FINAL YEAR PROJECT SIMULATING ROBOT SWARMS: β-ALGORITHM

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2025

Vansh Goenka

Supervised by Prof. Clare Dixon

Department of Computer Science

Contents

Word Count: 12345

List of Tables

List of Figures

Abstract

Swarm robotics offers robust and adaptable systems through the coordinated actions of multiple simple robots. A critical aspect of developing swarm robotics is coherence, which is the ability of the swarm to maintain a unified and organized structure (just like flocks of birds). This project involves around coherence and investigates various studied algorithms for attaining coherence. The research begins with a comprehensive literature review on swarm robotics, primarily focusing on such algorithms and possible optimizations for better performance. The Algorithm is then implemented in the desired simulation software which provides customizable parameters needed for experimentation. This approach allows us to systematically investigate how changes made to these parameters impact swarm coherence. Metrics such as convergence time and Cadence are used to evaluate swarm behavior. This tests the model's adaptability and flexibility. This research concludes with an understanding of the Algorithm's functionality and its potential for enhancing the control and coordination of swarm robotics, which offers a foundation for future extensions and real-world applications.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor, Professor Clare Dixon. This project would not have been possible without her valuable advice and constant support.

Chapter 1

Introduction

This chapter presents an overview of swarm robotics, explaining the driving force behind this project and the broader field of research. Subsequently, it lists the aims and objectives of the project.

1.1 Motivation

Robotics stands at the forefront of innovation, offering solutions for handling complex tasks across diverse fields. While significant advancements have been made in single-robotic systems, the potential of coordinated multi-robot systems presents a compelling direction in the robotic industry. This is where Swarm robotics comes in. The main inspiration for swarm robotics comes from the collective behavior of biological systems such as colonies of ants and schools of fish**[24]**. A number of relatively simple robots, each with limited sensing, cognition, and actuation capabilities, collectively work together**[37]** to complete a shared task. The most important thing about swarm robotics is that all the sensing and communication is done locally by the robots **[24]**.

Swarm robotics holds immense potential for real-life applications across diverse fields. This could range from environmental monitoring and disaster response such as search & rescue **[19]** to precision agriculture and warehouse logistics **[46]**. This wide range of real-life applications underscores the field's attractiveness. However, some fundamental challenges stand in the way of achieving these goals. One of them is swarm coherence **[37]**. This ability of the swarm to act in a unified and organized manner through local connectivity is critical for effective task execution. Without well-coordinated collective behavior, the potential of swarm robotics cannot be fully uncovered. Dealing with the complexities of swarm coherence requires innovative solutions. In this project, I aim to investigate some of these challenges, parallelly seeking insights that might contribute to the development of more effective and adaptable swarm systems. This report focuses on how swarm coherence can maintain large numbers of autonomous robots (e.g., drones) in a certain fixed formation. This has significant potential for applications such as drone shows and automated swarms in strategic military operations **[42]**.

1.2 Aim

To conduct a systematic investigation of the Beta Algorithm **[37]**, including the manipulation of its parameters to identify optimal configurations for maximizing swarm robotics coherence. This investigation will utilize metrics such as convergence time and stability and follow an iterative process, allowing for parameter refinement based on experimental results. The project will also explore the potential integration of Stoy's Algorithm to further enhance swarm performance.

1.3 Objectives

To achieve the aim of the project, the following objectives are set into play:

To achieve the aim of the project, the following objectives are set into play:

- Conduct a comprehensive review of existing literature on swarm robotics, with a focus on the β algorithm and related approaches.
- Setting up the simulator with required agents including an arena, robots and boundaries
- Defining Robot behavior which includes avoiding obstacles and communication
- Implement the original Beta Algorithm in simulation.
- Design experiments to test the impact of neighbor count (Beta values) on swarm coherence.
- Define metrics for evaluating swarm behavior (e.g., convergence time, Cadence)

- Test scalability with varying swarm sizes and parameters.
- Documentation and Reporting: Document all details and prepare a comprehensive research report

Chapter 2

Background

In this chapter we delve deeper into the background & history of swarm robotics, providing the concepts that would be crucial for a better understanding of the following chapters. This will include a review of relevant literature, focusing on swarm coherence and its importance for effective task execution.

2.1 History

Swarm intelligence is a concept where complex behaviors emerge from simple individual interactions. It finds its earliest examples in social insects such as ants, bees, and termites. Despite their limited individual capabilities, they demonstrate remarkable collective feats. Colonies can efficiently locate and exploit food sources, construct intricate nests, and adaptively respond to threats without centralized leadership **[25]**. These intricate behaviors fascinated early researchers, inspiring the field of swarm robotics.

Early swarm robotics research drew heavily on the concept of stigmergy, a form of indirect communication where robots interact by modifying their environment to influence the behavior of others **[43]**. Ants provide a classic example; they leave pheromone trails, guiding other ants to food sources. Similarly, ants selectively deposit pheromones to attract caretakers towards healthy larvae during brood sorting **[28]**. Researchers developed algorithms inspired by these principles. Robots might utilize virtual pheromones which are digital markers to signal explored areas or discovered resources. However, stigmergy-based approaches have their limitations. Since the communication is indirect, it can be slow and inflexible, hindering rapid adaptation

[43]. Designing effective robot behaviors based solely on manipulating the environment can also prove challenging, particularly in dynamic settings where changes occur quickly. Recognizing these limitations, the field of swarm robotics expanded significantly to include direct communication among robots **[51]**. This pivotal shift mirrored the forms of communication seen in some social insects, where individuals might exchange information through physical contact (like antennation in ants) or even produce sounds or vibrations **[30]**. In swarm robotics, direct communication enables robots to exchange information about their locations, sensor readings, and task progress in real-time. This newfound ability allows for more complex and responsive behaviors, enabling the swarm to adapt to unforeseen challenges in their environment fluidly **[21]**.

2.2 Literature Review

One of the early examples of swarm robotics research is the seminal work by Nembrini & Alan Winfield, where robots mimic the behavior of ants exploring their surroundings. In this approach, robots utilized a form of ”virtual pheromones,” depositing digital markers to signal explored areas, demonstrating a basic form of swarm intelligence **[37]**. The focus of swarm robotics has since broadened, with researchers developing more sophisticated communication and control systems to address the challenges faced for real-world deployment **[43]**. Key benefits of swarm systems that are used in modern approaches include:

- **Robustness:** Failure of individual robots does not hinder the overall function of the swarm. This also does not affect the tasks of other robots within the swarm **[43]**. This is analogous to how a flock of birds can still navigate effectively even with some of the individuals lost from the flock [20].

- **Adaptability**: Robots in some swarm systems use decentralized controls to analyze their environment. This does not require a centralized leader which could potentially cause a bottleneck. This flexibility can also be seen in colonies of ants, where they adjust their foraging patterns in response to shifting food sources **[38]**.

- **Scalability**: Many swarm algorithms are designed to function effectively across different swarm sizes **[43]**. This allows them to be deployed for tasks on varying scales (swarm sizes), much like how a bee colony can efficiently collect nectar with a few dozen or a few thousand individuals **[47]**.

Using this foundation, researchers like Nembrini & Winfield (2008) developed sophisticated communication and control for swarm systems. Their introduction of the β-Algorithm marked a significant step toward swarm coherence **[37]**. Coherence enabled adaptability through localized wireless communication and will be discussed in detail later.

2.2.1 Limitations of Early Foraging Algorithms

Challenges faced by swarm foraging were tackled by using systems that used swarm coherence instead. Early foraging algorithms were promising in simple environments but struggled in complex and unpredictable scenarios **[37]**. Foraging is very useful for resource collection, so when one of the robots loses its connectivity, it hinders resource collection thereby failing to complete its main task. Now imagine a swarm of robots tasked with locating and retrieving scattered resources. If they become disorganized and disconnected then they would spend significant time wandering rather than doing the intended task.

2.2.2 The Alpha Algorithm: Centralized Swarm Control

Due to the challenges faced by the foraging algorithm, new strategies were required for maintaining swarm connectivity. As mentioned previously, Nembrini & Winfield introduced two novel algorithms: the α-Algorithm and the β-Algorithm **[37]**. Both use direct communication for swarm coherence but achieve coherence differently. The pseudocode for α-Algorithm is given in Figure 2.1. The detailed explanation of α-Algorithm is discussed below:

1. **Threshold-Based:** Each robot has a threshold α. For each robot, if the number of connections to other robots falls below this threshold, the robot takes action to move and try to restore those connections.

2. **Neighbor Following:** There is no single leader and all robots operate according to the same algorithm.

3. **Goal:** The algorithm aims to promote swarm cohesion. The robots stay within a certain range of each other. It does not necessarily have a specific shape or movement pattern for the swarm.

```
Create list of neighbours for robot, Nlist
k = number of neighbours in Nlist
i = 0

loop forever {
  i = i modulo cadence

  if (i = 0) {
    Send ID message

    Save copy of k in LastK
    k = number of neighbours in Nlist

    if ((k<lastK) and (k < alpha)) {
      turn robot through 180 degrees
    }
    else if (k > LastK) {
      make random turn
    }
  }

  Steer the robot according to state
  Listen for calls from robots in range
  Grow Nlist with neighbours IDs

  i++
}
```

Figure 2.1: Pseudo-code for the α-algorithm **[37]**

2.2.3 Limitations of the Alpha Algorithm

The α algorithm offers a simple and intuitive approach to swarm coherence but can only be effective when the environment is controlled. It has some limitations that prevent it from being used in real-world applications. These will be discussed now. The algorithm is not vulnerable to the failure of each robot. However, if the count of robots that are failing or losing connection is big enough, then it can degrade the overall swarm cohesion. In large swarms, maintaining a constant connection requires constant readjustments and might lead to communication overhead making the system inefficient. If the environment changes radically i.e. if an obstacle appears suddenly, it could lead to the breakdown of swarm structure.

2.2.4 The Beta Algorithm: A Focus on Coherence

To address the challenges faced by earlier swarm systems, the β-Algorithm **[37]** introduced by Nembrini & Winfield is investigated in this project. The pseudocode for the algorithm is given in Figure 2.2. Robots have wireless communication, which allows them to broadcast information and receive messages from other robots within a certain range. This communication range is a crucial parameter in the algorithm. It determines the immediate ’neighborhood’ for each robot. Robots continuously share lists of their detected neighbors with each other. This exchange allows each robot to build an understanding of the swarm structure (i.e. which robots are within its range, and which robots are out of their range). These neighbors are also used to check for connectivity when one of the robots loses its connection. Each robot has a ’β’ threshold. This is the core parameter that influences the algorithm’s behavior. When a robot loses connection, it first checks how many of its remaining neighbors can still detect the lost robot. If this number of neighbor robots falls below the threshold β, the robot performs a 180-degree turn **[37]**.

```
Create list of neighbours for robot, Nlist
k = number of neighbours in Nlist
i = 0

loop forever {
  i = i modulo cadence

  if (i = 0) {
    Send ID message

    Save copy of k in LastK
    Set reaction indicator Back to FALSE
    k = number of neighbours in Nlist
    Create LostList comparing Nlist and OldList

    for (each robot in LostList) {
      Find nShared, number of shared neighbours
      if (nShared <= beta) {
        Set reaction indicator Back to TRUE
      }
    }
    if (Back = TRUE) {
      turn robot through 180 degrees
    }
    else if (k > LastK) {
      make random turn
    }
    Save copy of Nlist in Oldlist
  }

  Steer the robot according to state
  Listen for calls from robots in range
  Grow Nlist with neighbours IDs and connection info

  i++
}
```

Figure 2.2: Pseudo-code for the β-algorithm **[37]**

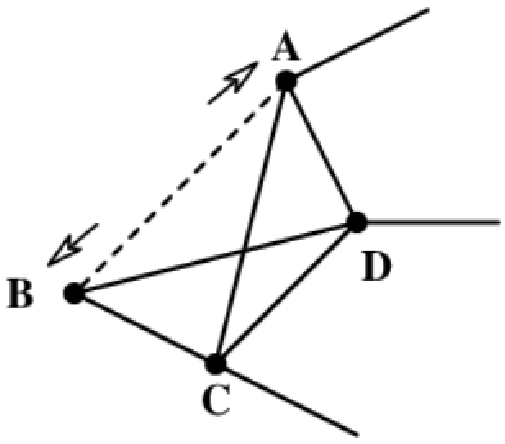


Figure 2.3: Shared Neighbor **[37]**

2.2.5 Significance of the Beta Algorithm

The β-Algorithm offers an adaptable approach to construct a decentralized swarm system. It emphasizes localized communication making it a robust option for when centralized control is impractical or unavailable. The algorithm's core concept of 'shared neighbor' (see Figure 2.3) count provides a simple yet effective tool for robots to assess local connectivity.

2.2.6 Further Considerations

While the Beta Algorithm offers a significant step forward in the world of swarm coherence, it's important to consider potential limitations and areas of improvement:

- **Communication Constraints:** The algorithm's effectiveness relies solely on reliable wireless communication. Environments with interference could make the communication unreliable, thereby affecting performance.
- **Parameter Tuning:** Finding the optimal 'β' value and swarm sizes can require careful experimentation and simulation.
- **Adaptation to Dynamic Environments:** While the Beta Algorithm provides a foundation for adaptability, it still needs further refinements for handling rapidly changing scenarios and highly unpredictable tasks.

Chapter 3

Design

The concepts introduced in the previous chapter outline the requirements for the simulation. This chapter is meant to present the requirements, address the design of the swarm robots simulation, and discuss the foraging algorithm.

3.1 Requirements

This section presents the requirements needed for implementing the simulation of β-Algorithm :

- The robots are placed in an Arena which have defined boundaries
- The placement of the robots are randomized
- Each of the robots has a number of sensors that are useful for detecting collisions and getting the position of each robot
- Each robot also has a transmitter and receiver to send and receive messages from each other
- The communication between the robots is the most crucial part of testing the algorithm
- If a robot goes beyond the communication range, it should communicate with other robots on what to do based on the algorithm
- The robot turns 180° if its position in the arena with respect to the other robots satisfies the algorithm

- Otherwise, the robot prints out lost communication

3.2 Design

The simulation uses an object-oriented approach to achieve a modular and scalable design. This approach decomposes the system into four well-defined classes each encapsulating specific functionalities and most importantly interacting with each other through well-defined interfaces (see Figure 3.1). The key classes include:

- **Robot:** This class represents an individual robot within the swarm. It encapsulates the robot's state (including position, sensors, neighbor list, and communication capabilities) and behaviors (such as movement, message handling, and the implementation of the beta algorithm logic). The robot's control system is inspired by finite state automata (FSA) **[37]**, and can be modeled as a collection of states and transitions between those states. The state transitions are triggered by events perceived by the robot's sensors or messages received from neighboring robots.

- **Swarm:** This class manages the creation, initialization, and overall behavior of the robot population. It is responsible for creating robot instances, placing them within the environment, and potentially coordinating their actions towards a collective goal.

- **Arena:** This class defines the physical environment within which the robots operate. It represents the boundaries of the simulation space and can optionally include obstacles or other environmental features that may influence robot behavior.

- **Sensors:** This class simulates the sensory input received by the robots. It might encompass functionalities for modeling proximity sensors, light sensors, or any other sensory modality relevant to the specific application.

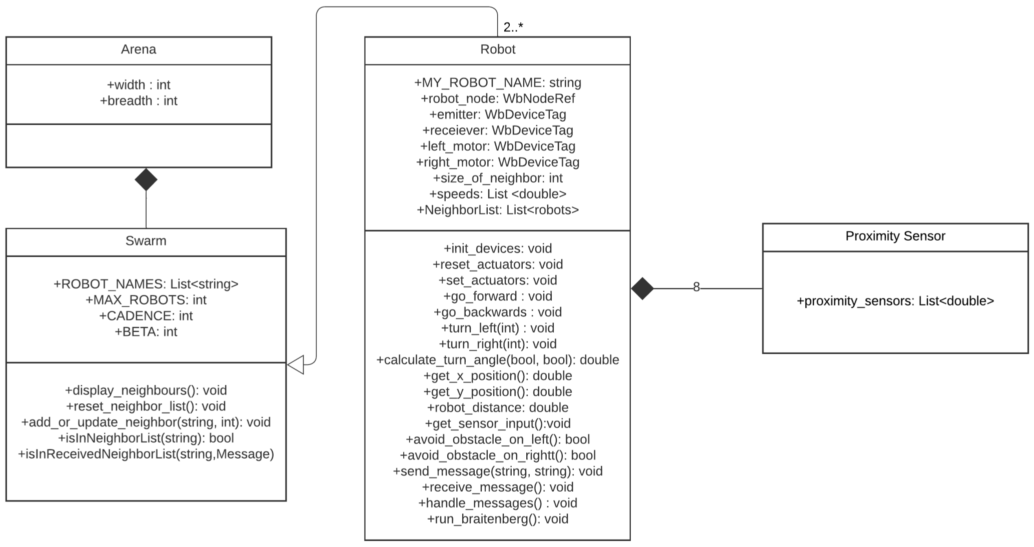


Figure 3.1: UML diagram

3.3 The β-Algorithm

The implementation of the beta algorithm is inspired by the pseudocode given in **[37]** in Figure 2.2. Robots periodically broadcast messages containing their unique identifier and a list of their currently perceived neighbors. This information exchange allows robots to maintain awareness of each other's relative positions within the swarm. The frequency of these messages is determined by the `CADENCE` parameter, which essentially sets the communication heartbeat of the swarm. A higher `CADENCE` value translates to more frequent updates and potentially faster response times but also increases communication overhead.

Upon receiving a heartbeat message from a neighbor, a robot updates its own neighbor list (see Figure 3.1). This list keeps track of all robots currently within communication range, based on a predefined distance threshold `THRESHOLD_FOR_DISTANCE`. Essentially, if the distance between two robots falls below this threshold, they are considered neighbors and add each other to their respective neighbor lists. Robots periodically assess their connectivity with neighboring robots. The `CADENCE` parameter triggers this assessment and involves comparing the received neighbor lists against the robot's list. Suppose a previously known neighbor is missing from the received messages, and the number of shared neighbors with that robot falls below a predefined threshold `BETA`.

In that case, the robot infers a potential loss of connection. If a loss of connection is inferred, the robot initiates a corrective action, typically a turning maneuver, to attempt to re-establish communication with its lost neighbor(s). The specific maneuver chosen might depend on the robot's design and the control logic implemented within the `Robot` class.

The robots in this simulation employ a Braitenberg-inspired approach (see Figure 3.4) for navigation and obstacle avoidance. Valentino Braitenberg's Vehicles **[22]** demonstrated that surprisingly complex behaviors can emerge from simple, direct connections between sensors and actuators. In this simulation, each robot possesses a set of proximity sensors. These sensors provide readings that indicate the distance to potential obstacles. The key to Braitenberg-style control lies in how these sensor readings are translated into motor actions. Rather than mapping sensor values directly to motor speeds, a weighting scheme influences their contribution. For instance, higher readings from left-facing sensors could translate to a stronger rightward bias in the motor control, encouraging the robot to turn away from a left-side obstacle. Conversely, high readings on the right side would promote a leftward turning preference. To promote overall movement, a slight baseline forward motion bias can be added. This helps prevent the robot from stalling in obstacle-free environments. The actual turning and forward movements are handled through dedicated movement functions (`turn_left`, `turn_right`, and `go_forward`). For instance, within a differential drive model, `turn_left` would increase the speed of the right motor relative to the left. The magnitude of the speed differences between the motors is likely influenced by the weighted sensor inputs. Importantly, Braitenberg-based navigation excels due to its simplicity and emergent qualities. The direct sensor-to-motor mappings ensure minimal computational overhead. As a result of this, there's no need for the robot to build an internal representation or map of its surroundings.

While the beta algorithm focuses on maintaining swarm connectivity, a separate but interwoven system handles obstacle avoidance. It is achieved by using the set of 8 proximity sensors placed strategically around the robot. There are two functions, `avoid_obstacle_on_left` and `avoid_obstacle_on_right` that detect whether there is an obstacle on the left and right side of the robot respectively. Upon some trial and error, a threshold value of `80.0` was decided as the `THRESHOLD` value for the proximity sensors for detecting the walls of the arena and robots. The readings from these helper functions are then used by the `run_braitenberg` function to navigate through any possible obstacles. If `avoid_obstacle_on_left` is `TRUE`, then the robot makes a right turn and if `avoid_obstacle_on_right` is `TRUE` then the robot takes a left turn. The angle of the turn is calculated by the function `calculate_turn_angle`. This function returns a large angle if there is an obstacle only on one side otherwise it returns a small angle. Initially, there was no check for corner cases, as a result, the robots were getting stuck in the corners of the rectangular arena as they failed to decide which turn to make (left or right). To counter this, a check for corner case was added. The variable `corner_case`, checks if 6 of the 8 proximity sensors (i.e sensors 1,2,4,5,7,8) are returning a value greater than the `THRESHOLD`, which means that there is an obstacle on both sides of the robot. If that is `TRUE`, the robot moves backward and turns left. The placement of the sensors will be discussed in detail in Chapter 4 and the reason for using only these specific sensors is discussed in Section 4.7. The Obstacle avoidance can detect walls easily, but sometimes it fails to detect other robots, and as a result, clash with each other. Interestingly, the initial avoid detection (see Figure 3.3) had better wall and robot detection. It used a turn counter to handle the corner cases and employed the function `passive_wait()` to halt any other process while making a turn.

```
method beta algorithm:
    THRESHOLD_FOR_DISTANCE = 30.0
        if heartbeat message is received:
            distance = calculate distance between sender and receiver
            update the neighbor list for the current robot
            if cad_count modulo CADENCE equals 0:
                for each robot name in ROBOT_NAMES:
                    if the robot is not in the NeighborList:
                        nShared = count robots in NeighborList
                        if nShared <= BETA:
                            turn 180 degrees
                            initiate braitenberg behavior
        cad_count = cad_count + 1
```

Figure 3.2: Pseudocode for the Beta Algorithm

```
method run_braitenberg:
    send a heartbeat message at each time step
    state = 0
    turn_counter = 0
    obstacle_on_left = check for the left obstacle
    obstacle_on_right = check for the right obstacle
    corner_case = check for corner situation

    switch (state):
        case 0:
            if corner_case:
                state = 4
            else if obstacle_on_left:
                state = 2
            else if obstacle_on_right:
                state = 1
            otherwise move forward
            turn_counter = 0
        case 1:
            turn_counter = turn_counter + 1
            keep turning left until turn_counter > TURN_TIMEOUT
            state = 0

        case 2:
            turn_counter = turn_counter + 1
            keep turning right until turn_counter > TURN_TIMEOUT
            state = 0

        case 3:
            move backwards
            wait for a short duration
            turn_counter = 0
            if no obstacle on left or right:
                state = 0

        case 4:
            activate specific LEDs to showcase corner detection
            state = 3
```

Figure 3.3: Pseudocode for the old braitenberg function (uses timeouts)

```
method run_braitenberg:
    send a heartbeat message at each time step
    state = 0
    obstacle_on_left = check for the left obstacle
    obstacle_on_right = check for the right obstacle
    corner_case = check for corner situation

    switch (state):
        case 0:
            if corner_case:
                state = 4
            else if obstacle_on_left:
                state = 2
            else if obstacle_on_right:
                state = 1
            otherwise move forward

        case 1:
            turn left at a calculated angle
            state = 0

        case 2:
            turn right at a calculated angle
            state = 0

        case 3:
            move backwards
            turn left 180 degrees
            state = 0

        case 4:
            activate specific LEDs to showcase corner detection
            state = 3
```

Figure 3.4: Pseudocode for the current braitenberg function (uses angles to make turns)

Chapter 4

Implemenatation

This chapter provides a rationale for the technological tools selected to support the development process. It then carefully explains the implementation, demonstrating its strategic alignment with the requirements, design considerations, and foraging algorithm presented earlier

4.1 Choice of Simulators

To prioritize the development and analysis of the β-Algorithm within the project's timeframe, careful consideration was given to the choice of simulator. The convenience of a pre-existing robot simulator, which comes with features like physics engines, pre-made robot models, and tools for visualization offered an easy workaround to develop and test the algorithm in question. Using a simulator is better than using a programming language since with the use of simulators, you don't need to manually code everything from scratch. However, the simulator does come with some drawbacks such as the potential learning curve associated with an unfamiliar simulator, along with possible limitations in customization. On the other hand, building a simulation from scratch in a familiar programming language would grant maximum control and flexibility, but could significantly increase development time. This chapter examines this crucial trade-off, detailing the factors that influenced the decision. It outlines the reasoning behind the chosen technological framework, explaining how it aligns to efficiently implement and evaluate the swarm connectivity behaviors governed by the beta algorithm.

For this purpose, several simulation software were investigated to test out which one of them suited the best for the purpose. These were Gazebo, NetLogo, Nvidia

Omniverse, and Webots. We investigated each of them one by one based on the features offered by them and concluded with the one that suited the needs of our purpose.

- **Gazebo:** Gazebo [4] is great for creating your own robots from scratch and rendering realistic 3D scenes which allows for a more visually appealing look. The robots can be programmed on either C++ or ROS [40]. However, the problem with Gazebo was that I use a Windows machine and it was only available on Linux and MacOS and not Windows [5]. I tried using Gazebo through a virtual machine running Linux, but the software was too glitchy to be operational.

- **NetLogo:** While NetLogo [8] offers a user-friendly platform for agent-based modeling, its focus on high-level behavioral simulations presented limitations for this project. The successful implementation of the beta algorithm would likely require a more granular level of control over robot movement and their interactions within a potentially physics-simulated environment. Furthermore, NetLogo's visualization capabilities lacked the necessary flexibility for the desired level of robot model customization and realistic rendering.

- **Nvidia Omniverse:** Nvidia Omniverse [9], despite offering powerful tools like ray-tracing for real-time simulation and collaboration, presented a significant complexity barrier. Its extensive feature set and the steep learning curve would likely have extended the development timeline significantly. This complexity felt unnecessary for the project's core focus on implementing and evaluating the beta algorithm.

- **Webots:** Webots [13] offers easy-to-use tools for rapid prototyping, featuring a long list of pre-made robots (including the famous Spot [11] & Atlas [1] from Boston Dynamics [2]) along with all the required sensors and actuators with it. It also has an in-built code compiler for easily editing the robot controllers. It supports several programming languages like (C, C++, Python, Java, MATLAB).

Several of the candidate simulators offered comprehensive functionalities, but their complexity could have resulted in a significant investment of time and effort for learning. Webots, in contrast, presented a user-friendly interface and intuitive design principles. This streamlined approach facilitated a rapid understanding of the core concepts and functionalities, enabling a more efficient development process. Consequently, Webots emerged as the clear choice for the primary simulation platform in this project.

Webots **[13]** is a professional-grade robot simulator offering integrated tools for modeling, programming, and simulating mobile robots. It was chosen for this project due to its built-in physics engine, pre-designed robot models, and versatile visualization environment. This streamlined workflow accelerated the development process, allowing for a focus on implementing core algorithms and behaviors. Webots provides a user-friendly interface and supports multiple programming languages (C, C++, Python, Java, MATLAB). C **[33]** was chosen in this project as it was the default language for the robot controller used. Converting the code to a different language would have introduced additional development time and potential compatibility issues. Additionally, its active community and extensive documentation provide valuable support resources **[15]**.

4.2 Webots Physics Engine

One of the key factors influencing the choice of Webots was its integration with a robust physics engine. This built-in engine is known as Open Dynamics Engine (ODE) **[26]** and is responsible for simulating the realistic movement of robots and their interactions with the surrounding environment. During simulation, Webots' physics engine accurately calculates the effects of forces (like motor commands from the robot controller) and environmental factors (like friction or gravity) on the robots' motion. This realistic simulation of physics is essential for ensuring that the robots' behavior within the virtual world closely mirrors how they would behave in a real-world setting. This fidelity is particularly important for research involving robot algorithms, as it allows researchers to test and refine their algorithms in a controlled, virtual environment before deploying them on physical robots.

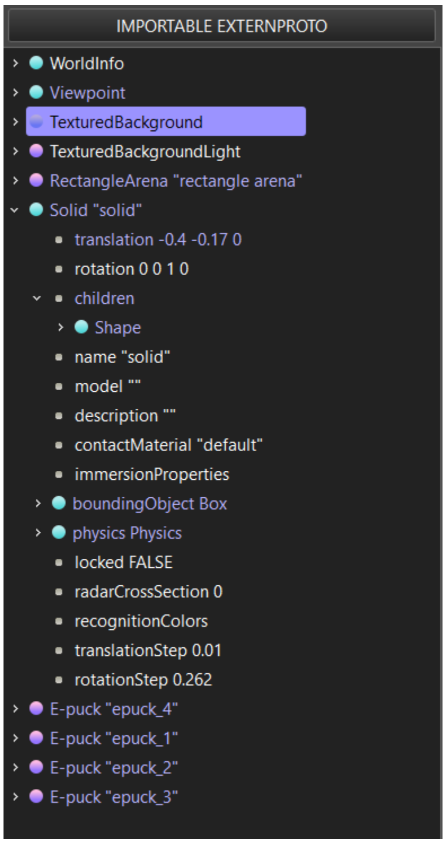
4.3 Simulation Setup

Following the selection of Webots and the decision to use C for programming the robot controller, the first step for setting up the simulation was designing the virtual world. In Webots, it is achieved by first creating a 'world' file. Then to add objects (like arena or robot) to this world file we use two types of nodes, either a Base node or a PROTO node which are arranged in a hierarchical tree structure (see Figure 4.1). We will now discuss how these nodes work in detail.

- **Base Node:** These are predefined elements within Webots that encompass a range of basic components. These include different types of shapes (Box, Cone, Sphere, etc.), Solid objects, chargers for possible robot or vehicle integration, Fluids, and billboards (see Figure 4.2).

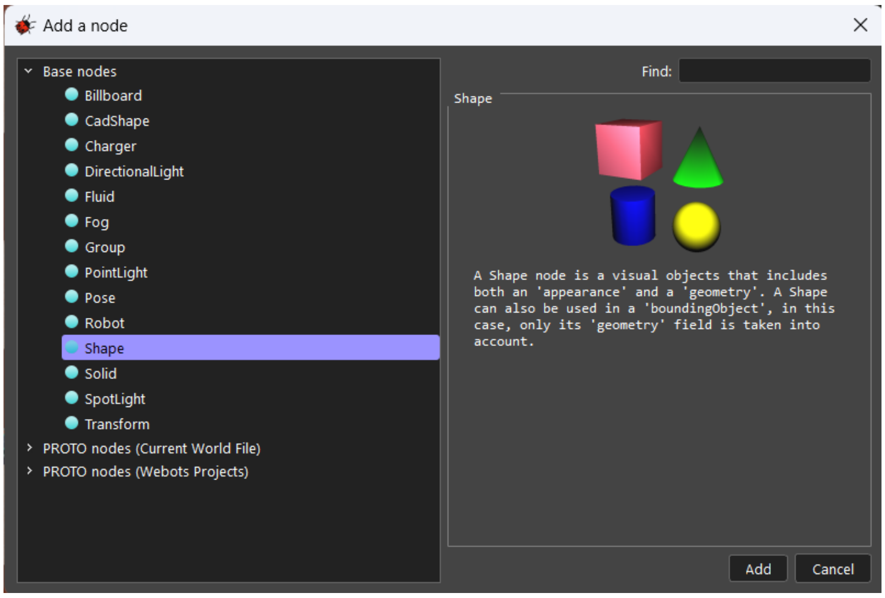
- **PROTO Node:** These are either user-defined templates or templates given by Webots. PROTO nodes **[16]** allow for the creation of customized objects or robot designs that can be reused and instantiated multiple times within the simulation (see Figure 4.3). These customized objects range from simple objects like a football to more complex ones like moving vehicles (e.g. BMW X5) **[12]** and commercial robots (e.g. Nao **[7]**).

The simulation environment was designed and constructed using suitable Base nodes to represent the arena, obstacles, and surfaces that might influence the robot's behavior. An appropriate robot model was selected and placed within the arena. Properties like translation and orientation were adjusted as needed. Finally, the provided controller template in C served as the foundation for development, with essential modifications focused on obstacle detection, sensor data access, motor functions, and integrating the beta algorithm's logic. We will now discuss each of them in detail.


IMPORTABLE EXTERNPROTO

```
› WorldInfo
› Viewpoint
› TexturedBackground
› TexturedBackgroundLight
› RectangleArena "rectangle arena"
⌄ Solid "solid"
    translation -0.4 -0.17 0
    rotation 0 0 1 0
  ⌄ children
    › Shape
    name "solid"
    model ""
    description ""
    contactMaterial "default"
    immersionProperties
  › boundingObject Box
  › physics Physics
    locked FALSE
    radarCrossSection 0
    recognitionColors
    translationStep 0.01
    rotationStep 0.262
› E-puck "epuck_4"
› E-puck "epuck_1"
› E-puck "epuck_2"
› E-puck "epuck_3"
```

Figure 4.1: Tree structure of nodes in Webots Simulator

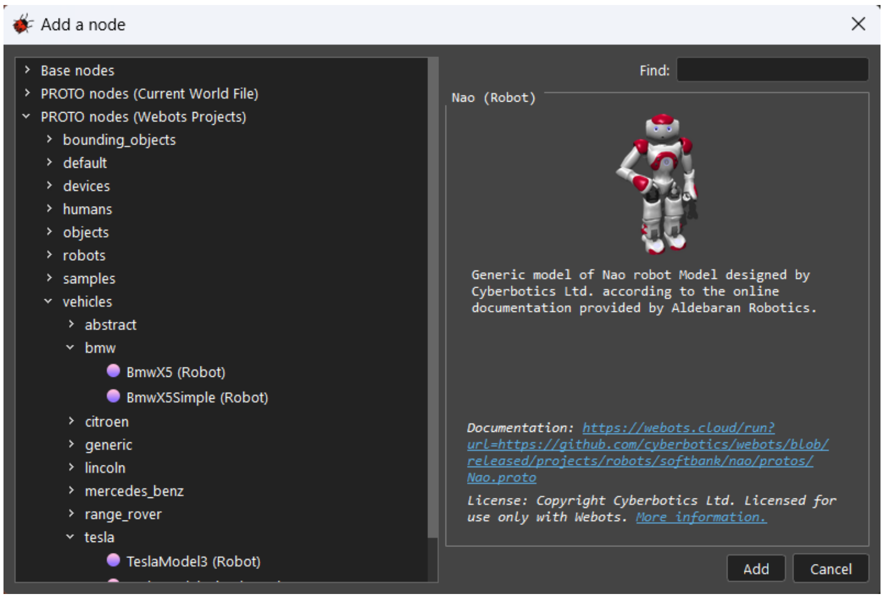


```
⌄ Base nodes
    Billboard
    CadShape
    Charger
    DirectionalLight
    Fluid
    Fog
    Group
    PointLight
    Pose
    Robot
    Shape
    Solid
    SpotLight
    Transform
› PROTO nodes (Current World File)
› PROTO nodes (Webots Projects)
```

Shape

A Shape node is a visual objects that includes both an 'appearance' and a 'geometry'. A Shape can also be used in a 'boundingObject', in this case, only its 'geometry' field is taken into account.

Figure 4.2: Base Node options in Webots Simulator



Generic model of Nao robot Model designed by Cyberbotics Ltd. according to the online documentation provided by Aldebaran Robotics.

Documentation: https://webots.cloud/run?url=https://github.com/cyberbotics/webots/blob/released/projects/robots/softbank/nao/protos/Nao.proto

License: Copyright Cyberbotics Ltd. Licensed for use only with Webots. More information.

Figure 4.3: PROTO Node options in Webots Simulator

4.4 Arena

The simulation arena was initially constructed within Webots using a Solid Base node **[17]**. A rectangular shape was selected via its children nodes and further customized using the Scene Tree editor to establish the desired dimensions and position. This square arena provided the boundary for the robots' operating environment (see Figure 4.4). During preliminary simulations, it was observed that the robots encountered difficulties navigating the square arena's corners, often becoming stuck. When a robot encountered an edge, its turn behavior was intended to allow it to pivot away. However, in the corners its sensors would detect the adjacent walls, resulting in multiple turns that kept the robot trapped. This issue highlights a common challenge in swarm robotics, where corner scenarios often require special consideration **[45]**. To address this cornering problem, two potential solutions were explored. One approach involved modifying the arena's shape to a hexagon. A hexagon's wider angles mitigate the corner problem, allowing robots to maneuver more easily and avoid repeated obstacle detection during turns. Simulations with the hexagonal arena verified that this change resolved the robot blockage issue **[31]**. However, in Webots, only rectangular or circular arenas could be created. An alternative approach was introducing a specific corner-case handling logic into the β -Algorithm. This involves additional sensor checks and modified behaviors after detecting a corner situation. In this case, while the hexagonal arena demonstrated clear advantages in resolving the cornering issue, limitations within Webots' simulator prevented its direct implementation. Therefore, the primary focus shifted towards refining the robot controller to incorporate robust corner-case handling. This approach involved introducing additional sensor checks and specialized movement behaviors tailored to corner situations, allowing the robots to navigate the rectangular arena successfully. The implementation of this is discussed in detail in section 4.7.

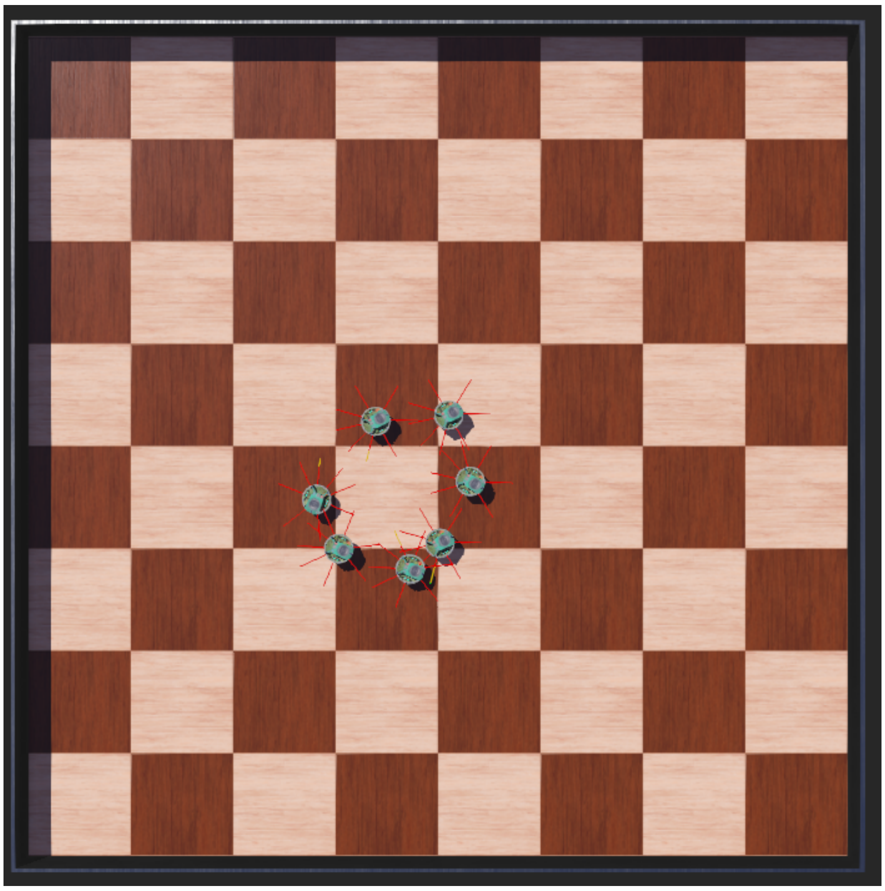


Figure 4.4: Rectangular Arena

4.5 Robot

A suitable robot platform was required to implement the β-Algorithm within the simulation. Webots' extensive library of pre-designed robots was explored. Ultimately, the 'epuck' **[3]** robot from Gctronic **[6]** was selected and instantiated using a PROTO node (see Figure 4.6). This robot model was added to the scene tree, where its properties, specifically its translation (position) and orientation, were customized to ensure proper initial placement within the arena. The 'epuck' robot offered several advantages for this project which were made possible due to its in-built devices (see Table 1 for a detailed view). Each of these devices offers unique features. These features include :

- **Distance Sensors:** The 'epuck' is equipped with eight IR (infrared proximity sensors) arranged around its body. This configuration provides a 360-degree view which is essential for obstacle detection and sensing the proximity of other robots which is a key component in the implementation of the β-Algorithm.

- **Motors:** Two differential drive wheels give the robot locomotion capabilities (the left and right wheel motor). This simple, yet versatile, movement system is well-aligns with the algorithm's focus on connectivity maintenance rather than complex navigation.

- **Additional Features:** LEDs for potential status indication, a camera with a resolution of 52x39 pixels for computer vision tasks, and other optional sensors such as the ground sensors can be added to the 'epuck'. While these were not essential for this specific project, the option for expansion could prove valuable for future research directions.

The 'epuck' was an ideal choice due to its simplicity, widespread use in swarm robotics, and alignment with the project's requirements. Its sensor configuration was well-suited for the beta algorithm, and its differential drive **[44]** system provided the necessary movement capabilities.

Webots provides a special class called "Supervisor" **[18]** that grants the user extended control over the simulation environment. While a Supervisor might not be necessary for all simulations, it can be a valuable tool for advanced tasks. For instance, the Supervisor can access precise robot positions **[50]** or other internal simulation data (which may not be directly measurable by the robot's sensors in a real-world setting). This might be useful for facilitating comparisons between simulated and ideal behavior. Additionally, the Supervisor can dynamically modify the environment during the simulation, such as adding obstacles or modifying terrain, allowing researchers to test the adaptability of their algorithms. The core behavior of the 'epuck' robot is governed by its controller program which for this project was written in C. This controller leverages Webots API functions to establish a communication loop between the robot and the simulator. The controller retrieves readings from the proximity sensors using relevant API functions **[14]** This sensor data is then processed following the β-Algorithm's logic, and implemented within the controller code. Finally, based on the processed data and the algorithm's decisions, the controller issues appropriate speed commands to the left and right motor wheels to maneuver the robot within the rectangular Arena.

Devices	Names in Webots
Motors	'left wheel motor' and 'right wheel motor'
Position sensors	'left wheel sensor' and 'right wheel sensor'
Proximity sensors	'ls0' to 'ls7'
LEDs	'led0' to 'led7' (e-puck ring), 'led8' (body) and 'led9' (front)
Camera	'camera'
Accelerometer	'accelerometer'
Ground sensors (extension)	'gs0', 'gs1' and 'gs2'

Table 4.1: Devices available for 'epuck' in the Webots simulator.

4.6 Sensors

The 'epuck' robot is equipped with a suite of sensors that enable it to perceive its simulated environment and gather information which is crucial for implementing algorithms like the β-Algorithm. While it possesses sensors such as light sensors, a microphone, an accelerometer, and a camera, its infrared proximity sensors play the most significant role in obstacle detection and robot-to-robot sensing within the swarm **[36]**. The 'epuck's' proximity sensing typically relies on a set of 8 proximity sensors that are distributed uniformly around its body (see Figure 4.5). This arrangement provides the robot with a nearly 360-degree field of view, enabling it to identify obstacles and other robots in its vicinity. Each infrared proximity sensor functions by emitting a beam of invisible infrared light from an IR LED. This emitted light interacts with objects encountered in the environment. If an object is present, a portion of the IR light reflects toward the sensor. The amount of reflected light is dependent on several factors, including how close the object is, and the object's material properties **[27]**. The proximity sensor contains a receiver component sensitive to infrared light. By measuring the intensity of the reflected infrared light, the sensor can provide an estimation of the distance to the object – stronger reflections generally indicate closer proximity **[23]**. Webots provide a realistic simulation of infrared sensor behavior. This means that the simulation can take into account the reflectivity of objects placed in the arena, along with variations in ambient lighting conditions, and even potential sensor noise **[35]**. This comprehensive simulation environment facilitates the robust development and testing of algorithms like the beta algorithm. The beta algorithm fundamentally relies on the ability to detect both obstacles in the environment and the proximity of other robots within the swarm **[39]**.

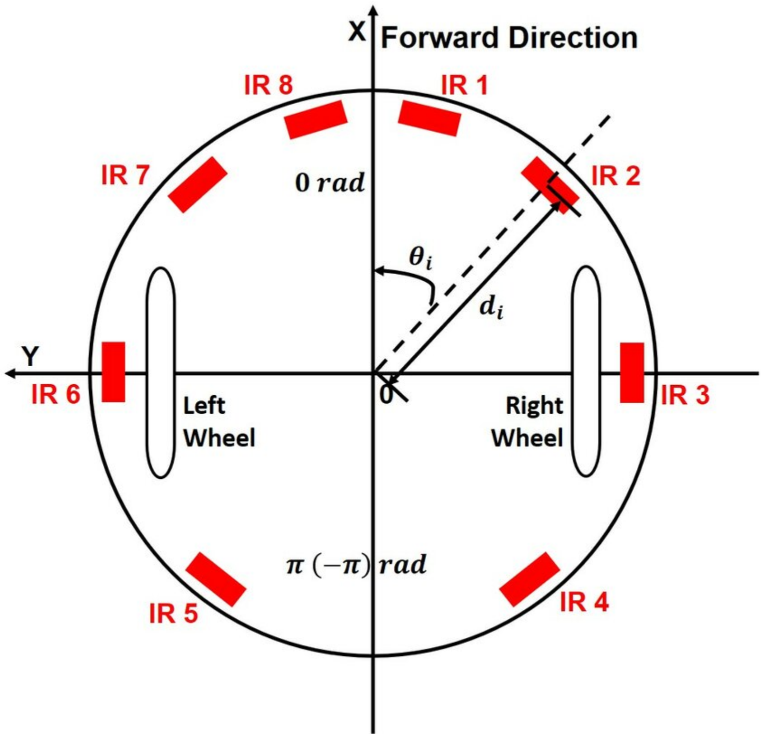


Figure 4.5: Distribution of IR sensors in epuck **[29]**

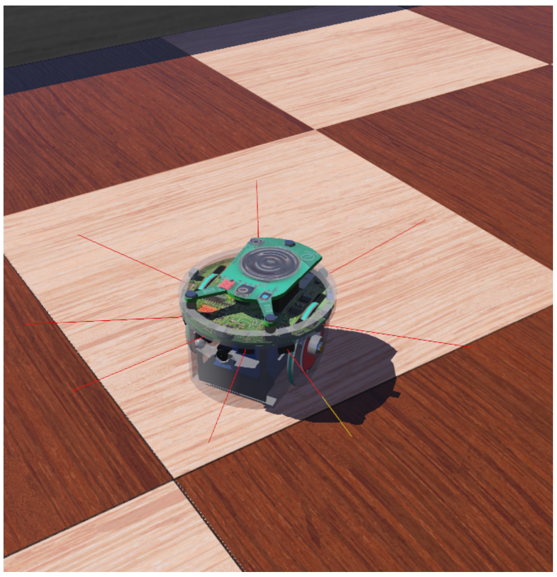


Figure 4.6: Distribution of IR sensors in epuck **[29]**

4.7 Controller

The controller is the heart of the robot since it controls all the necessary movements and calculations required by the robot. The controller code is written in C language **[33]** since it is a highly preferred language for robotics applications owing to its performance, direct hardware access, and portability. Its speed and efficiency are paramount in real-time systems where sensors and actuators demand rapid responses **[32]** . It can directly manipulate memory and interact with hardware components, This allows for precise control over the robot's motors, sensors, and communication interfaces **[10]**. Additionally, C's cross-platform compatibility ensures that code developed in simulations can be easily transitioned smoothly to embedded platforms used in physical robots**[41]**. These combined with C's simplicity and availability of well-established robotics libraries, make it a robust and dependable choice for the controller code language.

The controller code used in this project implements a hybrid approach combining elements of Braitenberg-style control **[22]** with an overarching connectivity maintenance behavior inspired by the β-Algorithm. The key features are discussed in detail below:

- **Sensor Integration & Pre-processing:** The controller primarily relies on the 'epuck's' array of infrared proximity sensors (defined by ps0-ps7 in the code). 'ps0' represents the first sensor, 'ps1' represents the second sensor, and so on. Values from these sensors are retrieved using the Webots API functions **[14]** (i.e. `wb_distance_sensor_get_value()`) and then the values are scaled to a range of values (0.0 - 1.0). This scaling normalizes the data, making calculations for obstacle detection pretty straightforward. The controller utilizes the emitter and receiver devices on the robot for communication within the swarm. The channel is explicitly set to '1' (`wb_emitter_set_channel()`) for all the robots of the swarm to reduce any loss in communication. A custom data structure, `Message` is defined to encapsulate data for transmission. Its components include:

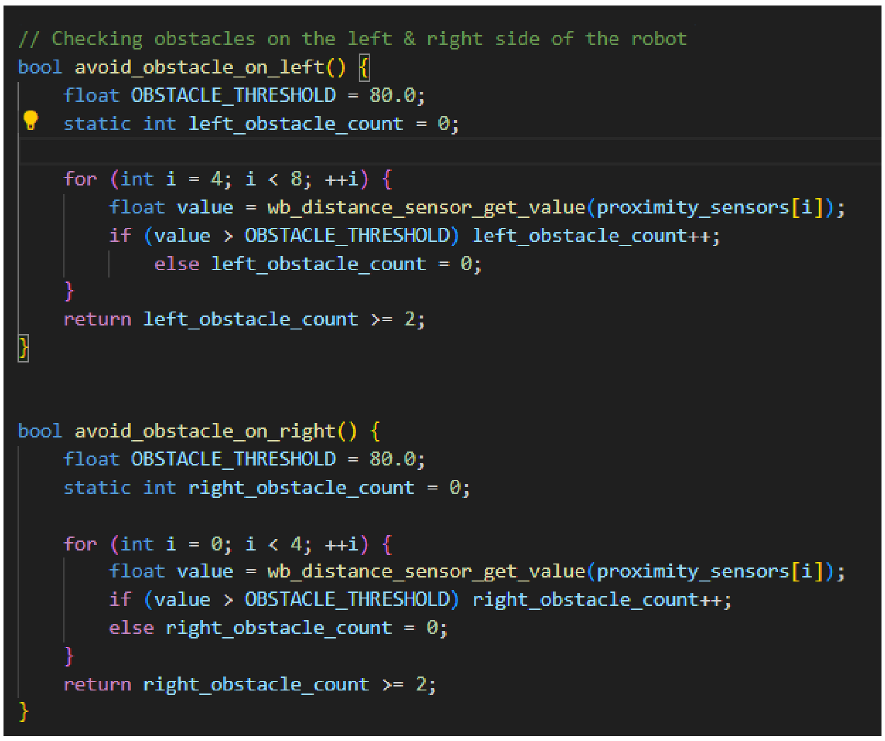
  1. **`type`:** A character flag to signify the message purpose (e.g., 'h' for heartbeat)
  2. **`data`:** A string field for additional information (e.g., "Corner detected")
  3. **`x_pos`:** The robot's current X-coordinate obtained via `get_x_position()`
  4. **`y_pos`:** The robot's current Y-coordinate obtained via `get_y_position()`

5. **`robot_name`:** The robot's unique identifier

- **Obstacle Avoidance (Braitenberg-Inspired) & Corner Detection:** As previously mentioned, the controller takes a Braitenberg-inspired **[22]** approach to navigation. It is a simple yet reliable approach that couples sensor readings into motor actions. This results in emergent behaviors **[34]** without any complex pre-planning. The primary obstacle avoidance (latest) is handled by the two functions `avoid_obstacle_on_left()` and `avoid_obstacle_on_right` (see Figure 4.6). These functions monitor the proximity sensors ('ps0-ps7') and if the threshold is exceeded on one side, indicating an obstacle, the robot executes a turn in the opposite direction by utilizing the functions `turn_left()` and `turn_right()`. The `calculate_turn_angle()` function dynamically adjusts the turn magnitude based on the sensor readings. It uses the boolean values from the functions `avoid_obstacle_on_left` & `avoid_obstacle_on_right` and evaluates whether to take a right or left turn using an XOR Condition (see Figure 4.7). The old version of obstacle detection made left and right turns using the function `passive_wait()`. This function halts all the processes for the robot while making a turn. This was discarded because it even halted communication with the other robots. Figure 4.9 & 4.10 shows how the new and old obstacle detection works. While detecting corner cases, only the proximity sensor '1', '2', '4', '5', '7', and '8' are used. This is because when detecting corners, the sensors in front of the 'epuck' (i.e. '1', '2', '7', and '8') are closer to the corners of the arena providing results before the sensors '3' and '6'. As a result, we discard these two sensors to improve efficiency.

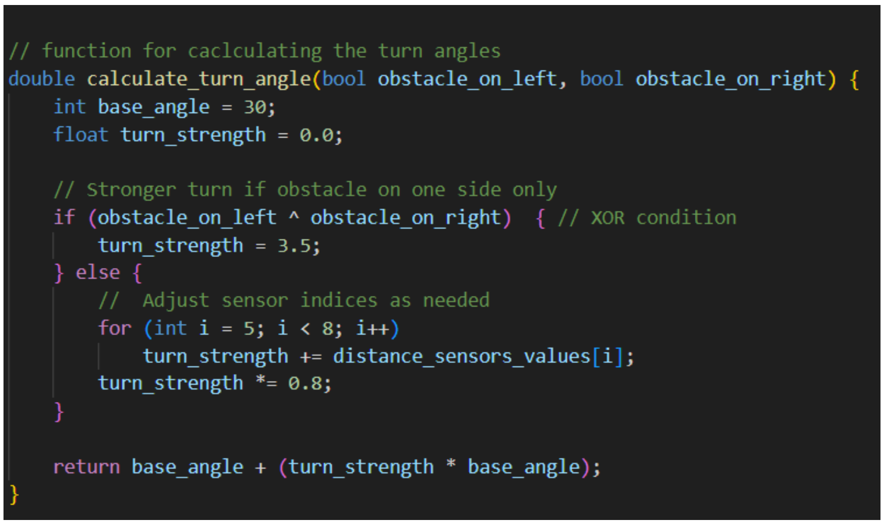
- **Connectivity Maintenance (β-Algorithm Implementation):** Above the reactive obstacle avoidance, is a communication and coordination behavior inspired by Nembrini's beta algorithm (See Figure 4.11). This aims to maintain connectivity within the swarm. Each robot periodically broadcasts "heartbeat" messages (type = 'h') at each time step using the send_message() function. These messages are then interpreted by the `handle_messages` function to perform the tasks essential for implementing the algorithm like calculating its distance from other robots within the swarm. The controller also maintains a NeighborList data structure to track robots within communication range. The `add_or_update_neighbor()`, `isInNeighborList()`,
and `isInReceivedNeighborList()` functions manage this list based on the

messages received. At regular intervals (controlled by the CADENCE parameter and `cad_count`), the robot checks its NeighborList. If a previously known robot is absent from it and the number of shared neighbors in heartbeat messages falls below the BETA threshold, it infers a potential loss of connection. Upon inferring a lost connection, the robot executes a 180-degree turn (`turn_right(180)`) followed by invoking the function (`run_braitenberg()`) to restart its movement. This maneuver aims to re-establish the connection with the lost robot(s).



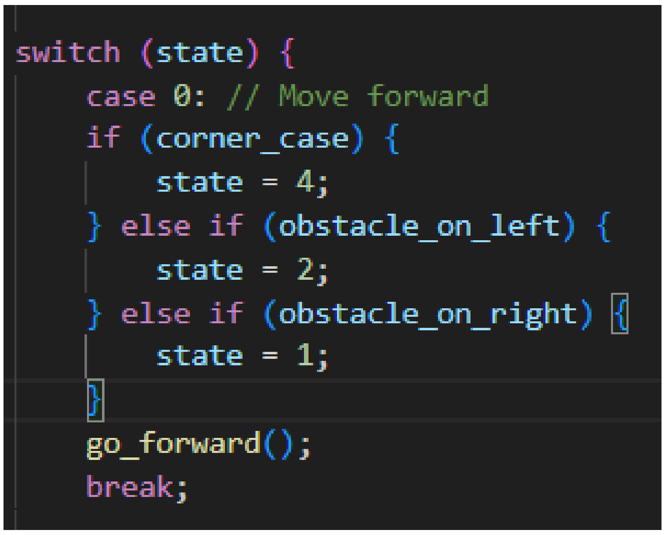
```c
// Checking obstacles on the left & right side of the robot
bool avoid_obstacle_on_left() {
    float OBSTACLE_THRESHOLD = 80.0;
    static int left_obstacle_count = 0;

    for (int i = 4; i < 8; ++i) {
        float value = wb_distance_sensor_get_value(proximity_sensors[i]);
        if (value > OBSTACLE_THRESHOLD) left_obstacle_count++;
            else left_obstacle_count = 0;
    }
    return left_obstacle_count >= 2;
}


bool avoid_obstacle_on_right() {
    float OBSTACLE_THRESHOLD = 80.0;
    static int right_obstacle_count = 0;

    for (int i = 0; i < 4; ++i) {
        float value = wb_distance_sensor_get_value(proximity_sensors[i]);
        if (value > OBSTACLE_THRESHOLD) right_obstacle_count++;
        else right_obstacle_count = 0;
    }
    return right_obstacle_count >= 2;
}
```

Figure 4.7: Code for functions which detect obstacles on left and right



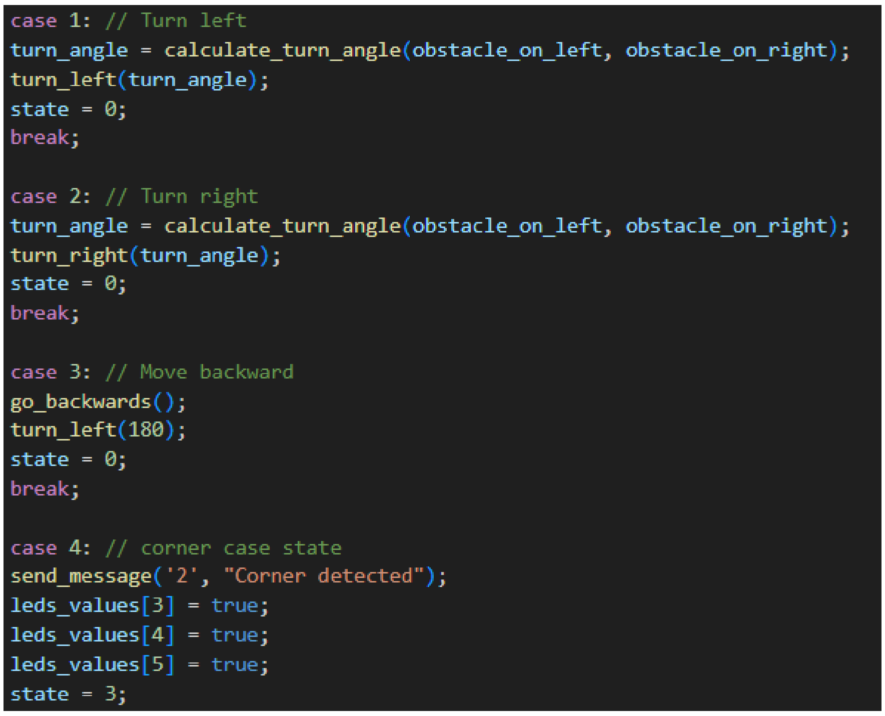
```
// function for caclculating the turn angles
double calculate_turn_angle(bool obstacle_on_left, bool obstacle_on_right) {
    int base_angle = 30;
    float turn_strength = 0.0;

    // Stronger turn if obstacle on one side only
    if (obstacle_on_left ^ obstacle_on_right)  { // XOR condition
        turn_strength = 3.5;
    } else {
        //  Adjust sensor indices as needed
        for (int i = 5; i < 8; i++)
            turn_strength += distance_sensors_values[i];
        turn_strength *= 0.8;
    }

    return base_angle + (turn_strength * base_angle);
}
```

Figure 4.8: Code for the function turn_angle



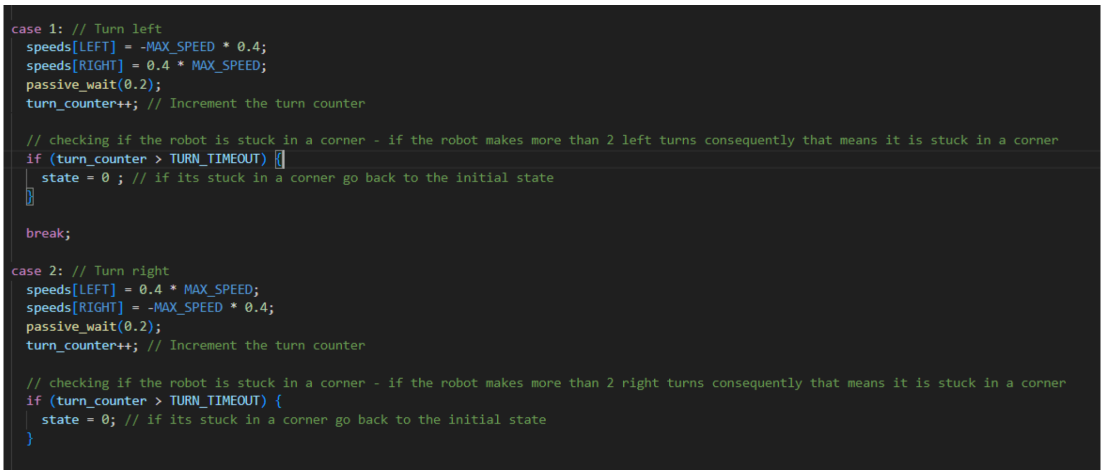
```
switch (state) {
    case 0: // Move forward
    if (corner_case) {
        state = 4;
    } else if (obstacle_on_left) {
        state = 2;
    } else if (obstacle_on_right) {
        state = 1;
    }
    go_forward();
    break;
```

Figure 4.9: Code showing the use of obstacle detection on left and right to make turning decisions



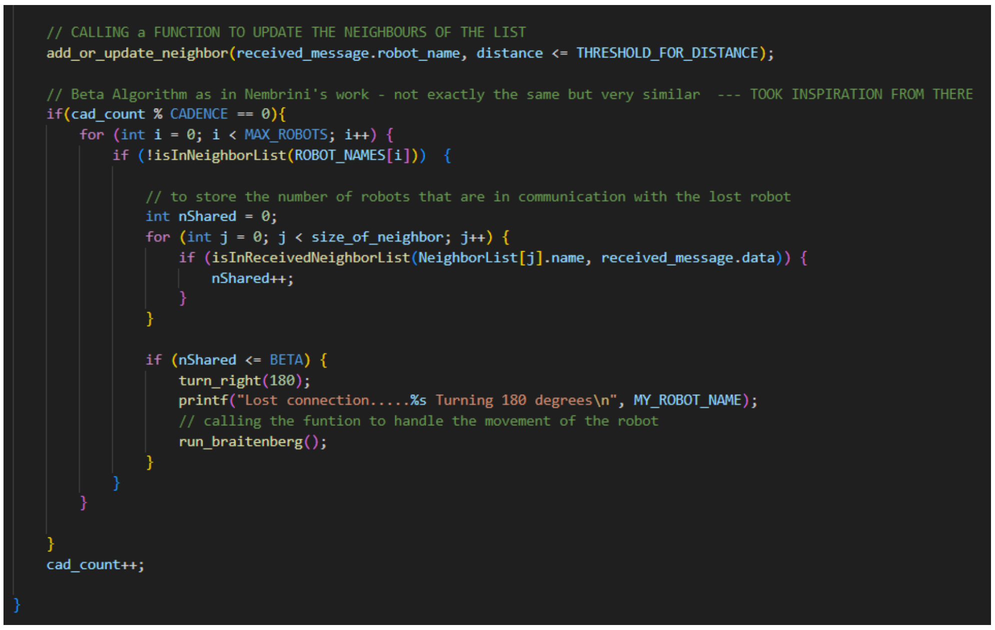
```
case 1: // Turn left
turn_angle = calculate_turn_angle(obstacle_on_left, obstacle_on_right);
turn_left(turn_angle);
state = 0;
break;

case 2: // Turn right
turn_angle = calculate_turn_angle(obstacle_on_left, obstacle_on_right);
turn_right(turn_angle);
state = 0;
break;

case 3: // Move backward
go_backwards();
turn_left(180);
state = 0;
break;

case 4: // corner case state
send_message('2', "Corner detected");
leds_values[3] = true;
leds_values[4] = true;
leds_values[5] = true;
state = 3;
```

Figure 4.10: Code for new obstacle detection



```
case 1: // Turn left
  speeds[LEFT] = -MAX_SPEED * 0.4;
  speeds[RIGHT] = 0.4 * MAX_SPEED;
  passive_wait(0.2);
  turn_counter++; // Increment the turn counter

  // checking if the robot is stuck in a corner - if the robot makes more than 2 left turns consequently that means it is stuck in a corner
  if (turn_counter > TURN_TIMEOUT) {
    state = 0 ; // if its stuck in a corner go back to the initial state
  }

  break;

case 2: // Turn right
  speeds[LEFT] = 0.4 * MAX_SPEED;
  speeds[RIGHT] = -MAX_SPEED * 0.4;
  passive_wait(0.2);
  turn_counter++; // Increment the turn counter

  // checking if the robot is stuck in a corner - if the robot makes more than 2 right turns consequently that means it is stuck in a corner
  if (turn_counter > TURN_TIMEOUT) {
    state = 0; // if its stuck in a corner go back to the initial state
  }
```

Figure 4.11: Code for old obstacle detection



```
    // CALLING a FUNCTION TO UPDATE THE NEIGHBOURS OF THE LIST
    add_or_update_neighbor(received_message.robot_name, distance <= THRESHOLD_FOR_DISTANCE);

    // Beta Algorithm as in Nembrini's work - not exactly the same but very similar  --- TOOK INSPIRATION FROM THERE
    if(cad_count % CADENCE == 0){
        for (int i = 0; i < MAX_ROBOTS; i++) {
            if (!isInNeighborList(ROBOT_NAMES[i]))  {

                // to store the number of robots that are in communication with the lost robot
                int nShared = 0;
                for (int j = 0; j < size_of_neighbor; j++) {
                    if (isInReceivedNeighborList(NeighborList[j].name, received_message.data)) {
                        nShared++;
                    }
                }

                if (nShared <= BETA) {
                    turn_right(180);
                    printf("Lost connection.....%s Turning 180 degrees\n", MY_ROBOT_NAME);
                    // calling the funtion to handle the movement of the robot
                    run_braitenberg();
                }
            }
        }

    }
    cad_count++;
}
```

Figure 4.12: Code for the implementation of the Beta algorithm

Chapter 5

Experiments

Building on the implementation details from the previous chapter, this section focuses on evaluating how well the foraging robots and the simulation itself perform. We'll run an experiment with a slight change to the robots' decision-making process (modeled by a probabilistic finite state machine) and analyze the results. Additionally, we'll experiment with different settings to assess the swarm's behavior from various angles. Finally, to gauge the robot swarm's effectiveness, we'll compare our results to those reported in reference [23].

All the experiments were carried on my laptop Dell G15 5510, 16GB RAM, 512 SSD, Intel I5, and Nvidia GTX 1650.

5.1 Simulation Evaluation

My initial goal was to directly replicate the experimental framework used by Nembrini and Winfield in their seminal work on the beta algorithm. Their focus on metrics like edge connectivity provided a compelling model for evaluating swarm coherence. However, limitations within the Webots simulation environment prevented me from calculating these metrics as they did. Rather than abandoning the project, I demonstrated adaptability by devising my experimental approach. Maintaining a focus on swarm coherence, I designed experiments to investigate the impact of varying the beta parameter and potentially the size of the swarm. While my experimental setup differed from Nembrini's, it allowed me to gain valuable insights into how these factors influence the beta algorithm's behavior. Despite these necessary adjustments, understanding Nembrini's original methodology remains important. In subsequent sections, I'll discuss their experiments in detail, followed by an analysis of my own findings and a comparison of how simulation limitations shaped the project's trajectory. Even after the challenges faced due to the limitations of webots, I was successful in implementing the β-Algorithm. The code for the controller is not perfect but is successful in displaying swarm coherence. The robots are not moving perfectly in a random manner but rather moving in concentric circles. This is due to the fact a lot of adjustments were required to deal with Webots' limitations and most of the time was spent overcoming these limitations. Basic functionalities such as obstacle avoidance and corner detection were working flawlessly except for robot-to-robot avoidance. As mentioned previously (state the section) the latest obstacle avoidance was unable to detect robots sometimes. Also, I was not able to increase the size of the swarm to more than 7 because then my laptop was unable to handle the load from the software, resulting in crashes. As a result, experiments are crafted keeping these limitations in mind. Since the neighbor list for each robot was made using a custom data structure, it was not perfect. After the simulation had run for approximately 2 minutes (state the parameters), the neighbor list started to display wrong values in the list. During these 2 minutes, the list was accurate.

5.2 Nembrini's Experiment Results

The experiments made by Nembrini demonstrated that the β-algorithm offers significant improvements over the α-algorithm in maintaining the connectivity of robotic swarms, particularly as they increase in size. Importantly, the β threshold allows for direct control of swarm connectivity and the area it covers. Increasing the threshold results in a more compact swarm while decreasing it leads to rapid expansion. This relationship holds even when the swarm size increases. Further, the β-algorithm shows remarkable robustness, as its performance degrades minimally in the presence of noise affecting communication, sensors, and actuators.

5.3 Affect on coherence with the change in β threshold

I experimented with different values of β to check what effect it had on swarm coherence specifically 'swarm spread'. The factor that I used to test the algorithm was the difference between each pair of robots in the simulation. If the distance between each of the robot pairs is almost the same, that would mean that the spread is equal and hence the robots are indeed within the swarm. I was unsuccessful in implementing the algorithm, I would have expected a vast difference in these values. I conducted this test with 4 different β values, namely '1', '2', '3', and '5'.

5.3.1 When β = 1

When β was set to one, the swarm maintained coherence for the first 50 seconds, and then some of the robots started to move away from the central swarm. 2 minutes into the simulation, the robot controller crashed along with the failure of the neighbor list and all the robots stopped moving. This might be because of hardware limitations or any inefficiency in the controller code. However for the first 1 minute, the robots maintained coherence, but not as perfectly as you would expect from a polished implementation of the β algorithm like Nembrini**[37]**. In Figure 5.1, you can see that most robots tried to stay within the swarm and their distance was slightly uniform. Most of the robot pairs maintained a distance of 30 cm, which is the communication threshold. However, you can see that some of the robots have values larger than the distance threshold indicating that these robots went out of the uniform swarm spread. The difference in peak shows that when the beta is 1, the range of the swarm is greater. This was also observed by Nembrini & Alan Winfield in their work **[37]**. The parameters used for this experiment are mentioned in the Table 5.1.

Parameter	Value
Arena Width	2m
Arena Height	2m
Number of robots	8
Communication range Range	30.0 cm
Cadence	4
Time in Simulation	2min 06 seconds
β	1

Table 5.1: Parameters used when β was 1

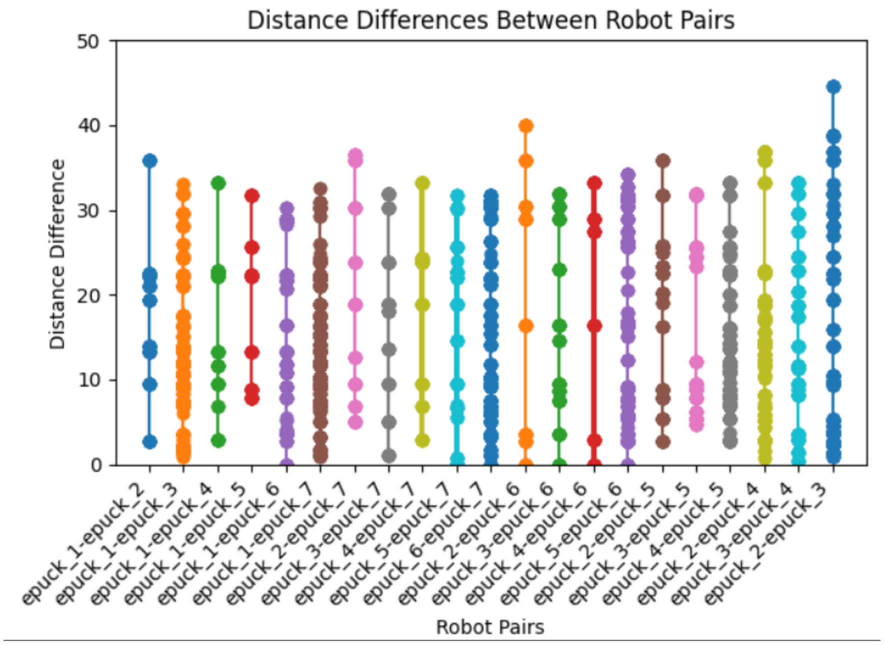


Figure 5.1: Gragh showing the distance between all the robot pairs, when β is 1

5.3.2 When β = 2

When β was set to 2, the performance of swarm coherence was much better. The swarm showcased coherence and uniform spread for a longer period (about 1-minute & 20 seconds) before the controller crashed. In Figure 5.2, you can see that most robots tried to stay within the swarm and their distance was quite uniform (better than when β was 1). Most of the pairs maintain a distance of 30-35 cm resulting in a uniform spread. However, even with these values some of the pairs have values larger than the distance threshold indicating that these robots went out of the uniform swarm spread, but these robots were very few. The parameters used for this experiment are mentioned in Table 5.2.

Parameter	Value
Arena Width	2m
Arena Height	2m
Number of robots	8
Communication range Range	30.0 cm
Cadence	4
Time in Simulation	2min 30 seconds
β	2

Table 5.2: Parameters used when β was 2

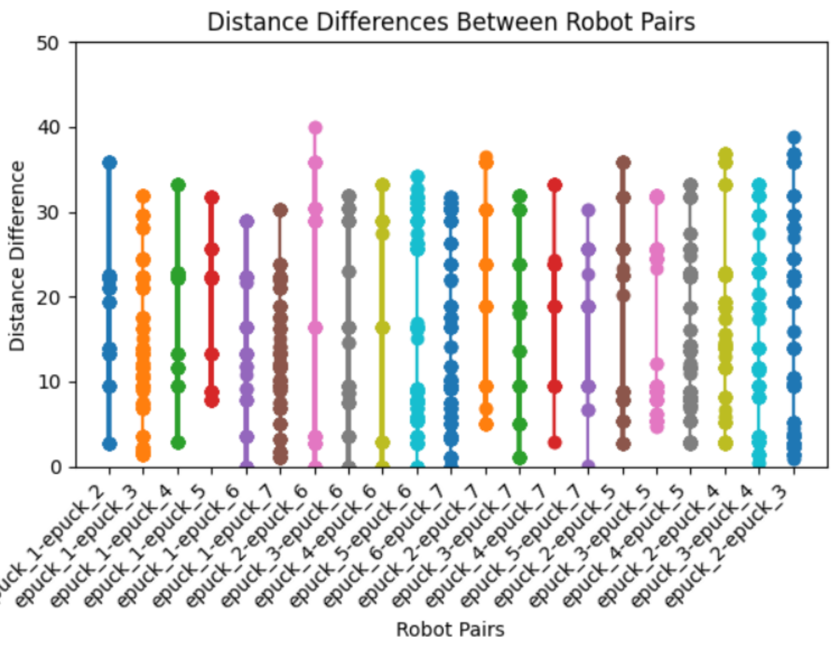


Figure 5.2: Gragh showing the distance between all the robot pairs, when β is 2

5.3.3 When β = 3

When β was set to 3, the performance of swarm coherence was ideal, but not the best. The swarm showcased coherence and uniform spread for almost 1 minute before the controller crashed. In Figure 5.3, you can see that most robots tried to stay within the swarm and their distance was quite uniform. Most of the pairs maintain a distance of 30-35 cm resulting in a uniform spread. However, there were still peaks that indicated that robots were outside the coherence spread. The parameters used for this experiment are mentioned in Table 5.3.

Parameter	Value
Arena Width	2m
Arena Height	2m
Number of robots	8
Communication range Range	30.0 cm
Cadence	4
Time in Simulation	2min 30 seconds
β	3

Table 5.3: Parameters used when β was 3

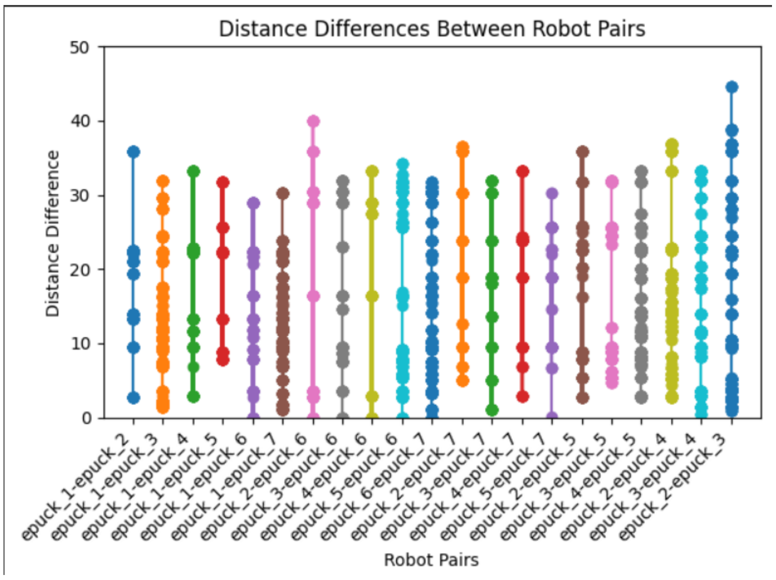


Figure 5.3: Gragh showing the distance between all the robot pairs, when β is 3

5.3.4 When β = 5

When β was set to 5, the performance of swarm coherence was one of the worst. The swarm showcased coherence and uniform spread for a short while (approx. 1 minute) before the controller crashed. In Figure 5.4, you can see that most robots tried to stay within the swarm and their distance was quite uniform but had many peaks that indicated that robots were outside the coherence spread. The parameters used for this experiment are mentioned in Table 5.4.

Parameter	Value
Arena Width	2m
Arena Height	2m
Number of robots	8
Communication range Range	30.0 cm
Cadence	4
Time in Simulation	2min 30 seconds
β	5

Table 5.4: Parameters used when β was 5

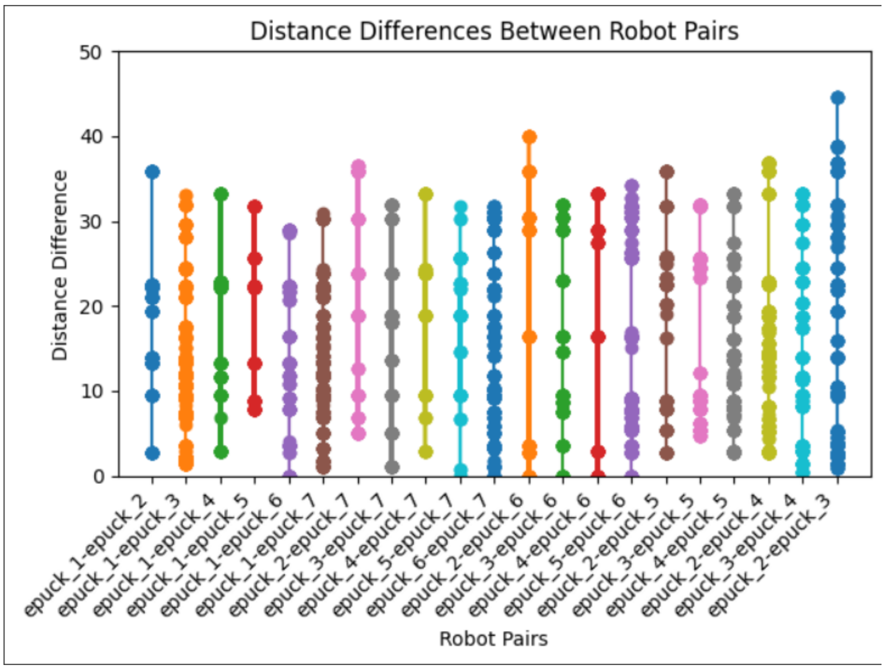


Figure 5.4: Gragh showing the distance between all the robot pairs when β is 5

5.4 Experiment & Evaluation Conclusion

The experiments investigating the effect of the β threshold on swarm coherence demonstrated a clear relationship. While all tested β values led to controller crashes (potentially indicating hardware or code limitations), there were distinct performance differences before the crashes. When β was 1, coherence was maintained briefly, with the swarm spread becoming uneven as some robots moved away. When β was 2, the simulation had the best result. Coherence and uniform swarm spread were improved, lasting for a longer duration. When β was 3, coherence remained strong, but there were still instances of robots breaking away from the formation. When β was 5, coherence was weaker, with a greater number of deviations from a uniform swarm spread. The best value of β was 2, which showcased the best results in maintaining swarm coherence.

Despite the limitations imposed by the Webots simulation environment, my experiments revealed a clear influence of the β threshold on swarm coherence. While my approach deviated from Nembrini's due to technical constraints, the findings provide valuable insights into the β algorithm's behavior. The results support the concept that increasing the β value, within a certain range, improves swarm coherence and maintains a more uniform spread. However, an excessively high β value appears to degrade performance. The consistent controller crashes highlight a significant limitation of the Webots platform. These crashes restricted the duration of the experiments and potentially masked the full effect of different β values. Issues with robot-to-robot detection and neighbor list calculations further influenced the results. To overcome these limitations and achieve more robust comparisons to Nembrini's work, exploring alternative simulation environments with better support for swarm robotics, potentially transitioning to physical robot experiments, and optimizing the controller code for improved stability would be beneficial. Overall, while the simulation constraints limited a direct replication of Nembrini's experiments, my work establishes a foundation for understanding the β algorithm's behavior. The findings motivate further exploration using more robust tools to unlock the algorithm's potential for swarm robotics.

Chapter 6

Conclusion

In this chapter, we will discuss what I achieved by doing the project and also talk about future extensions which could improve swarm coherence.

6.1 Achievements

This project was a very challenging yet incredibly rewarding journey. This project introduced me to the fascinating world of swarm robotics. I began the project with limited knowledge in this field but after intensive research and meticulous study of academic papers focussing on swarm robotics, I was able to gain invaluable insights into the concepts and algorithms used to attain swarm coherence. This dedication laid a robust foundation that enabled me to grasp the complexities of swarm dynamics and the crucial role connectivity plays in collective behavior. After attaining the required conceptual knowledge, I transitioned to the actual implementation of the β-Algorithm. This presented me with a unique set of challenges that required me to translate theoretical concepts into functional code. Selecting C as my programming language, not only helped me develop the required logic for building the robot controller but also deepened my understanding of C language's strengths and intricacies. Iterative development, testing, and debugging were essential in analyzing the robot's behavior and pinpointing unexpected interactions. These steps were crucial in refining the algorithm's implementation for achieving the desired swarm coherence. Here's a breakdown of some key milestones I've achieved through this project:

- **Understanding the Nuances of Swarm Coherence:** After the thorough in-depth literature review and algorithm analysis, I gained a stronghold of swarm coherence's significance. Early swarm algorithms often struggled to maintain connectivity, especially in dynamic environments. The β-Algorithm's simple neighbor-tracking and reaction mechanisms presented an elegant solution to the problem. Witnessing how the promotion of local connectivity translates into large-scale swarm cohesion has been incredibly insightful.

- **Developing the Swarm Simulation:** Exploring a variety of simulation software was no easy task. Finding a Simulation software that aligned with the project and supported by my machine was quite a journey. Then building a simulation environment demanded both creativity and technical skills. At first, learning how to model the virtual world and how to use the nodes in Webots was a stressful task. However, on using the software repetitively, I started getting used to the software and developed a slight interest in it. Incorporating basic physics principles for robot movement, and designing the logic governing robot-robot and robot-environment interactions was the part of the software I enjoyed the most. Finally designing the robot's sensory range for facilitating smooth turns and solving connectivity threats among the swarm reinforced my grasp on both the β-Algorithm.

- **Appreciating the Art of Debugging:** Troubleshooting and debugging became an integral part of the development process. Due to the nature of the experiment, a minor code adjustment could lead to major differences in the overall swarm pattern. For example, just by changing the distance threshold (i.e. the threshold defining the communication range of each robot), we see major changes in the spread of the swarm. Analyzing unexpected results and then tracing them back to their root cause, required attention to detail and a good understanding of the logic with which the robots interact within the swarm.

- **Translating Theory to Practice:** The project highlighted vast differences between theoretical algorithms and their actual implementation in the real world. I learned the crucial roles of tuning and playing with parameters to get the best results, and issues that emerge when handling complex multi-robot systems in a simulated environment.

- **Designing and Conducting Experiments:** One of the most satisfying aspects of the project was the design and execution of my experiments. By taking inspiration from the research papers during the background review, I tried implementing the same experiments to test the algorithm's behavior under varying parameters like swarm size, communication range, and the β value. Rigorous experiments were done which demanded careful and statistical analysis to recognize the limitations of my simulated environment.

- **Growing as a Researcher:** The part of the project I appreciate is the fact that this project gave me a valuable taste of how professionals craft their research papers and construct their tests. From reading other research papers to finally communicating my methodology and findings, this experience introduced me to the workflow followed by researchers in the field of robotics. This understanding will undoubtedly prove beneficial as I continue exploring robotics and related fields.

6.1.1 Broader Impact and Skill Development

Apart from the outcomes due to the actual project content, this journey has impacted me positively in my overall development. This project enhanced my proficiency in the C language as it forced me to move beyond simple code writing and gain a deep understanding of memory management, data structures, and performance optimizations. While these were challenging, they significantly enhanced my fluency in the language. Another skill I gained from the project is to write clearly and concisely about my research, methodology, findings, and results. These skills will be an asset in future research.

6.2 Challenges and Limitations

Completing this project was no easy task and it came with a set of setbacks and challenges. These challenges significantly shaped the overall project, demanding flexibility and creative solutions. Here's a breakdown of the key limitations I encountered and their impact:

6.2.1 Roadblocks faced while deciding the Simulation Software

After doing the Literature review, the goal was to explore an established robotics simulator to expedite the development process for the swarm coherence behavior. The initial step involved researching several promising options, including Gazebo **[4]**, Net-Logo **[8]**, NVIDIA Omniverse **[9]**, and Webots **[13]**. Each offered unique features, and I carefully considered factors like ease of prototyping, swarm customization capabilities, and virtual environment design flexibility. Unfortunately, early roadblocks arose. Initially drawn to Gazebo, I invested time learning the platform, consulting its manual and documentation. However, a major compatibility issue emerged: My Windows-based machine did not support the software as it was designed primarily for macOS or Linux operating systems. Determined to proceed, I installed a virtual machine and set up a Linux environment. However, even after installation, Gazebo proved unstable even on the virtual machine. The demanding nature of the software led to frequent glitches, crashes, and even worse, frustrating losses of the project's progress. The limitations of running it within a virtualized environment made it an unreliable tool for this project. After extensive time investment, I reluctantly had to abandon this simulator and explore alternative solutions. A thorough research on available simulation software led me to consider Webots. Learning from past mistakes, after carefully reviewing its manual and hardware requirements, I was sure that Webots would be an ideal choice.

6.2.2 Problems with Webots

As I transitioned to Webots, I was hoping to overcome the limitations I encountered while using Gazebo, However, to my surprise. I faced a new set of challenges. These hurdles, though smaller in scale, had significant implications for the project's advancement. Initially, Webots presented its learning curve. While the software wasn't overly complex, understanding its structure, programming paradigms, and the nuances of robot modeling required time and dedicated learning. This investment was necessary to gain the proficiency needed to implement the beta algorithm effectively. Once I had mastered the basics of Webots, a major obstacle emerged during the development of the robot's controller. Although my robots successfully avoided the arena's walls, they frequently collided with each other. This necessitated a focused effort to design and implement a robust collision avoidance solution. Multiple iterations of the controller code were required, involving careful adjustments to sensor usage and the logic governing robot turning behaviors. Through this process, I gained a deeper appreciation for the complexities of collision avoidance within a swarm, even when individual robot behaviors appear deceptively simple. After resolving the robot-robot collision issue, another challenge emerged. The robots would often become stuck in the corners of the rectangular arena. This issue is frequently encountered in robotics **[45]**. To mitigate this, researchers often utilize hexagonal arenas, as their less acute angles reduce the complexity of corner navigation. However, since creating a hexagonal arena was not directly feasible in Webots, I had to develop a specialized corner case handling. My initial approach relied on tracking the number of turns the robot executed. If this number exceeded a threshold, then it indicates that the robot was likely trapped in a corner. The controller would then command the robot to reverse and turn left, attempting to free itself from the corner. While this first approach functioned, I refined it for better optimization. The new method focused on directly sensing the corner situation. When the specific sensors (mentioned in the above sections) for corner detection, were all true, indicating a corner encounter, the robot executed a simple 180-degree turn. This proved to be a more efficient and elegant solution to the corner navigation problem.

This experience showcased the iterative nature of robotics development. Even after achieving reliable general movement and collision avoidance, the interplay between robot behavior and the arena's shape led to an unexpected issue. Identifying the problem, devising solutions, and comparing their efficiency highlighted the kind of creative problem-solving frequently required in this field. The beta algorithm's core mechanism hinges on the fact that each robot has access to information about its neighbors within the swarm. Ideally, a normal coding environment would provide a ”global neighbor list” that any robot could query to assess the swarm's local connectivity. However, this type of shared data structure was not supported by Webots. Each robot's controller could track its neighbors, but there was no built-in way to consolidate this information into a centralized, swarm-wide understanding.

To address this limitation, I had to develop my solution. A custom data structure was constructed to store the neighbor list information, ensuring accessibility across all robot controllers. It wasn't merely enough to store robot IDs, further checking was required such as checking if a robot was already on the list, if not then a check was carried out to see if the robot was within its range. Moreover, I wrote helper functions to manage this structure. These functions handled adding & removing robots from the list, updating information as the swarm moved, and executing the queries required by the beta algorithm's logic. Although this workaround proved time-consuming and added complexity to the project, it was essential for accurately implementing the β-Algorithm. Unexpectedly, this challenge became one of the most enjoyable parts of the project. It forced me to think creatively about problem-solving, deepening my understanding of data structures and how to manipulate them as per the situation.

While Webots proved to be a useful tool for implementing the beta algorithm, it presented certain challenges when it came to the experimental phase of the project. Conducting reliable experiments, particularly those involving large datasets, was not as straightforward as initially desired. The primary limitation stemmed from the lack of built-in capabilities for data collection within the webots simulator. To track relevant metrics (such as swarm cohesion over time,), I had to implement custom logging mechanisms within the simulator. This involved writing code to capture the desired data during the simulation runs, storing it in an appropriate format, and potentially developing additional scripts or tools to extract and analyze the results. The lack of integrated support within Webots meant a greater time investment was needed in designing effective data collection workflows and potentially integrating external analysis tools.

While Webots enabled me to implement and study the beta algorithm, it presented obstacles in fully replicating the experimental scale described in Nembrini's original work. His research often involved swarm sizes ranging from a few robots to upwards of 60. Unfortunately, Webots struggled to handle simulations of this magnitude. The sheer number of robots, each with its own controller, sensors, and communication logic, put a significant strain on computational resources. This likely led to slower simulation speeds, making lengthy experiments impractical, or potentially even simulation crashes. Furthermore, even if I could run simulations with larger swarms, Webots' console output would become overwhelmed with data. There were limited options for exporting data logs in manageable formats or integrating visualization tools to aid in interpreting the massive amount of information produced by a large-scale swarm. This experience highlighted how unexpected challenges can arise at any stage of a robotics project. It also reinforced the importance of iterative development, where persistent testing and refinement are crucial for achieving the desired swarm behavior.

6.3 Future Work: Expanding the Capabilities of the Swarm

The implemented algorithms successfully established a foundation for a cooperative foraging swarm. However, there's immense potential to enhance the swarm's efficiency, robustness, and real-world applicability. This section talks about several exciting avenues for future exploration, that are built upon the groundwork laid in this project.

6.3.1 Refining the β-Algorithm for Enhanced Swarm Cohesion

While the current implementation of the β-Algorithm maintains basic connectivity within the swarm, further refinements can significantly improve its effectiveness. One crucial area for development is tailoring the algorithm to prioritize swarm cohesion. This can be achieved by incorporating metrics that not only measure individual robot proximity but also consider the overall compactness and directionality of the swarm.

- **Exploring Integration with Situated Communication Principles:** An interesting avenue to explore is the integration of principles from Støy's work on situated communication **[48]** with the β algorithm. His research focuses on making robots adapt to changing environments. He suggests that robots could adjust how important it is for them to stay close together, versus staying connected, depending on how many other robots are nearby. Think of it like a group of friends walking through a crowd. Sometimes you want to stick close together, but other times it's better to spread out to move more easily. This kind of flexibility could help robot swarms work better in different situations, making them more efficient overall.

- **Shape-Based Considerations:** The shape of a robot swarm affects how well it can search an area. If the robots are too close to each other, they get in each other's way and can't explore the area properly. But if they spread out too far, they can lose contact with each other and can't work together as a team. We should look at ways to make them form a longer line or a fan shape while searching. This way, more robots can cover the task area, and still stay connected to their teammates **[49]**.

6.4 Conclusion

This project delved into the complexities of swarm robotics, and I'm honestly a little surprised it worked as well as it did. From diving into the theory behind the β-Algorithm to the nuts and bolts of Webots, there were some frustrations (like those corner collision detection). Despite the hurdles, seeing the robots exhibit basic swarm behavior was incredibly rewarding, underscoring the potential for refinement.

Looking ahead, I'm keen to make the swarm more cohesive, focusing on how the β-Algorithm could encourage a tighter, more unified group dynamic. Integrating situated communication principles, as Støy explores, could be a game-changer, enabling the swarm to adapt its strategies based on the environment. Experimenting with how the swarm's shape impacts task efficiency also seems promising. Adding more sophisticated obstacle avoidance would also significantly increase the swarm's real-world applicability. And of course, the ultimate test will be transitioning to physical robots – that will certainly reveal a whole new set of challenges and exciting possibilities.

Overall, this project has been both challenging and inspiring. The problem-solving, iterative development, and moments when I was able to finally solve the persisting issues in the controller was a roller coaster of a journey. This project has fueled my passion for swarm robotics. I'm excited to continue learning and pushing the boundaries of what these robot swarms can achieve.

Bibliography

[1] Atlas by boston dynamics. `https://bostondynamics.com/atlas/`. (Accessed: April 19, 2024).

[2] Boston dynamics. `https://bostondynamics.com/`. (Accessed: April 19, 2024).

[3] Epuck by gctronic. `https://e-puck.gctronic.com/`. (Accessed: April 26, 2024).

[4] Gazebo. `https://gazebosim.org/home`. (Accessed: April 19, 2024).

[5] Gazebo not available on windows. `https://gazebosim.org/docs/all/getstarted`. (Accessed: April 19, 2024).

[6] Gctronic. `https://www.gctronic.com/`. (Accessed: April 26, 2024).

[7] Nao. `https://www.aldebaran.com/en/nao`. (Accessed: April 19, 2024).

[8] Netlogo. `https://ccl.northwestern.edu/netlogo/`. (Accessed: April 19, 2024).

[9] Nvidia omniverse. `https://www.linux.org/`. (Accessed: April 19, 2024).

[10] Spong + robot modeling and contro. `https://www.researchgate.net/profile/Mohamed_Mourad_Lafifi/post/How_to_avoid_singular_configurations/attachment/59d6361b79197b807799389a/AS%3A386996594855942%401469278586939/download/Spong+-+Robot+modeling+and+Control.pdf`. (Accessed: April 26, 2024).

[11] Spot by boston dynamics. `https://bostondynamics.com/products/spot/`. (Accessed: April 19, 2024).

[12] Vehicles in webots. `https://www.cyberbotics.com/doc/automobile/car?version=cyberbotics:R2019a`. (Accessed: April 19, 2024).

[13] Webots. `https://cyberbotics.com/`. (Accessed: April 19, 2024).

[14] Webots api documentation. `https://cyberbotics.com/doc/reference/nodes-and-api-functions`. (Accessed: April 26, 2024).

[15] Webots blog posts. `https://cyberbotics.com/doc/blog/index`. (Accessed: April 19, 2024).

[16] Webots proto node. `https://cyberbotics.com/doc/automobile/proto-nodes`. (Accessed: April 26, 2024).

[17] Webots solid node. `https://cyberbotics.com/doc/reference/solid`. (Accessed: April 26, 2024).

[18] Webots supervisor class. `https://cyberbotics.com/doc/reference/supervisor`. (Accessed: April 26, 2024).

[19] R. D. Arnold, H. Yamaguchi, and T. Tanaka. Search and rescue with autonomous flying robots through behavior-based cooperative intelligence. *Journal of International Humanitarian Action*, 3(1):1–18, 2018.

[20] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, et al. Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study. *Proceedings of the national academy of sciences*, 105(4):1232–1237, 2008.

[21] L. Bayındır. A review of swarm robotics tasks. *Neurocomputing*, 172:292–321, 2016.

[22] V. Braitenberg. Vehicles, 1984.

[23] V. Braitenberg. *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.

[24] M. Brambilla, E. Ferrante, M. Birattaria, and M. Dorigo. Swarm robotics: a review from the swarm engineering perspective. pages 1–2, 2013. Accessed: April 14, 2024.

[25] S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraula, and E. Bonabeau. Self-organization in biological systems. In *Self-Organization in Biological Systems*. Princeton university press, 2020.

[26] E. Drumwright, J. Hsu, N. Koenig, and D. Shell. Extending open dynamics engine for robotics simulation. In *Simulation, Modeling, and Programming for Autonomous Robots: Second International Conference, SIMPAR 2010, Darmstadt, Germany, November 15-18, 2010. Proceedings 2*, pages 41–45. Springer, 2010.

[27] H. Everett. *Sensors for mobile robots*. AK Peters/CRC Press, 1995.

[28] N. R. Franks and A. B. Sendova-Franks. Brood sorting by ants: distributing the workload over the work-surface. *Behavioral Ecology and Sociobiology*, 30:109–123, 1992.

[29] R. Groß, Y. Gu, W. Li, and M. Gauci. Generalizing gans: A turing perspective. 11 2017.

[30] B. Hölldobler and E. O. Wilson. *The ants*. Harvard University Press, 1990.

[31] F. Iida, P. Maiolino, A. Abdulali, and M. Wang. *Towards Autonomous Robotic Systems: 24th Annual Conference, TAROS 2023, Cambridge, UK, September 13–15, 2023, Proceedings*, volume 14136. Springer Nature, 2023.

[32] Y. Jin, J. Liu, Z. Xu, S. Yuan, P. Li, and J. Wang. Development status and trend of agricultural robot technology. *International Journal of Agricultural and Biological Engineering*, 14(4):1–19, 2021.

[33] B. W. Kernighan and D. M. Ritchie. The c programming language. 2002.

[34] Z. Li, C. H. Sim, and M. Y. H. Low. A survey of emergent behavior and its impacts in agent-based systems. In *2006 4th IEEE international conference on industrial informatics*, pages 1295–1300. IEEE, 2006.

[35] O. Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[36] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on*

autonomous robot systems and competitions, volume 1, pages 59–65. IPCB: Instituto Politécnico de Castelo Branco, 2009.

[37] J. Nembrini and A. F. W. F. Emergent swarm morphology control of wireless networked mobile robots. 2012. Accessed: April 14, 2024.

[38] K. M. Passino. Biomimicry of bacterial foraging for distributed optimization and control. *IEEE control systems magazine*, 22(3):52–67, 2002.

[39] J. Pugh and A. Martinoli. Multi-robot learning with particle swarm optimization. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 441–448, 2006.

[40] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[41] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[42] B. L. G. D. S. (Retd). Swarm drones - new frontier of warfare. 2021. Accessed: April 14, 2024.

[43] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

[44] F. A. Salem. Dynamic and kinematic models and control for differential drive mobile robots. *International Journal of Current Engineering and Technology*, 3(2):253–263, 2013.

[45] T. Salter, K. Dautenhahn, and R. Bockhorst. Robots moving out of the laboratory-detecting interaction levels and human contact in noisy school environments. In *RO-MAN 2004. 13th IEEE International Workshop on Robot and Human Interactive Communication (IEEE Catalog No. 04TH8759)*, pages 563–568. IEEE, 2004.

[46] A. R. Scott Dresser, VP. Amazon announces 2 new ways it's using robots to assist employees and deliver for customers. pages 1–2, 2023. Accessed: April 14, 2024.

[47] T. D. Seeley. *The wisdom of the hive: the social physiology of honey bee colonies*. Harvard University Press, 2009.

[48] K. Støy et al. Using situated communication in distributed autonomous mobile robotics. In *SCAI*, volume 1, pages 44–52, 2001.

[49] A. E. Turgut, H. Çelikkanat, F. Gökçe, and E. Şahin. Self-organized flocking in mobile robot swarms. *Swarm Intelligence*, 2:97–120, 2008.

[50] G. A. Vargas, O. G. Rubiano, R. A. Castillo, O. F. Avilés, and M. F. Mauledoux. Simulation of e-puck path planning in webots. *International Journal of Applied Engineering Research*, 11(19):9772–9775, 2016.

[51] A. F. Winfield and J. Nembrini. Safety in numbers: fault-tolerance in robot swarms. *International Journal of Modelling, Identification and Control*, 1(1):30–37, 2006.