbours[i].timestamp = -1;
170       neighbours[i].shape_index = 1000;
171     }
172   }
173 }
174 
175 //data structure to encapsulate the concept of a point in space
176 struct point
177 {
178   float x;
179   float y;
180   int dist;
181 };
182 
183 void setup()
184 {
185   for(uint8_t i = 0; i < MAXIM; i++)
186   {
187     neighbours[i].flag = -1;
188     neighbours[i].dist = -1;
189     neighbours[i].timestamp = -1;
190     neighbours[i].gradient = -1;
191     neighbours[i].shape_index = 1000;
192   }
193 
194   status = WAIT;
195   if(kilo_uid <= 3)
196   {
197     gradient = 0;
198     localised_flag = 1;
199     status = LOCAL;
200   }
201   else
202   {
203     gradient = 255;
204     localised_flag = 0;
```

```
205   }
206
207   switch (kilo_uid)
208   {
209   case 0:
210     shape_index = 0;
211     break;
212   case 1:
213     shape_index = 1;
214     break;
215   case 2:
216     shape_index = 2;
217     break;
218   case 3:
219     shape_index = 3;
220     break;
221
222   default:
223     shape_index = 1000;
224   }
225
226   set_message();
227 }
228
229 void loop()
230 {
231   //check for incoming messages and update the neighbours array
232   if(new_message == 1)
233   {
234     new_message = 0;
235     last_update = kilo_ticks;
236     set_neighbour();
237     if(received_local_flag == 1)
238     {
239       distance_array[received_shape_index] = received_dist;
240     }
241     message_hop++;
242   }
243
244   //check if the number of total messages has been reahced
245   if(message_hop == total_message_hop)
246   {
247     message_hop = 0;
248     int dist_min = 200;
249     int gradient_min = 254;
250     int gradient_max = 0;
251
252     int valid = 0;
253     struct point aux[4];
254     int index = 0;
255     for(uint8_t i = 0; i < MAXIM; i++)
256     {
257       if(neighbours[i].dist != -1 && neighbours[i].dist < dist_min)
258         dist_min = neighbours[i].dist;
259       if(neighbours[i].dist < 55 && neighbours[i].gradient != -1 && neighbours[i].gradient <
      gradient_min)
260           gradient_min = neighbours[i].gradient;
261       if(neighbours[i].dist < 55 && neighbours[i].gradient != -1 && neighbours[i].gradient >
      gradient_max)
262           gradient_max = neighbours[i].gradient;
263     }
264
265     //update the gradient
266     if(kilo_uid > 3)
267       gradient = gradient_min + 1;
268
269     //edge detection part
270     if(kilo_uid > 3 && gradient > gradient_max)
271       edge_wait++;
272     else if(kilo_uid > 3)
273       edge_wait = 0;
274
275     if(edge_wait > 32 && status == WAIT)
276       status = MOVING;
277
278     //update the message that will be sent to other robots
```

```
279     set_message();
280
281     // switch (gradient)
282     // {
283     // case 0:
284     //  set_color(RGB(0,1,1));
285     //  break;
286     // case 1:
287     //  set_color(RGB(1,1,0));
288     //  break;
289     // case 2:
290     //  set_color(RGB(0,0,1));
291     //  break;
292     // case 3:
293     //  set_color(RGB(1,0,0));
294     //  break;
295     // case 4:
296     //  set_color(RGB(1,0,1));
297     //  break;
298     // case 5:
299     //  set_color(RGB(0,1,0));
300     //  break;
301
302     // default:
303     //  set_color(RGB(0,0,0));
304     //  break;
305     // }
306
307     //depending on the status of the robot either orbit around the group or chenge the color
      of the led
308     if(kilo_uid >= 3 && dist_min < 200 && (status == MOVING || status == INSIDE_SHAPE))
309       orbit(dist_min);
310     else
311       set_motion(STOP);
312
313     if(status == LOCAL)
314       set_color(RGB(3,0,3));
315
316     if(status == INSIDE_SHAPE)
317       set_color(RGB(0,3,3));
318
319     //check for a valid position in the shape
320     if (kilo_uid > 3 && status != LOCAL)
321     {
322       float aux_dist;
323       int index;
324       for(int i = 0; i < rowsShape; i++)
325       {
326         char ok = 1;
327         for(int j = 0; j < colsShape; j = j + 2)
328         {
329           aux_dist = shape_array[i][j + 1] * 50;
330           index = shape_array[i][j];
331           if( (distance_array[index] < ( aux_dist - 10 )) || (distance_array[index] > (
      aux_dist + 10 )))
332             ok = 0;
333         }
334
335         if(ok == 1)
336         {
337           shape_index = i + 4;
338           status = LOCAL;
339           localised_flag = 1;
340           orbit(dist_min);
341           set_message();
342
343           break;
344         }
345       }
346     }
347   }
348   eliminateNeighbour();
349 }
350
351 //function for sending messages
352 message_t* message_tx()
```

```
353 {
354   return &message;
355 }
356
357 //function for receiving messages
358 void message_rx(message_t *m, distance_measurement_t *d)
359 {
360     new_message = 1;
361     received_dist = estimate_distance(d);
362     received_id = m->data[0];
363   //received_timestamp = m->data[1];
364   received_gradient = m->data[2];
365   received_shape_index = m->data[3];
366   received_local_flag = m->data[5];
367 }
368
369 int main()
370 {
371     kilo_init();
372     kilo_message_rx = message_rx;
373     kilo_message_tx = message_tx;
374
375     kilo_start(setup, loop);
376
377     return 0;
378 }
```

Anexa E
Code for Matrix Generation

```
#include<iostream>
#include<fstream>

using namespace std;

//data structure for encapsulating an element inside the matrix
struct Index
{
  int i;
  int j;
  int id;

  Index()
  {
    i = j = 0;
    id = -1;
  }

  void set(const int& a, const int& b, const int& new_id)
  {
    i = a;
    j = b;
    id = new_id;
  }

  friend ostream& operator<<(ostream& os, const Index& Aux)
  {
    os << Aux.id << " -> " << Aux.i << " " << Aux.j << endl;
    return os;
  }
};

int main()
{
  //files used for input and output
  ifstream f("plane.txt");
  ofstream g("matrix.txt");
  ofstream h("only_matrix.txt");

  int n;
  int m;
  int nodes = 0;

  f >> n >> m;
  cout << n << " " << m;

  // the original image
  int map[300][300];
  //an array to copy the original image only with the seed robots and the starting point
  int matrix[300][300];
  //an array to store the matrix that will be sent to the robots
  float shape_index[2000][4];

```

```
54  Index seed[4];
55  Index start_point;
56
57  //read the image frim the file
58  for (int i = 0; i < n; i++)
59    for (int j = 0; j < m; j++)
60    {
61      f >> map[i][j];
62      if (map[i][j] == 7)
63        map[i][j] = -1;
64    }
65
66  for (int i = 0; i < n; i++)
67    for (int j = 0; j < m; j++)
68    {
69      matrix[i][j] = map[i][j];
70
71      if (map[i][j] == -1 || map[i][j] == 4)
72      {
73        matrix[i][j] = 0;
74        nodes++;
75      }
76
77      if (map[i][j] == 1 || map[i][j] == 2 || map[i][j] == 3)
78        seed[map[i][j] - 1].set(i, j, map[i][j]);
79
80      if (map[i][j] == 4)
81        start_point.set(i, j, 4);
82    }
83
84  cout << endl;
85  cout << nodes << endl;
86  cout << start_point.i << " "<< start_point.j << endl;
87
88  int valid_nodes = 0;
89  int i;
90  int j;
91  bool ok = false;
92  int vertex = 0;
93  int current_node = 0;
94  int id = 4;
95
96  //check a the point in the original matrix
97  while (valid_nodes != nodes && current_node < nodes && start_point.i < (n - 2) &&
      start_point.j < (m  - 2) && start_point.i > 1 && start_point.j > 1 && start_point.id < (
      nodes + 4) && start_point.id >= -1)
98  {
99    i = start_point.i;
100   j = start_point.j;
101   vertex = 0;
102
103   //cout << start_point << endl;
104
105   //check the 8 elements surrounding the starting point
106   //when one valid element is found update the shape_index
107   for (int a = (i - 1); a < (i + 2); a++)
108     for (int b = (j - 1); b < (j + 2); b++)
109     {
110       if (matrix[a][b] != 0 && vertex < 4)
111       {
112         shape_index[current_node][vertex++] = matrix[a][b]; //the index
113         shape_index[current_node][vertex++] = ((a + b) % 2 == (start_point.i + start_point.j
      ) % 2) ? 1.41 : 1; //the distance
114       }
115     }
116
117
118   if (vertex == 4)
119   {
120     valid_nodes++;
121     matrix[i][j] = start_point.id;
122     map[i][j] = start_point.id;
123     current_node++;
124   }
125
126   int aux_vertex = 0;
```

```
127     ok = 0;
128     for (int i = 1; i < n - 2; i++)
129     {
130       for (int j = 1; j < m - 2; j++)
131       {
132         if (map[i][j] == -1)
133         {
134           //check if point inside map can be the new starting point
135           aux_vertex = 0;
136           for (int a = (i - 1); a < (i + 2); a++)
137             for (int b = (j - 1); b < (j + 2); b++)
138             {
139               if (matrix[a][b] != 0 && aux_vertex < 4)
140               {
141                 aux_vertex++;
142                 aux_vertex++;
143               }
144             }
145
146           //if point has neighbours inside the auxiliary matrix set it as the new starting
      point
147           if (aux_vertex == 4)
148           {
149             id++;
150             start_point.set(i, j, id);
151             ok = 1;
152             break;
153
154           }
155         }
156       }
157
158       if (ok == 1)
159         break;
160     }
161   }
162
163   //write the matrix to file
164   for (int i = 0; i < nodes; i++)
165   {
166     g << i + 4 << " : ";
167     for (int j = 0; j < 4; j++)
168       g << shape_index[i][j] << " ";
169     g << endl;
170   }
171
172   //write the matrix containing the order of the robots in another file
173   for (int i = 0; i < n; i++)
174   {
175     for (int j = 0; j < m; j++)
176     {
177       h << map[i][j] << " ";
178     }
179     g << endl;
180     h << endl;
181   }
182
183   cout << endl;
184   system("pause");
185   return 0;
186 }
```

Anexa F
Code for Ramer-Douglas-Peucker Algorithm

The code for generating the reduced data set

```
clear;
Img = imread('plane.jpg');

figure(1);
imshow(Img);

%remove the background
S = size(Img);
for i = 1:S(1)
    for j = 1:S(2)

        if( Img(i,j, 3) > 150 && (Img(i,j, 1) < 100 || Img(i,j, 2) < 100))
           Img(i,j, 1) = 0;
           Img(i,j, 2) = 0;
           Img(i,j, 3) = 0;

        end
    end
end

%make the picture grayscale and apply a median filter
Gray =  0.299*Img(:,:,1) + 0.587*Img(:,:,2) + 0.114*Img(:,:,3);

S = size(Gray);
Gray = medfilt2(Gray);

%binarize the image
for i = 1:S(1)
    for j = 1:S(2)

        if( Gray(i,j) < 50)
            Gray(i,j) = 0;
        end
    end
end


for i = 1:S(1)
    for j = 1:S(2)

        if( Gray(i,j) > 40)
            New_Matrix(i,j) = 7;
        else
            New_Matrix(i,j) = 0;
        end
    end
end

figure(10);
%extract the contour
[C,H] = imcontour(New_Matrix,1);
```

```
52
53 S = size(C);
54 for i = 1:S(1)
55     for j = 2:S(2)
56         Aux(i,j - 1) = C(i,j);
57     end
58 end
59
60 figure(11);
61 plot(Aux(2,:), Aux(1,:))
62 S = size(Aux);
63 S(2) = S(2) - 2;
64 Points_plane = Aux(: , 1:S(2));
65 Points_plane = Points_plane';
66
67 %dispaly the original contour points
68 figure(1)
69 plot(Points_plane(:,2) ,Points_plane(:,1), 'ko');
70 hold on
71 plot(Points_plane(:,2) ,Points_plane(:,1));
72
73 %display the reduced data set after RDP with epsilon = 0.5
74 epsi = 0.5
75 Result1 = RDP(Points_plane, epsi);
76 Result1 = unique(Result1,'rows','stable');
77
78 figure(2)
79 plot(Result1(:,2) ,Result1(:,1), 'ko');
80 hold on
81 plot(Result1(:,2) ,Result1(:,1));
82
83 %display the reduced data set after RDP with epsilon = 5
84 epsi = 5
85 Result = RDP(Points_plane, epsi);
86 Result = unique(Result,'rows','stable');
87
88 figure(3)
89 plot(Result(:,2) ,Result(:,1), 'ko');
90 hold on
91 plot(Result(:,2) ,Result(:,1));
```

The function for Ramer-Douglas-Peucker algorithm

```
1  function [ResultList] = RDP(Points,epsilon)
2
3      dmax = 0;
4      index = 0;
5      S = size(Points);
6      end_point = S(1);
7      Line = [Points(1,:);Points(end_point,:)]
8      n = (S(1) - 1);
9
10     for i = 2:n
11         d = perpendicularDistance( Points(i,:), Line);
12
13         if d > dmax
14             dmax = d;
15             index = i;
16         end
17     end
18
19     ResultList = zeros(end_point, 2);
20     if ( dmax > epsilon)
21         recursiveResult1 = RDP( Points(1:index, :), epsilon);
22         recursiveResult2 = RDP( Points(index:end_point, :), epsilon);
23
24         ResultList = [recursiveResult1; recursiveResult2]
25     else
26         ResultList = [ Points(1,:);Points(end_point,:) ] ;
27     end
28
29 end
```