Universidad Zaragoza

Trabajo Fin de Máster

Development of an environment for multi-robot distributed control simulations on the Robotarium platform

Author

Pablo Rived Foncillas

Supervisors

Rodrigo Aldana López

Enrique Teruel Doñate

MASTER IN ROBOTICS, GRAPHICS AND COMPUTER VISION

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2024

Abstract

The development of multirobot algorithms is an interesting topic in modern robotics, with applications ranging from autonomous exploration to logistics and emergency response. This project highlights the importance of testing algorithms capable of efficiently coordinate teams of robots in distributed environments. These systems are capable of dividing tasks to operate simultaneously and collaborate to achieve common goals. This enhances efficiency, scalability, and robustness against failures.

The project consist of developing a library called Distributed Robotics Execution and Management (DREAM) for simulations focused on distributed multi-robot strategies. Subsequently, to test the library, several algorithms focused on different areas of distributed strategies are implemented. These algorithms are simulated using the library to evaluate both their effectiveness and the library's flexibility to adapt to various concepts. Then same algorithms are tested on physical robots, validating their functionality in real-world scenarios in order to compare differences between virtual and real environments.

This dual approach with simulation and practical experimentation tests if the library supporting different algorithms is not only theoretically robust but also applicable in real-world settings, thus providing a tool for efficient application development.

Contents

Chapter 1

Introduction

Multi-robot algorithms represent the core of one of the most innovative areas in the field of autonomous robotics. As technologies advance, robots are expanding their capabilities beyond single limited tasks, to collaborate in complex systems where teamwork and communication are essential to solve problems efficiently. In this context, a multi-robot algorithm is a set of instructions or rules designed to coordinate, organise and optimise the activities of multiple robots working together in order to enable these agents to function as a single, cohesive unit in pursuit of a shared goal.

Collaboration is the main idea of multi-robot systems. Each robot becomes a "piece" that contributes to a joint effort, adapting to the actions of its teammates to meet collective goals. Robots work in parallel or sequentially, dividing or joining efforts as needed. Dynamic task assignment allows each robot to perform the activity that best suits its skills, location and current state. The ability of robots to communicate with each other defines the efficiency of the system. Algorithms may reassign tasks in real-time, based on each robot's efficiency and proximity to tasks, maximising collective performance.

These systems have multiple applications in a wide range of fields, Figure 1.1. In agriculture, robots are used to monitor and treat crops [10]. For rescue work in disaster areas, multiple robots search for and locate survivors in dangerous terrain without the need to expose human personnel to unnecessary risk [19]. In automated warehouses, these algorithms organise the movements of cargo robots to maximise logistical efficiency [17]. In space exploration, multiple rovers can be deployed on unknown surfaces to map and analyse large areas simultaneously, quickly and efficiently.

Figure 1.1: Some applications of multi-robots. From left to right: Smart farming with an irrigation coverage system, Smart warehouse with an AGV system for item sorting, Drone swarm for terrain mapping in a forest [19].

1.1 Multi-robot simulators

Simulation and experimental environments play a fundamental role in the development and testing of algorithms. Simulation platforms are great for testing ideas in controlled environments before taking them into the physical world, saving time and resources. There are many simulators, each offering different approaches, tools, and capabilities tailored to the needs and complexity levels of robotics development, Figure 1.2.

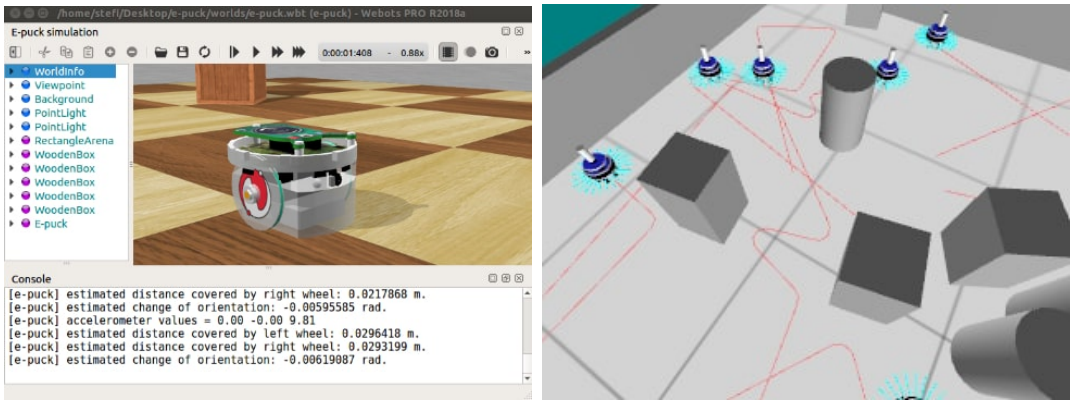
Webots, developed by Cyberbotics [9], is a simulation platform for creating and testing robots. It focuses on realistic simulation, achieved through a 3D graphics engine and a physics engine that enables precise interactions between objects.

ARGoS (Autonomous Robots Go Swarming) [12, 13] is an open-source simulation platform specifically designed for simulating robotic swarm systems. ARGoS is highly optimized to handle multiple robots simultaneously and efficiently, allowing for the simulation of thousands of robots in real-time.

Gazebo [2, 7] is a widely recognized open-source 3D simulation platform noted for its ability to model highly detailed and realistic simulation environments. Gazebo integrates an advanced and customizable physics engine, enabling the precise simulation of forces and collisions, making it possible to develop complex robots in near-real-world simulation conditions.

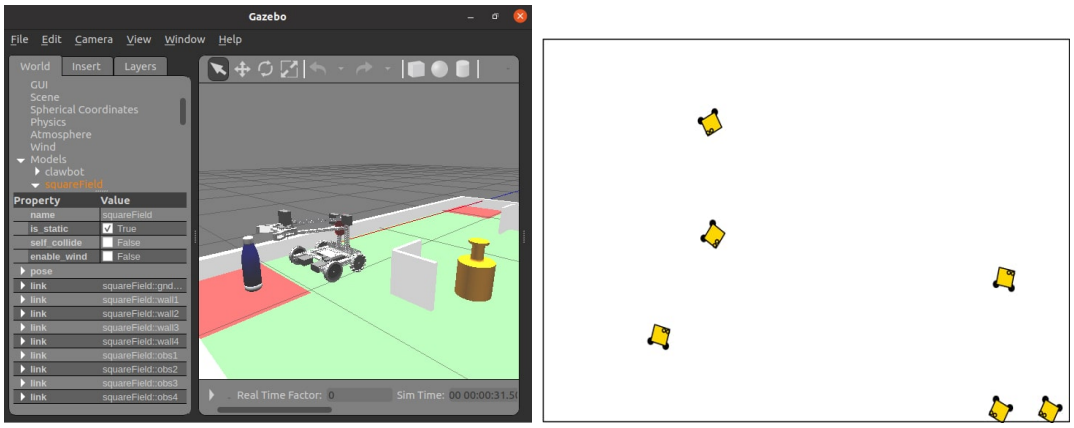
Robotarium [15] is an experimental platform created by the Georgia Institute of Technology (Georgia Tech) that provides global access to a physical testing environment for robotic systems remotely. Unlike other simulation platforms, Robotarium provides access to real physical robots, bridging the gap between simulation and testing in the physical world.

Webots, ARGoS, and Gazebo are simulation platforms that allow for designing, testing, and refining algorithms in virtual environments. However, each one has its problems and limitations. Webots has a dependency on the ODE library for physics that can severely limit its performance for multi-robot simulations. ARGoS has a



Webots Simulator

ARGoS Simulator



Gazebo Simulator

Robotarium Simulator

Figure 1.2: Screenshots of Webots, ARGoS, Gazebo and Robotarium simulators.

modular design that can introduce latencies in the communication between modules and can affect synchronisation. Gazebo uses physics engines such as ODE, Bullet or DART. Its performance decreases significantly with high robot density simulations or complex geometries, due to scalability issues.

In terms of testing algorithms with real robots, Gazebo is fully integrated with ROS and Webots partially via ROS2, allowing for algorithms to be executed on real robots. ARGoS also can be connected with ROS but its procedure is more complex. Instead, other frameworks such as Kilobot, which is designed to work with real swarms, are often used. The problem with all this is that in order to test the algorithms on real robots, it is necessary to buy, calibrate and, if needed, modify them.

In the case of Robotarium, it is not necessary to do that since the platform offers a free remote laboratory where the developed algorithms can be tested. Users can download the simulator to program in Python or Matlab and conduct experiments virtually, then send those same experiments to the lab to verify the algorithm's real-world effectiveness without cost. This way, possible errors, which are harder to simulate, can be tested without the need to acquire physical robots.

The problem with Robotarium is that is limited to the robots that the lab itself offers. The robots are all controlled from a central computer that tells them exactly what to do at all time. This means that if one want to test algorithms that involve the use of some kind of extra element, such as a position sensor, it will have to be simulated in some way, as the real robot is not customisable.

The Table 1.1 shows the main differences and use cases of the four discussed simulators.

	Webots	**ARGoS**	**Gazebo**	**Robotarium**
Main Purpose	Simulation of robots and virtual environments for research and education	Scalable and flexible multi-robot simulation	Simulation of physical robots in realistic environments	Remote execution of algorithms on real physical robots (in a shared environment)
Main Focus	Physical modeling and programming of individual robots or small groups	Optimization for large-scale multi-robot simulations	High physical fidelity for advanced industrial and robotic applications	Experimental robotics accessible for collaborative learning
Key Features	3D environments, support for multiple languages (C, Python, MATLAB), advanced physics with engines like ODE	Modularity, scalability, integration with physics simulators like Bullet and ODE	Accurate modeling of sensors and actuators, integration with ROS	Accessible platform via API, real execution on physical robots, limited simulation
Scalability	Moderate, suitable for individual robots or small groups	High, designed for massive and customizable simulations	Limited, It is highly expensive to scale to many agents	Limited, due to shared use of physical hardware
Use Cases	Education, development of mobile robots, robotic arms, and drones	Research in cooperative robotics, distributed systems, and swarming	High-fidelity physical simulations for development and testing of robotic systems	Remote experimentation with real robots for students and researchers

Table 1.1: Comparative table with different features of Webots, ARGoS, Gazebo and Robotarium.

1.2 Goals and tasks

Robotarium comes with a default library containing basic functions for localization, collision avoidance, and graph processing, all approached from a centralized perspective. In a centralized control system, there is a single control point (a central controller) that coordinates the actions of all robots.

Goal 1: develop a library that can be utilized across various simulators, including the Robotarium. For the Robotarium, the library will act as an intermediary layer between its base code and the algorithm to be executed in the simulator. The library, called Distributed Robotics Execution and Management (DREAM), is designed to support distributed systems. Distributed systems are characterized by the absence of a central control unit, so each robot must make its decisions based on its local observations and the information shared with nearby robots. Developing a distributed library will allow experimentation and testing of algorithms that reflect scenarios where robots operate autonomously. This would encourage research in areas such as multi-agent systems, cooperative robotics and applications in environments with communication limitations or without a central controller. It also offers a more versatile tool for exploring the scalability and robustness of distributed systems, extending the scope of Robotarium's capabilities and allowing more complex situations to be simulated.

Goal 2: develop a series of distributed algorithms using this same intermediate layer, covering different concepts of multi-robot systems, and successfully executing them both in the simulator and in the physical remote laboratory provided by the Robotarium. This will serve to demonstrate its versatility and functionality in a wide range of scenarios and multi-agent problems by offering concrete use cases to validate the performance of the library in tasks such as swarm formation, coverage or synchronisation in robotic networks.

1.3 Methodology and tools of the project

This project has been developed using the Robotarium simulator in Python, version October 2023 [16]. The Python version used is Python 3.10.12, with the IDE Pyzo 4.14.4. As for the libraries required for Robotarium to function, the following versions were used: scipy 1.12.0, cvxopt 1.3.2, numpy 1.26.3, and matplotlib 3.7.1.

An outdated version of matplotlib was used because, at the time of the project's completion, the Robotarium simulator had compatibility issues with more recent versions of the library.

Chapter 2

Breakdown of the DREAM library

This chapter expands on the proposed DREAM library to be used for distributed robotic systems simulation. The Robotarium, as it is designed, offers a great deal of freedom when it comes to programming over it. However, it features a highly centralized simulation, where all agents are controlled from a master program. Our aim is to introduce a middleware between the application and Robotarium to enable simulating distributed algorithm.

When creating an algorithm, the user inputs a $2 \times N$ matrix, where N is the number of active robots in which the first row is the linear velocity value of each robot and the second row is the value of its respective angular velocity. Taking advantage of this, the library will be composed of a series of classes and subclasses with basic functions that will be used to provide the robots with perception, control and communication capabilities so that all these elements pursue the final objective of building this matrix.

The library will consist of four main classes adapted to distributed systems:

- A Robot class, where the robot parameters will be defined and the movement of the actuators will be calculated.
- A Sensor class, which will be in charge of creating all the robot's perception devices.
- A Radio class that will serve as a communicator, allowing the robot to send and receive messages.
- A Timer class that will be used to control when each of the robot's components must be activated.

This will be done to simulate the elements that physical robots do not have (perception and communications gadgets) and to simulate the delays that each of the parts of the robot may have (timer) because they often cannot be constantly updated in the real

setting. Figure 2.1 shows how the final structure of a project is composed, having an application running over DREAM which itself runs over the simulator.

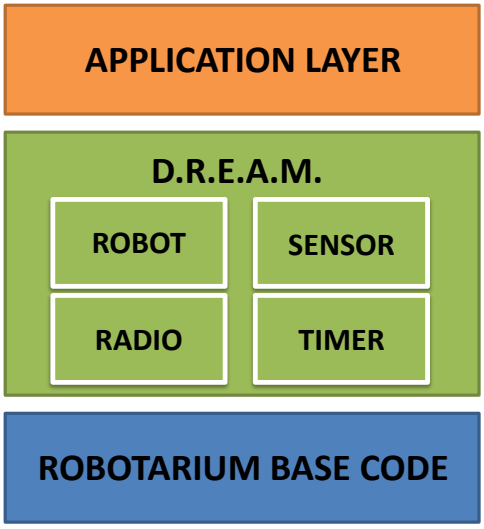


Figure 2.1: Layers of the final code of an algorithm: on top of the base code of the robotarium is the library, on top of which the application is programmed.

2.1 Robot class

The Robot class is the main core of DREAM. It contains all the physical parameters of the robot and all the necessary functions to calculate the robot's movement. The Robot itself cannot communicate with the outside world, in the sense that it cannot know where it is or if there is anything near its position. The Robot must receive all this information through the different accessories that are attached to it because the Robot class must be completely independent from the simulator. It is the other elements that will be adapted to each type of environment while maintaining an invariant robot class. Figure 2.2 shows the communications of the Robot. Receives data from Sensors, Radios and Timer, and can send data to Radios and actuators.

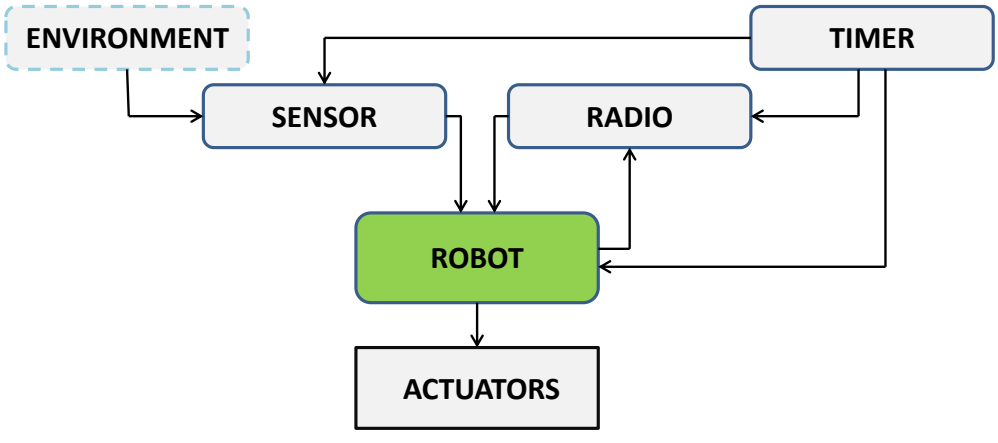


Figure 2.2: DREAM communications: Robot.

The actuators in this case, will be the control applied to the wheels of the robots to make them move. In Robotarium, this is done by constructing a $2 \times N$ matrix with N being the number of robots in the simulation and sending it to Robotarium's `set_velocities(...)` function. Each column corresponds to a single robot, with the first row being its linear velocities and the second row its angular velocities. The object of the Robot class is to calculate these velocities so that the matrix can then be constructed and sent to the function.

Main Robot class:

When creating an algorithm, a Robot class inherited from this one must be created containing the algorithm to be tested. This class simply contains the basic functions and parameters for the robots to exist.

- **Parameters:**
  - `robot_id`: ID number of the Robot. This number is the identifier in the Robotarium. Its main function is to indicate its position in the position matrix provided by the Robotarium and in the velocity matrix to be sent to the Robotarium.
  - `tag`: Name tag of the Robot.
  - `vel_gain`: Velocity gain used for linear velocity calculation. This adjusts the ratio between the desired speed command and the actual speed the robot achieves.
  - `ang_gain`: Velocity gain used for angular velocity calculation. This adjusts the relationship between the angle command and the actual rotation of the robot.
- **Attributes:**
  - `pos_x`, `pos_y`, `rot_w`: Individual variables for storing robot x, y position and w rotation data.
  - `dxi`: Used to store velocities in single-integrator dynamics system. This variable is used as a placeholder for this type of speeds. Depending on the application to be developed, it may not be necessary to be used.
  - `dxu`: Used to store velocities in unicycle dynamics system. This variable is used as a placeholder for this type of speeds. Depending on the application to be developed, it may not be necessary to be used.
- **Methods:**
  - `actuator_limit()`: This method is used to limit the calculated speed to

the actual limits of the robot. After calculating the speed, it is possible that the result may exceed the maximum permissible rotational limits of the robot's wheels. Calling this method reduces the linear speed and maintains the angular speed in such a way that the maximum rotational speed of the wheels is not exceeded. It has been decided to reduce the linear speed instead of the angular speed because the angular speed has been considered more important for the execution of the final movement.

- `dxi_to_dxu()`: Most of the calculations performed are done in the single-integrator dynamics system. As the final result of velocities must be in the unicycle dynamics system, calling this function transforms the single-integrator data contained in the variable dxi to unicycle and stores it in the variable dxu.

Robot_hrvo subclass:

A subclass with the Hybrid Reciprocal Velocity Obstacle (HRVO) algorithm already implemented. The aim of this subclass is to provide a system to avoid collisions of robots with each other. Robots that have been created by inheriting this class will be able to use the algorithm to recalculate their movements to avoid hitting each other. It has been created in a separate class as this algorithm will not always be wanted to be used. The explanation of the algorithm will be detailed in chapter 3.

- **Parameters:**

  In addition to the parameters of the main class, this subclass has these two extra parameters

  - `safety`: Safety radius. Minimum distance the robot can be brought closer to others.

  - `check_boundaries`: Indicates whether the HRVO must take terrain boundaries into account when recalculating movement.

- **Methods:**

  - `HRVO(detections, desired_velocity)`: This method receives as input parameters an array with the positions and velocities of the surrounding robots and the velocity that the robot itself would like to have in the single-integrator system. The method stores the calculated single-integrator velocity value in the variable dxi.

2.2 Sensor class

The Sensor class is responsible for detecting and measuring changes in the environment or in the internal state of the system. This data obtained will allow the robot to perceive the world around it in order to facilitate decision making and task execution.

There is no maximum or minimum limit to the number of Sensors that can be attached to a robot. Communication in the programmed classes between robot-sensor is done via callback-methods, so that the robot has a method that can be called during the sensor update in order to transfer data. Figure 2.3 shows the communications of the Sensor. Receives data from the environment and the Timer, and can send data to the Robot.

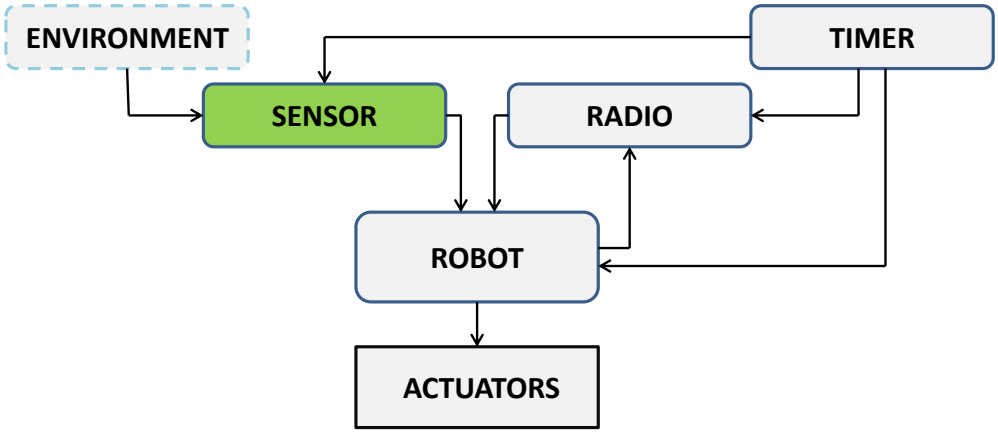


Figure 2.3: DREAM communications: Sensor.

Main Sensor class:

Main Sensor class. Contains basic parameters and functions for all sensor classes that will inherit from here.

- **Parameters:**
  - `robot_id`: ID number of the Robot to which this Sensor is attached.
  - `tag`: Name tag of the Sensor.
- **Attributes:**
  - `pos_x`, `pos_y`, `rot_w`: Individual variables for storing its x, y position and w rotation data. (The same as its Robot)
- **Methods:**
  - `add_noise(values, noise)`: This method receives as input parameters an array with some data and a float number as noise value. The aim of this method is to pass the data detected by the sensor and add some noise to it by

using a normal distribution in order to simulate non-perfect measurements.

- `update(x)`: This method receives x as the position matrix from the Robotarium and updates the Sensor position as a starting point for measurements. This is used because for simulation purposes, the data read from the sensor may not be perfect, but its point of origin should be.

Movement_sensor subclass:

This subclass is designed to be compatible with the `robot_hrvo` class as it is responsible for detecting the positions and velocities of all robots that fall within its detection range.

- **Parameters:**

  In addition to the parameters of the main class, this subclass has these extra parameters. Figure 2.4 shows how this sensor works.

  - `callback`: This must be a method of the robot that makes it possible for it to receive the data detected by the sensor.
  - `range`: Numerical value indicating how far away the sensor can detect things.
  - `direction`: Numerical value indicating the rotation relative to the front of the robot in radians of the sensor.
  - `wide`: Numerical value indicating the opening angle of the sensor in radians.
  - `noise`: Numerical value indicating the noise value of the detections in order to simulate non-perfect detections.

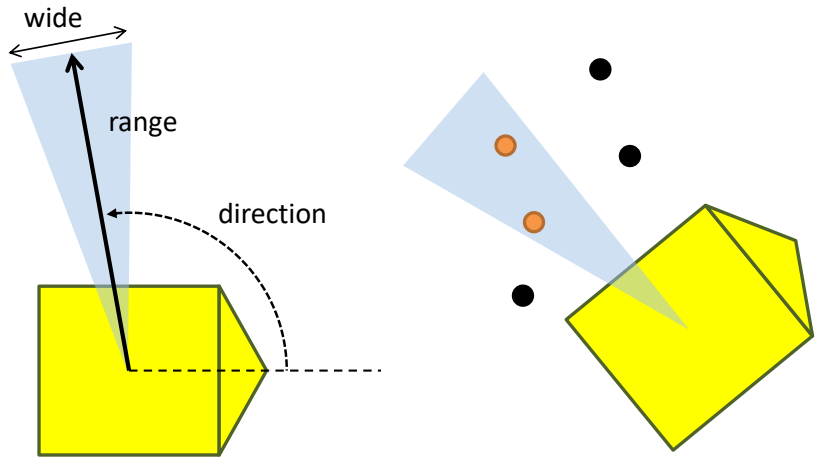


Figure 2.4: Visual representation of the sensor orientation parameters over a robot, assuming that the front of the robot is the tip. In orange, the detected elements. In black, the undetected elements.

- **Methods:**
  - `update(x, v)`: This method receives the position matrix from the robotarium and the velocity matrix of the robots. With this data, it will form a list of position and velocity detections based on the positions of the robots on the ground and the detection area of the sensor itself.

Odometry_sensor subclass:

This subclass inherits from the main sensor class. Its purpose is to simulate the calculation of the robot's movement that an odometry sensor would do, so that the robot has a sense of where it is located.

- **Parameters:**

  In addition to the parameters of the main class, this subclass has these two extra parameters
  - `callback`: This must be a method of the robot that makes it possible for it to receive the odometry data by the sensor.
  - `noise`: Numerical value indicating the noise value of the detections in order to simulate inaccurate odometric calculations.
- **Methods:**
  - `update(x)`: This method receives the position matrix from the robotarium. With this data, it will some noise and then send it to the Robot by its callback method.

2.3 Radio class

The Radio class will be in charge of establishing communications between the robots. This class will be in command for managing the sending and receiving of messages from each of the robots. Unlike the other elements that can be attached to the Robot, its operation is bidirectional, since the Robot must be able to receive data from the radio and send information to it.

There is no maximum limit to the number of radios that can be attached to the robot, and communication with the Robot is carried out in the same way as with the sensor, by using callback methods. Likewise, it is not mandatory to attach the Radio to any Robot in the case of wanting to simulate a signal located at a precise point. Figure 2.5 shows the communications of the Rensor. Receives data from the Robot and Timer, and can send data to the Robot.

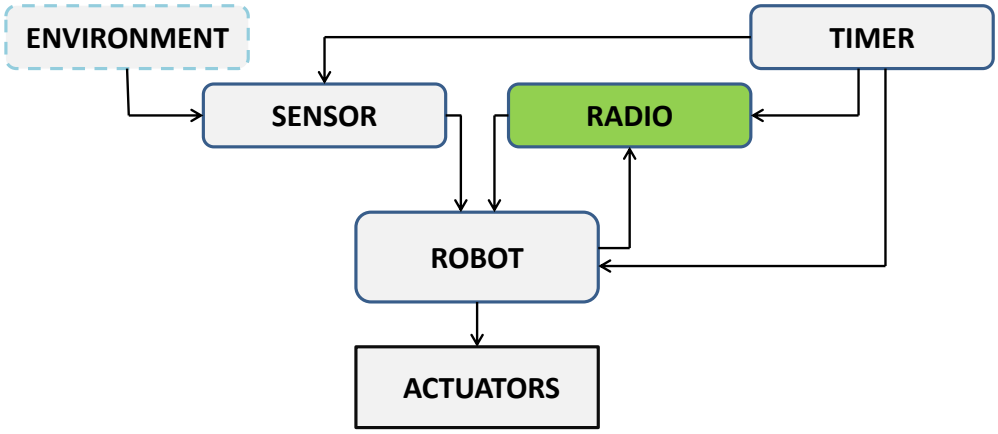


Figure 2.5: DREAM communications: Radio.

It works as follows: There is a global dictionary to store the messages in which the Radio will publish the messages of each robot. When a message is published, it will be sent there, assigning the Robot ID as key and the message sent as item. When a robot is going to download a message, the Radio will search this list for the correct messages and send them to the Robot. It is up to the application what kind of messages will be sent or received.

`Messages dictionary` = {`robot_id` : `data`, ...}

Main Radio class:

Main Radio class. Contains basic parameters and functions for all Radio subclasses that will inherit from here.

- **Parameters:**
  - `robot_id`: ID number of the Robot to which this Radio is attached.
  - `tag`: Name tag of the Radio.
- **Attributes:**
  - `pos_x`, `pos_y`, `rot_w`: Individual variables for storing its x, y position and w rotation data (The same as its Robot).
- **Methods:**
  - `send_data(data)`: This method is used to send messages to the global message dictionary, assigning the id of the Robot in the key and the received data as an item.
  - `update(x)`: This method receives the position matrix from the Robotarium and updates the Radio position as a starting point in case a position-based

send-receive data is programmed.

Radio_range class:

This subclass is inherited from the main Radio class. It is responsible for downloading messages via proximity. In Figure 2.6, all robots post their respective message in the dictionary. In the case of robot 1, when downloading data, it searches by range for nearby robots and downloads their corresponding data, leaving robot 0's data undownloaded as it is too far away.

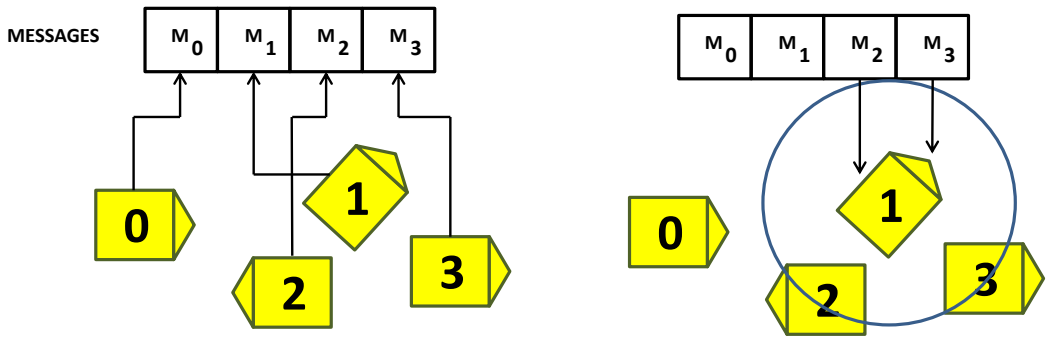


Figure 2.6: Radio send and download messages. Robot 1 downloads data from robots 2 and 3 but not from 0.

- **Parameters:**

  In addition to the parameters of the main class, this subclass has these extra parameters

  - `callback`: This must be a method of the Robot that makes it possible for it to receive and send the data detected by the radio.
  - `range`: Numerical value indicating how far away the Radio can download data.

- **Methods:**

  - `update(x)`: This method receives the position matrix from the Robotarium. With this data, it will form a list of messages based on how close are the Robots within this Radio. If the Robots are within the defined range, the data will be downloaded. In addition, the callback method used to send the data to the Robot will return the data that the Robot wants to be published via the send_data() method.

2.4 Timer class

This Timer class is in charge of indicating when to activate or deactivate all the elements of the robot. The Timer must be connected to all the elements of the system that must be executed in a non-immediate way. In the real world, an element such as a sensor is not registering information with a non-existent time window. It is executed in discrete time, i.e. a countable number of times per second. The Timer is created precisely to simulate these time gaps. The Timer is intended to be executed in each iteration of the simulation as if it were the main controller of the Robot. It will control which part of the Robot will be activated during that iteration. The Figure 2.7 shows how the Timer influences the other classes Robot, Sensor and Radio.

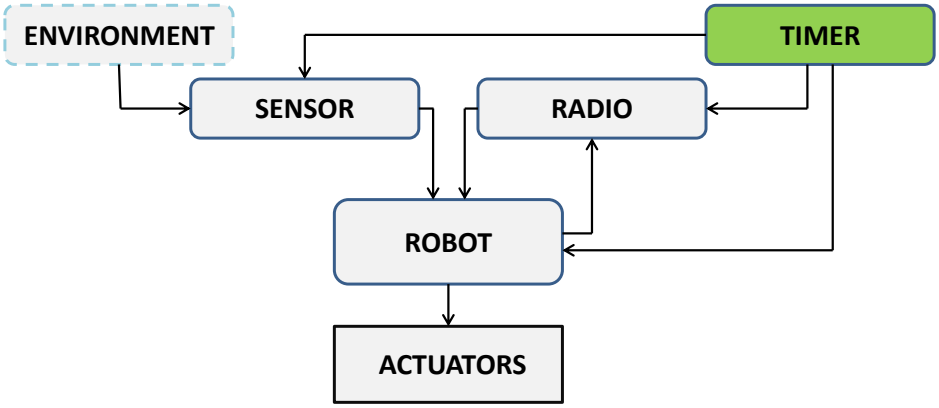


Figure 2.7: DREAM communications: Timer.

The Timer class, unlike the others, is not intended to create subclasses of it since it is not a physical part that would be attached to a real robot. This class consists of the class itself to control a Robot and two extra functions to create and update the Timer. The Robotarium runs at thirty iterations per second, so the minimum time that can be associated with a gap is 0.033 seconds, or 33 milliseconds. In Figure 2.8, we see an example of something that is being updated every three simulation steps, indicating that during the two intermediate steps, that element is not on work.

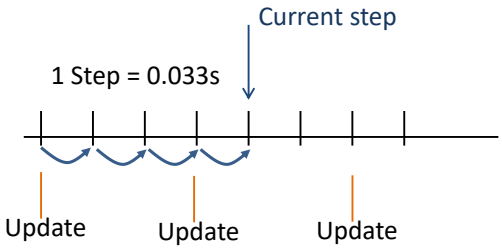


Figure 2.8: Example of the control applied of the Timer: Something updating every three steps while the remaining steps is resting.

Function `create_timer(···)`:

The Timers are stored in a global list. This function is in charge of creating the Timers and adding them to this list.

- **Parameters:**
  - `robot_id`: ID number of the Robot to which this Timer is attached.
  - `timer_delay`: Numerical value indicating the time that elapses until the timer starts counting time, or in simulation terms, the time the Robot is off until it is switched on and starts running.
  - `list_of_timers`: The list of gap times associated with the Timer. It must be a list of dictionaries. Three keys-items must be added to each dictionary:
    - "delay": a numerical value indicating the number of gap time between updates.
    - "tag": a string referring to the callback method to be called at the specified time.
    - "call": the callback method to be called.

Function `update_timers(···)`:

This function iterates over the Timers list and call its `update()` function. It is expected to be called at each iteration of the simulation.

- **Parameters:**
  - `current_time`: Current time of the simulation.
  - `param`: Dictionary whose keys are the tags associated to each method in the Timers list and as item a tuple with the input parameters of those methods. `parameters dictionary` = {`tag` : (`params`), ...}

Chapter 3

Experimental validation

This chapter describes several algorithms implemented over DREAM, in order to test its capabilities and benefits. A total of four experiments have been programmed using different concepts of multi-robot algorithms: collision avoidance, rendezvous, consensus, swarm robotics and coverage.

- For collision avoidance, the Hybrid Reciprocal Velocity Velocity Obstacle (HRVO) algorithm has been implemented and as described in chapter two, it is integrated into the base architecture so that it is available to be interfaced with any other algorithm.
- Rendezvous and consensus are going to team up on an algorithm that leads the robots to perform a straight line formation, to test types of communication between robots. HRVO will also be used to avoid collisions between them.
- In terms of swarm robotics and coverage, an algorithm will be implemented that is capable of distributing robots evenly throughout the proving ground using a technique that uses voronoi diagrams.
- The last experiment tests a state-of-the-art algorithm for leader-follower tracking affine formations by consensus.

This aims to test if DREAM is not only suitable for one type of algorithm but is compatible with all kinds of different multi-robot strategies.

3.1 Hybrid Reciprocal Velocity Obstacle

3.1.1 Theoretical background

The Hybrid Reciprocal Velocity Obstacle (HRVO) presents a technique to avoid collisions between multiple moving agents whose trajectories are at risk of crossing. As seen in *The Hybrid Reciprocal Velocity Obstacle* by *Snape, J* [14], this technique improves real-time trajectory computation seeking a hybrid solution that reduces unnecessary

oscillations by combining the concepts of Velocity Obstacles (VO) and Reciprocal Velocity Obstacles (RVO).

In the context of multi-agent navigation, a Velocity Obstacle (VO) defines a set of velocities that would lead to a collision with a moving obstacle. For each agent, a VO is constructed based on its velocity and relative position with respect to the obstacle (or neighbouring agent). If an agent's velocity is within the VO, the agent runs the risk of colliding with the obstacle in the near future, so it must change its velocity to avoid the collision.

The Velocity Obstacle (VO) is defined as:

$$VO_{A|B} = \{\mathbf{v} \mid \exists t > 0,\, \mathbf{p}_A + \mathbf{v}t \in D(\mathbf{p}_B + \mathbf{v}_B t, r_A + r_B)\} \tag{3.1}$$

where:

- $\mathbf{p}_A, \mathbf{p}_B$: positions of agents A and B, respectively.
- $\mathbf{v}_A, \mathbf{v}_B$: velocities of A and B, respectively.
- r_A, r_B: radii of agents A and B.
- $D(\mathbf{c}, r)$: a disk of radius r centred at $\mathbf{c}$.

Reciprocal Velocity Obstacle (RVO) is a VO enhancement that instead of assuming that only one of the agents will adjust its velocity to avoid a collision, it assumes that both agents collaborate and adjust their velocities simultaneously. This is done by taking into account an "average speed" between the agents. By doing so, cooperation is achieved which, in theory, reduces the risk of zigzagging. The Reciprocal Velocity Obstacle (RVO) is defined as:

$$RVO_{A|B} = \{\mathbf{v} \mid \exists t > 0,\, \mathbf{p}_A + (\mathbf{v} - \mathbf{v}^r_{AB})t \in D(\mathbf{p}_B + \mathbf{v}_B t, r_A + r_B)\} \tag{3.2}$$

where:

- $\mathbf{v}^r_{AB} = \frac{\mathbf{v}_A + \mathbf{v}_B}{2}$: the relative average velocity.
- The remaining terms $(\mathbf{p}_A, \mathbf{p}_B, \mathbf{v}_A, \mathbf{v}_B, r_A, r_B, D(\mathbf{c}, r))$: are the same as in VO definition.

HRVO combines the best of both VO and RVO concepts. The main difference is that HRVO takes the current position and velocity of the agent, but instead of calculating the average velocity as in RVO, it projects the velocity of the agent at the edge of the RVO. The projection ensures that any change in velocity stays as close as possible to the RVO velocity, avoiding even more oscillations.

The Hybrid Reciprocal Velocity Obstacle (HRVO) is defined as:

$$HRVO_{A|B} = \{\mathbf{v} \mid \exists t > 0,\ \mathbf{p}_A + (\mathbf{v} - \mathbf{v}^h_{AB})t \in D(\mathbf{p}_B + \mathbf{v}_B t, r_A + r_B)\} \tag{3.3}$$

where:

- $\mathbf{v}^h_{AB} = \mathbf{v}_B + \frac{\mathbf{v}_A - \mathbf{v}_B}{2}$: the hybrid relative velocity.
- The remaining terms $(\mathbf{p}_A, \mathbf{p}_B, \mathbf{v}_A, \mathbf{v}_B, r_A, r_B, D(\mathbf{c}, r))$: are the same as in VO and RVO definitions.

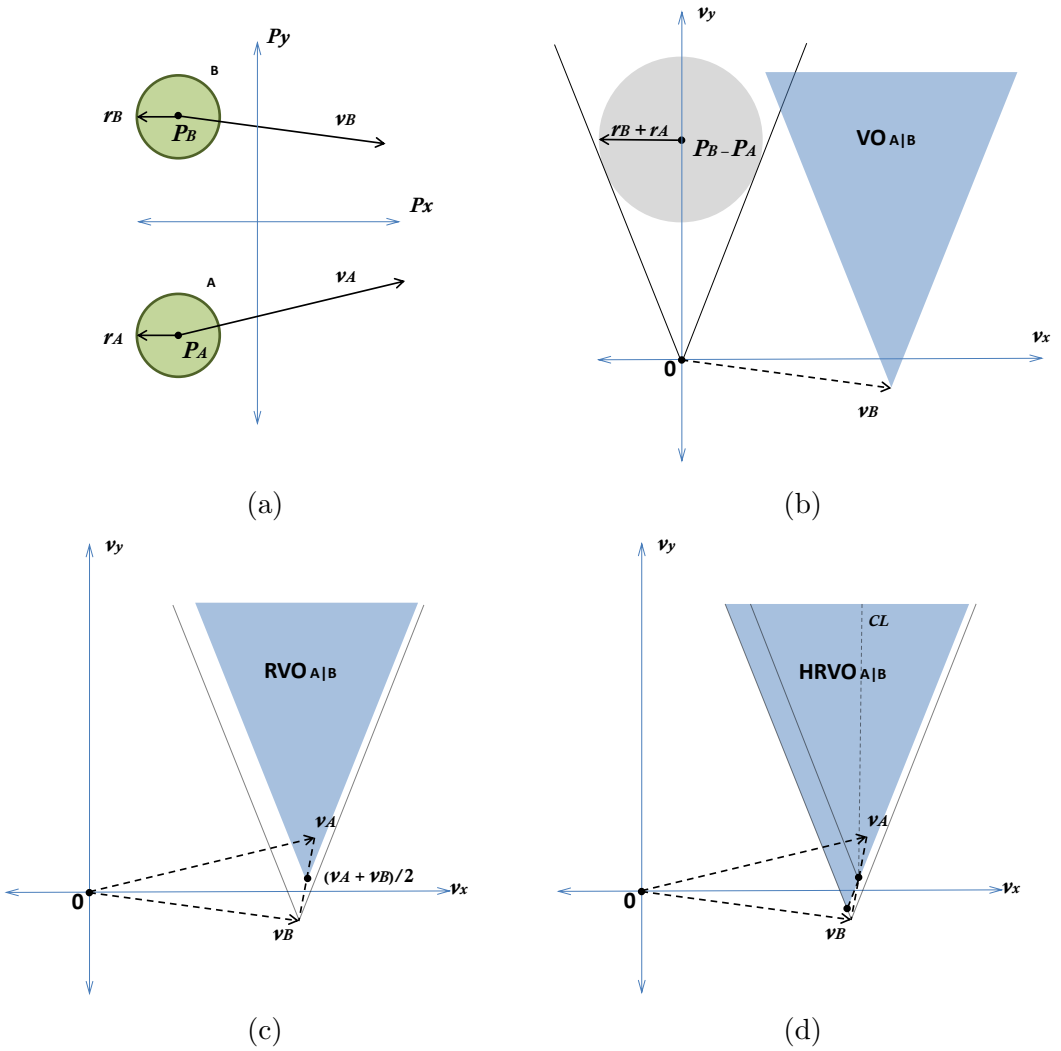


Figure 3.1: a) Two agents in a space with some velocities that may lead to a collision. b) Set of velocities from A that will lead to a collision to B. c) Set of RVO velocities from A that will lead to a collision to B, with its vertex in the mid point between vA and vB d) Final HRVO set of velocities from A that will lead to a collision to B, with left side triangle extension.

To define the set of velocities that will lead to collisions, the positions and velocities of the agents involved must first be defined. We assume that agent A wants to avoid

collisions with agent B. The positions (P), velocities (v) and the safety radius (r) of the agents A and B are shown in Figure 3.1a.

The VO represents the set of velocities of A that will lead to a collision with B. This set is constructed by taking the position of A as the origin and projecting onto the position of B a diameter that will be the sum of the radii of both agents. From the origin, two lines are drawn in the form of a triangle that will be tangent to the circle formed around B. Finally, the velocity of B is taken and, starting from the origin, the triangle constructed is transferred to that point. Figure 3.1b shows the triangle formed. Any velocity within the blue area is a velocity that leads to a collision.

The next step is to construct the RVO by averaging the velocities of both agents. As seen in Figure 3.1c, the vertex point of the triangle is located right in the middle of the A and B velocities.

The last step is to construct HRVO by extending one of the sides of the RVO to its corresponding side of the VO. To do this, a centre line (CL) is drawn starting from the vertex and dividing the triangle in half. Depending on which side of the line the velocity of A is on, the triangle must be extended in the opposite direction. As seen in Figure 3.1d, the velocity of A is on the left side of CL, so the triangle has been extended to the left side.

This process can be repeated for each agent within the detection range of A. Figure 3.2 shows the result of the sum of A-ineligible speeds defined by three external agents. Any speed chosen that does not fall within the blue area is a safe speed. The last step of the algorithm is to choose from all safe options the one that is closest to the original speed, or in other words to the desired speed.

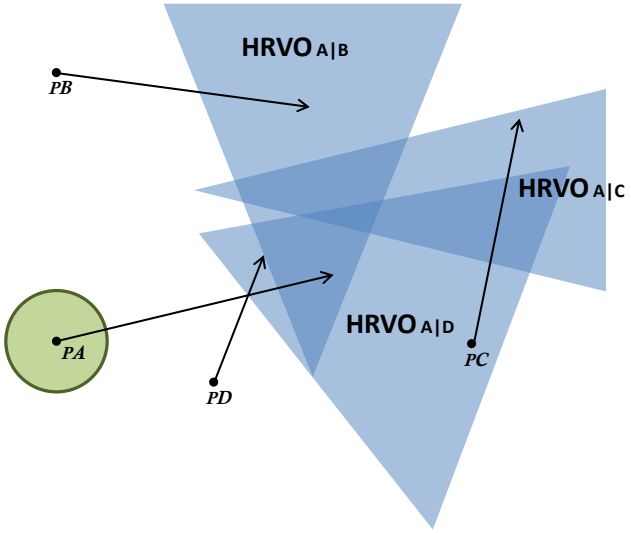


Figure 3.2: Set of collision velocities from A to three external agents.

3.1.2 Implementation over DREAM

This experiment consists of testing the effectiveness of the HRVO algorithm implemented in the code base. Ten robots will be used for the simulation.

These robots will have a random starting position and will have to form a circle formation. This formation is already given by the code, i.e. from the beginning the robot knows the coordinates of the point it has to reach. This path must be made by dodging the other robots by detecting them with the sensors. As they reach their destination, they will tell each other by radio that they have arrived. Once everyone has arrived and everyone has found out that the others are also in place, the robots will start moving again to form a new circle, this time with their position changed to the diametrically opposite position. This will be repeated until three circles have been formed in total.

The following classes have been used in the code:

- **Robot**: The robot programmed for this experiment will inherit from DREAM's Robot_HRVO class. This robot will manage movement and use HRVO to avoid other robots.
- **Sensor**: They will use two types of sensors each. DREAM's odometry sensor to know their own position and DREAM's movement sensor to check all nearby robots.
- **Radio**: The DREAM's range-operated radio will be used to communicate whether the robot is in position or not. It will also communicate what it knows about the other robots. So, when the robot knows that everyone is in position, it can continue.

Goals of the experiment:

- Test the effectiveness of the HRVO by having zero collisions at the end of the experiment.
- Visualise how the HRVO works by drawing on the ground the recalculated trajectory of the robot at each moment.
- Check the use of the motion sensor to correctly detect neighbouring robots, the odometry sensor to check position and the radio to send and receive messages.

3.1.3 Experimental results

When the experiment starts, the robots appear in random positions distributed on the ground. In Figure 3.3a dots and colored lines show which position each one is going to head for. The larger points indicate the final target of the robot. The smaller points that are close to the robot are the path defined by the HRVO as the best path at the moment for the robot to reach the target.

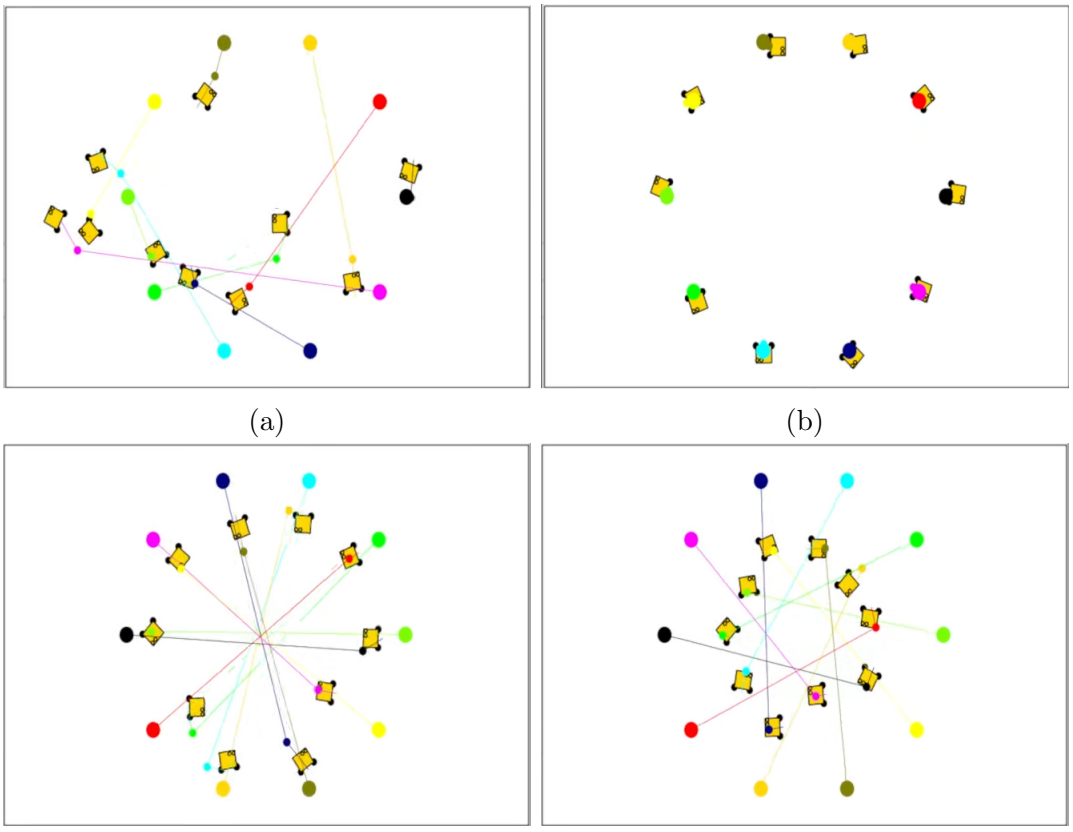


Figure 3.3: Performance on the simulator. a) Random starting positions of the robots with a marked path to their goal. b) Circle formation. Robots stay in position until they get the information that everyone is also in position. c) Robots advance towards the centre because that is the shortest path to their diametrically opposed goal. d) Robots rotating clock-wise to avoid collisions. The simulation can be found at https://youtu.be/Tsl9rE_bZK0.

Figure 3.3b shows the robots when they have reached the position. Here, they have to send a message that they have arrived. Each robot stores in its memory the status of all the other robots and must radio back both its own status and what it knows of the status of the others. The radio works by range, so the robots depend on their neighbors to communicate the status of robots further away, with whom they cannot communicate, creating a chain of messages.

As soon as the robot has detected that all the others are in place, it updates its

target and moves to the next position. As all the robots start almost at the same time, they all start to move to the centre and end up rotating to avoid each other as seen in Figures 3.3c and 3.3d.

After successfully completing the experiment, the experiment was sent to the Robotarium, with a very similar result as in the simulator. Although the initial position of the robots was not the same, as this position was random, the robots did not show any problem when it came to dodging, communicating and reaching their respective destinations.

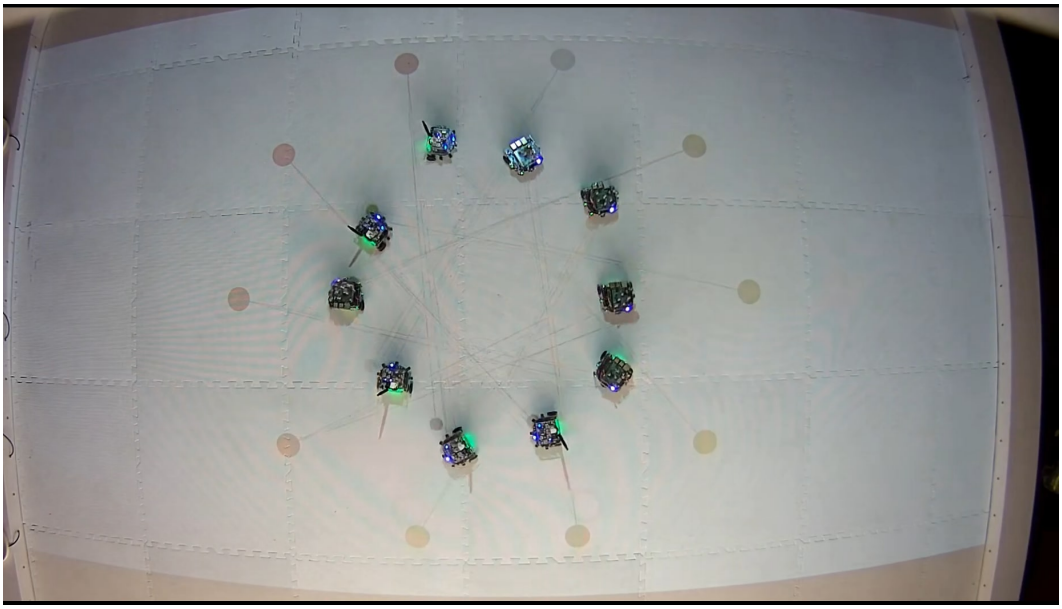


Figure 3.4: Performance on the Robotarium. The robots describe similar behaviour than the simulator, such as rotation to avoid collisions is also clock-wise. The video can be found at https://youtu.be/F-MiuYq3QhU.

3.2 Line Deployment

3.2.1 Theoretical background

In *Consensus and Cooperation in Networked Multi-Agent Systems* [11], a robust theoretical explanation of consensus in multi-agent systems is presented.

Consensus in multi-agent systems refers to the process by which autonomous agents manage to agree on certain state values in a decentralised manner. The key is that these agents communicate only with their immediate neighbours based on graph theory allowing for greater scalability and robustness to failures.

Each agent is represented as a node in a graph, while connections between nodes indicate the possibility of direct communication between two agents. Each agent has

a state (e.g. its position) that it updates using the information it receives from its neighbours.

A multi-robot system is modelled as a graph $G = (V, E)$ where:

- $V = 1, 2, ..., n$: A set of nodes, each representing a robot.
- $E \subseteq V \times V$: Set of edges, representing the communication channels between robots.

In addition to that, there is an adjacency matrix $A = [a_{ij}]$ corresponding to:

$$a_{ij} = \begin{cases} > 0 & \text{if there is a link between robots i and j} \\ 0 & \text{if there is no link} \end{cases} \tag{3.4}$$

The state of each robot i at time t is denoted as $x_i(t)$. The goal of consensus is for the states $x_i(t)$ of all robots to converge to a common value x^* as $x \to \infty$. An ideal algorithm to perform such purpose is:

$$\dot{x}_i(t) = \sum_{j \in \mathcal{N}_i} a_{ij}\big(x_j(t) - x_i(t)\big) \tag{3.5}$$

Where:

- $x_i(t), x_j(t)$: State of robot i, j at time t.
- $\mathcal{N}_i$: Neighbours of node i in the network (robots with which i can communicate with)
- a_{ij}: Weight of the connection between the robots i, j.

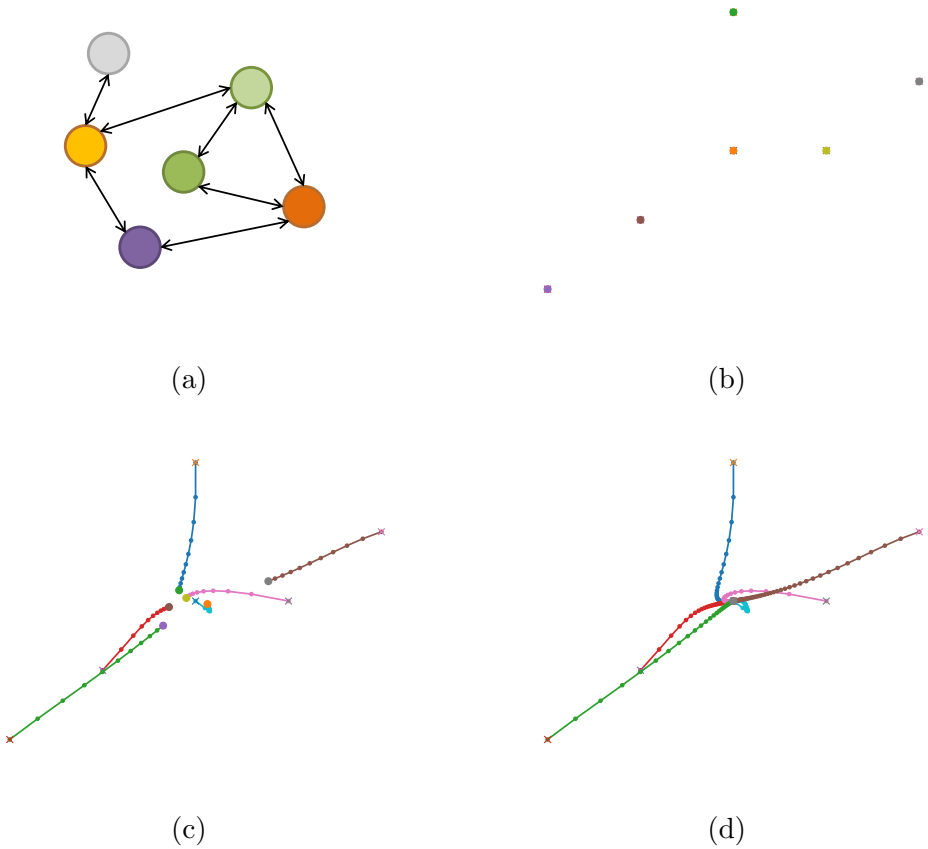
In the following, we use a simplified discrete-time version of (3.5) to perform consensus.

The rendezvous problem is a strategy in which a group of autonomous robots, which are normally dispersed in a region, coordinate to meet at a single point or at nearby positions, regardless of their initial location. *The Multi-Agent Rendezvous Problem* [5] paper proposes a resolution of this problem by consensus in which to reach the end point in a decentralised way each agent uses a local strategy that allows it to reduce the distance with respect to its neighbours, guided by a gradient towards the centroid of its nearby positions.

To calculate the centroid (C_x, C_y), the average position of each robot with its neighbours is calculated.

$$C_x = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} x_j \qquad C_y = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} y_j \tag{3.6}$$

where $|\mathcal{N}_i|$ is the number of Neighbors of agent i.



(a) (b)

(c) (d)

Figure 3.5: Example of rendezvous. a) The agents are linked by a graph that they will use to communicate. b) c) d) They start at random positions on the ground and end up converging at the same point.

3.2.2 Implementation over DREAM

For this experiment, what will be done is a variation of the rendezvous using the same technique described in *The Multi-Agent Rendezvous Problem* [5]. The goal is not for the robots to reach the same point in space, but for them to end up in a line formation. To do this, they will have to equalise their y positions and make a distribution along the x axis.

The robots will use the same sensors as in the previous exercise and will use the HRVO to avoid colliding with each other. As for the radio, a new radio will be created whose data download criterion will not be based on range but in a graph neighbour consensus based.

The following classes have been used in the code:

- **Robot**: The robot programmed for this experiment will inherit from DREAM's Robot_HRVO class. This robot will manage movement and use HRVO to avoid other robots.
- **Sensor**: They will use two types of sensors each. DREAM's odometry sensor to know their own position and DREAM's movement sensor to check all nearby robots.
- **Radio**: A new radio will be created from DREAM's main radio whose criteria for downloading data will not be limited by range but by links between robots according to some adjacency matrix. The adjacency matrix for this exercise is as follows with sufficient connections to ensure it is a connected graph, meaning that there is a communication path between each pair of agents:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

To be distributed along the x axis, the Robot ID number will be used as the position on the line and there shall be a defined distance between robots. Each robot shall calculate its relative x position based on the current positions of its neighbours and taking into account the distance between robots. The information to be sent by radio should be the xy position itself as in the article and also its robot id.

With d as a defined distance between robots, Rid as a Robot ID and P as (xy) positions, for each Robot i with n neighbours its new position P_i its calculated with each neighbour's P_j:

$$P_{ix} = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} (P_{jx} + (\mathsf{Rid}_j - \mathsf{Rid}_i) \cdot d) \qquad P_{iy} = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} P_{jy} \tag{3.7}$$

Goals of the experiment:

- Conduct a line formation of six robots with a consensus-rendezvous based system.
- Test creating new inherited radio class that perform different functions than the already created in DREAM.
- Test an algorithm with dynamic end positions and not fixed positions.

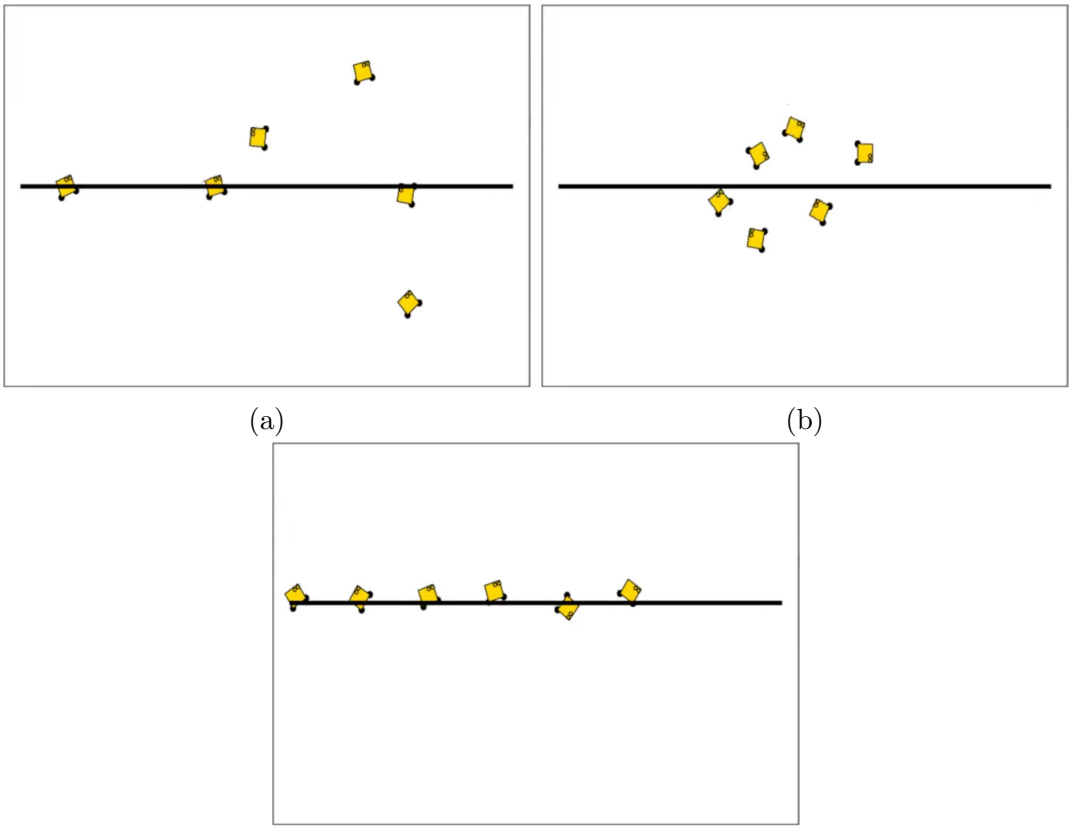


Figure 3.6: Performance on the simulator. a) Initial random position of the robots in the terrain. b) Robots moving towards its recalculated position. They need to dodge as they are not able to go in a straight line. c) Final line formation along the printed black line. The simulation can be found at https://youtu.be/JRtoymMNYD4.

3.2.3 Experimental results

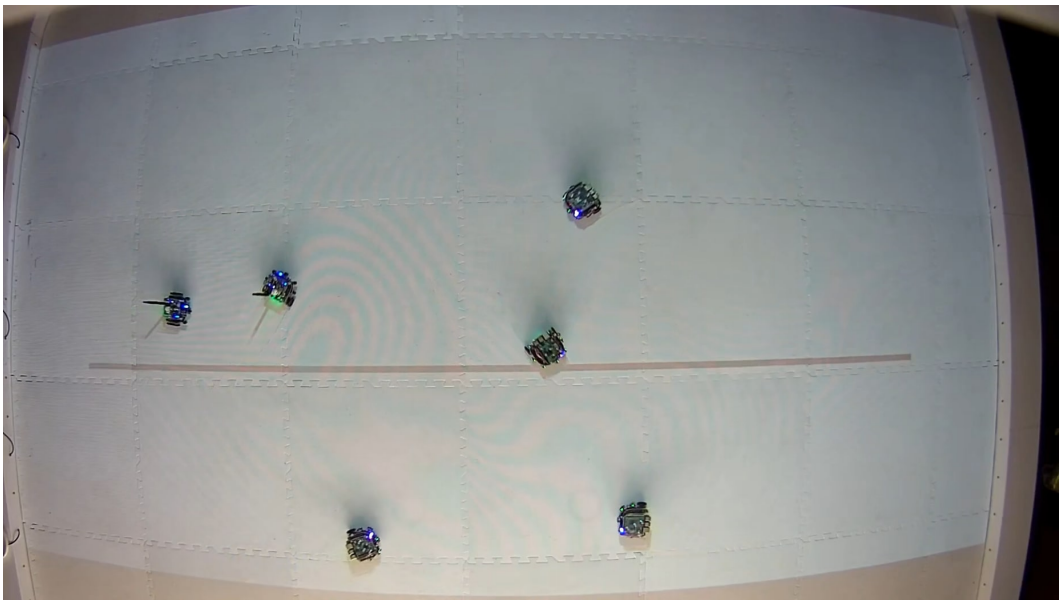
The experiment starts with the robots placed in random positions Figure 3.6a. There is a line is in a middle position between all the robots. For illustrative purposes only, a line has been drawn on the ground that is located at the average y position of all robots. As the experiment progresses, this line should stop oscillating vertically and the robots should converge just above it, maintaining a final equilibrium position.

As the simulation progresses, the robots reorganise themselves to form the position in a straight line. The robots are getting closer together and the oscillation of the marked line is getting smaller and smaller Figure 3.6b. The robots, unlike in the previous experiment, do not have a defined final position and are recalculating it each time the robot controller is updated.

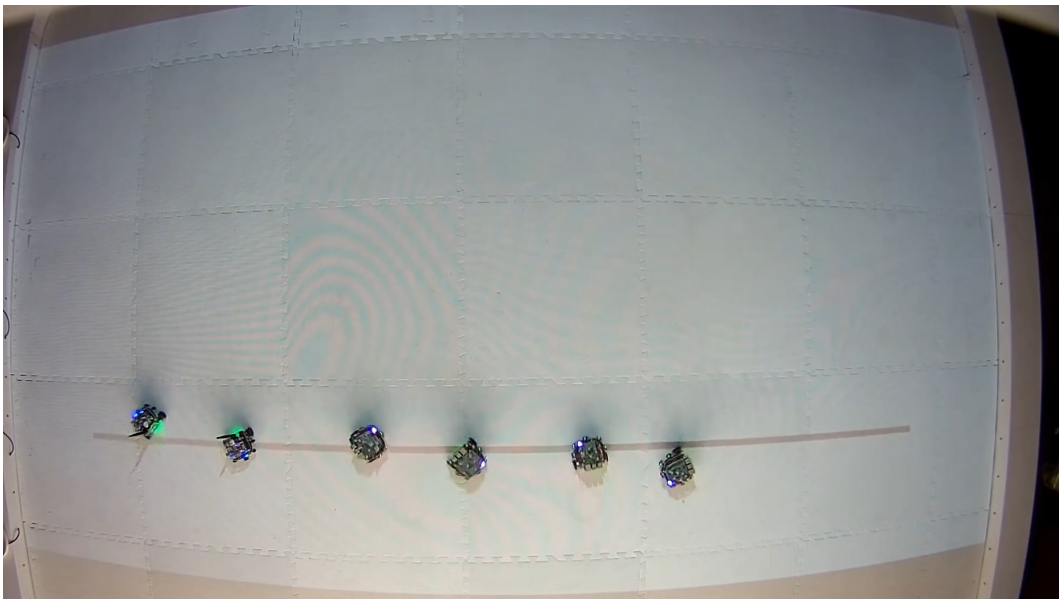
In the end, the robots end up standing still in their final position forming a line and with a regular separation between them Figure 3.6c. Here it has been observed that the time gap given to the radio is quite important, since the radio is relied upon

to calculate the position to which they are heading. Times of one second have led to robots taking much longer to stabilise than lower times.

The experiment was completed successfully, with the expected result and no collisions, and was sent to the robotarium. Here, no differences in behaviour could be observed when compared with the simulator. A total of ten sessions have been successfully carried out in the robotarium with variable convergence times that are due to the random initial position of the robots.



(a)



(b)

Figure 3.7: Performance on the Robotarium. a) Robots moving towards its position after starting in random positions. b) Final line formation along the line on the lab. The video can be found at https://youtu.be/vDBZq_OZdz4.

3.3 Coverage using voronoi diagrams

3.3.1 Theoretical background

A voronoi diagram divides a space into regions based on proximity to a given set of points. Each region contains all the points in the space that are closer to a specific point (called the generator) than to any other.

Voronoi diagrams are a useful tool for efficiently distributing agents over a terrain with the goal of covering the entire area equally. When using Voronoi diagrams, each agent is associated with a region of the terrain that will contain all points closer to that agent than to any other. This ensures that the space is divided into well-defined parts, and each agent has the responsibility to cover a specific area, ensuring that there are no unattended areas or unnecessary overlaps.

In addition, this proximity-based division allows agents to move autonomously to adjust their positions and improve ground coverage, especially when conditions change or denser coverage is needed in certain areas. By using decentralised control, each robot only needs to know its own region and the location of the others to optimise its position, without the need for centralised coordination.

The Voronoi diagram is described in the *Coverage Control for Mobile Sensing Networks* [3] as:

$$V_i = \{q \in \mathcal{Q} \mid \|q - p_i\| \leq \|q - p_j\|, \forall j \neq i\} \tag{3.8}$$

Where:

- V_i is is the Voronoi region associated with point p_i. In other words, it is the set of all points q in space that are closer to q_i than to any other point q_j in the set.
- $p \in \mathcal{Q}$ p is any point within the space $\mathcal{Q}$ on which the work is being carried out.
- $\|q-p_i\|$ is the distance between the point q and the generator p_i, which in practice is calculated using the Euclidean distance.
- $\|q - p_j\|$ is the same as $\|q - p_i\|$. The formula looks for the q point to be closer to p_i than to any other q_j
- $\forall j \neq i$ is the condition that q must be closer to p_i than to any other generator.

In the experiment, there will be obstacles on the ground that need to be taken into account in order to divide the regions. One strategy for approaching the obstacles is to assume that the obstacle is another agent in the field. Each agent that detects an obstacle will interpret it as a virtual agent located at twice the distance in the direction of the point closest to the obstacle, Figure 3.9. In this way, all points that

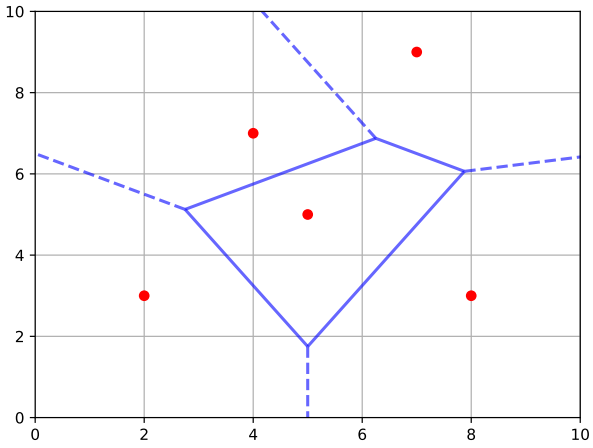


Figure 3.8: Example of voronoi diagram. Each region contains all the points that are closer to each agent (red dots).

would correspond to the obstacle will not be taken into account for the voronoi region of the agent since they belong to the virtual agent by proximity.

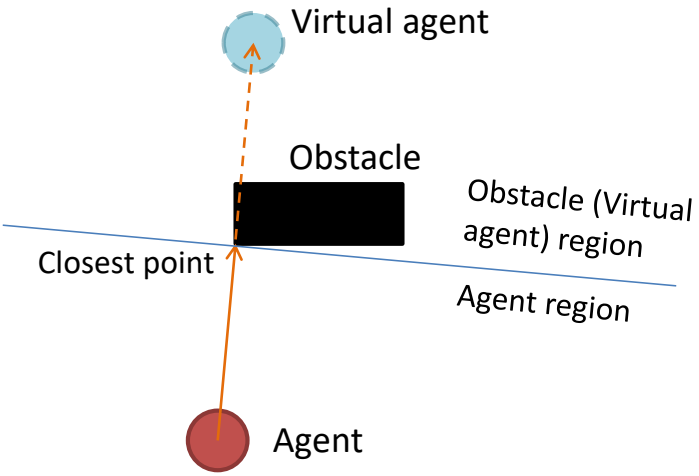


Figure 3.9: Example of obstacle detection. A virtual agent is projected symmetrically to the real agent from the point closest to the obstacle and is taken into account when dividing the regions.

3.3.2 Implementation over DREAM

This experiment will consist of distributing the robots evenly over the terrain using Voronoi regions. Each of the robots will be an agent (generator) that will move around the terrain positioning itself in such a way that its Voronoi region coverage area is as large as possible. A final equilibrium state should be reached in which all robots are

stationary because they are already covering the maximum possible terrain capacity between them.

The following classes have been used in the code:

- **Radio**: The robots will communicate with each other via DREAM's range-operated radio and the information they will send to each other will be their positions. In this way, the robots will know the position of all nearby robots in order to calculate the Voronoi region.
- **Sensor**: They will use two types of sensors each. DREAM's odometry sensor to know their position and a new sensor designed for this experiment that will detect obstacles in the terrain.
- **Obstacle**: The obstacle class is also created as a terrain element. This class will be in charge of representing areas that robots cannot pass through because it would be considered a collision and are to be detected by that new sensor.
- **Robot**: The robot programmed for this experiment will inherit directly from the main class, so it will not use HRVO. The Voronoi algorithm should be robust enough not to rely on another system for collision avoidance.

Algorithm 1 Voronoids coverage procedure

for each Robot r_i **do**:
  Draw a circle of radius 1 around r_i, with r_i as the center.
  Check for obstacles and other robots (r_j)
  Remove from the circle the points q such that $\|q - r_j\| < \|q - r_i\|$ for $r_j \neq r_i$.
  Compute the centroid C_i of the remaining region.
  Set C_i as new goal point for r_i
  Compute r_i movement towards C_i
end for

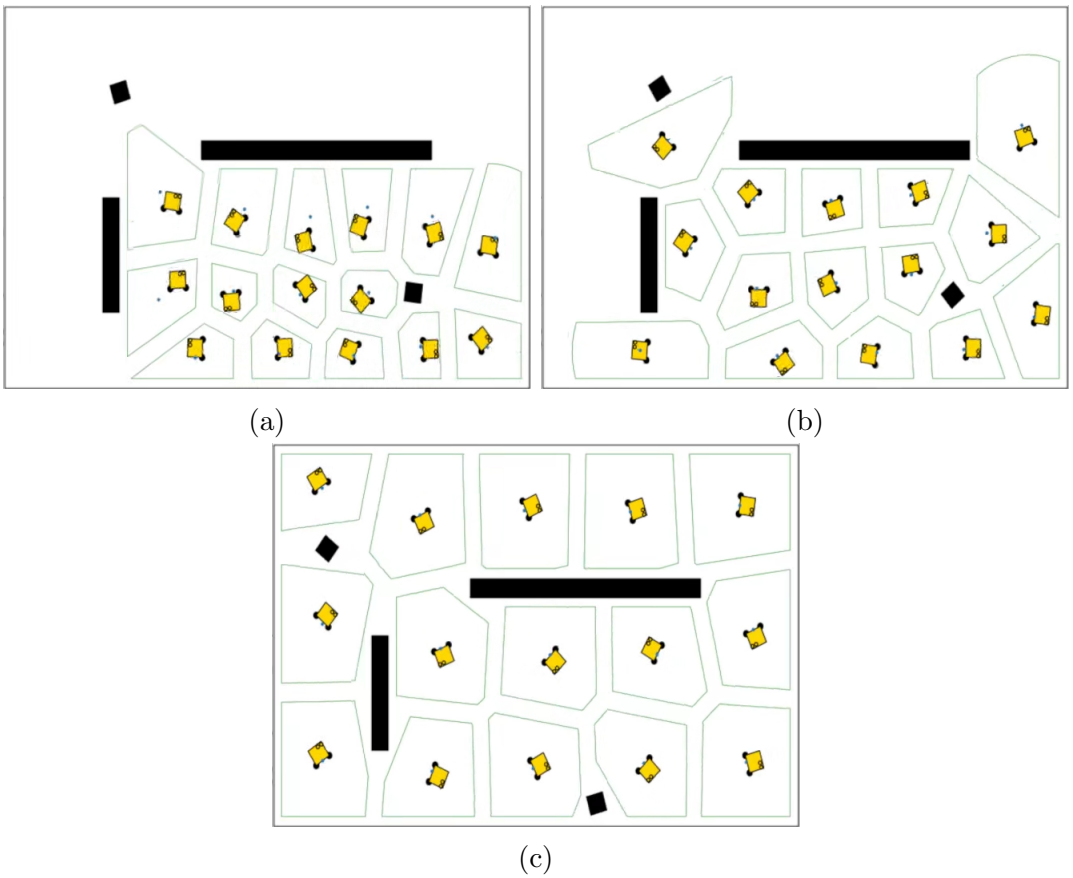
Goals of the experiment:

- Perform a balanced terrain distribution using voronoi diagrams as a coverage technique.
- Test creating a new inherited sensor class that perform different functions than the already created in DREAM.
- Test an algorithm that itself also serves as a collision avoidance system.

3.3.3 Experimental results

In this experiment the robots will always start in a fixed position, in the bottom right corner Figure 3.10a. There will be a total of fifteen robots that will have to spread out across the terrain. In addition to avoiding collisions with each other, they must also avoid the obstacles on the ground, which are represented by black blocks. There are two large blocks that are fixed obstacles, i.e. they do not move, and two smaller blocks that are moving obstacles (they move in random directions).

The robots furthest from the bottom right corner start to move quickly, as when calculating the voronoi regions, the centroid is completely displaced given the large amount of free space in the opposite direction to the large crowd of robots behind it as seen in Figure 3.10b.



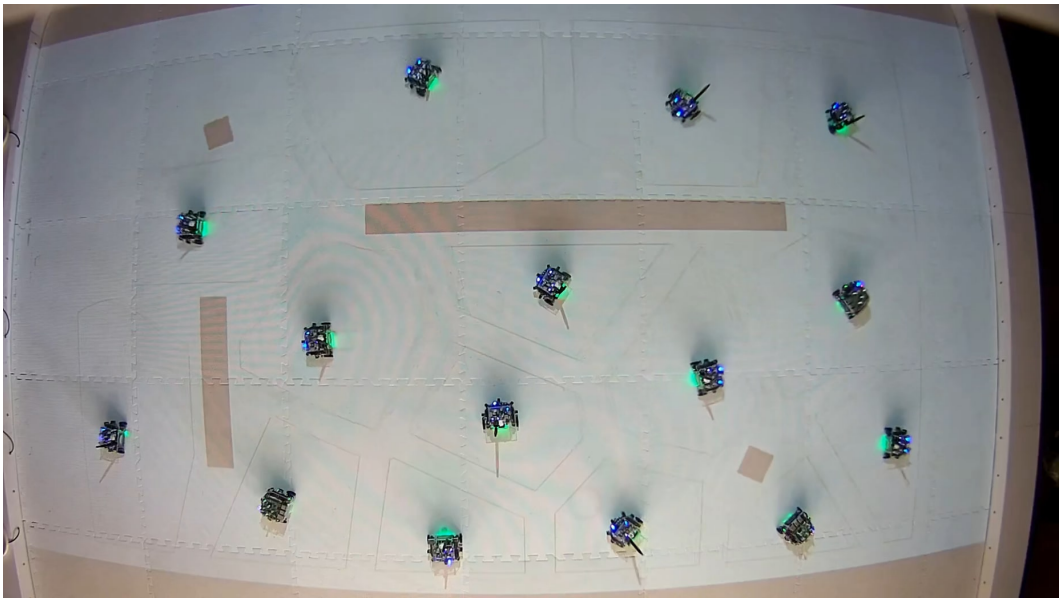
(a) (b)

(c)

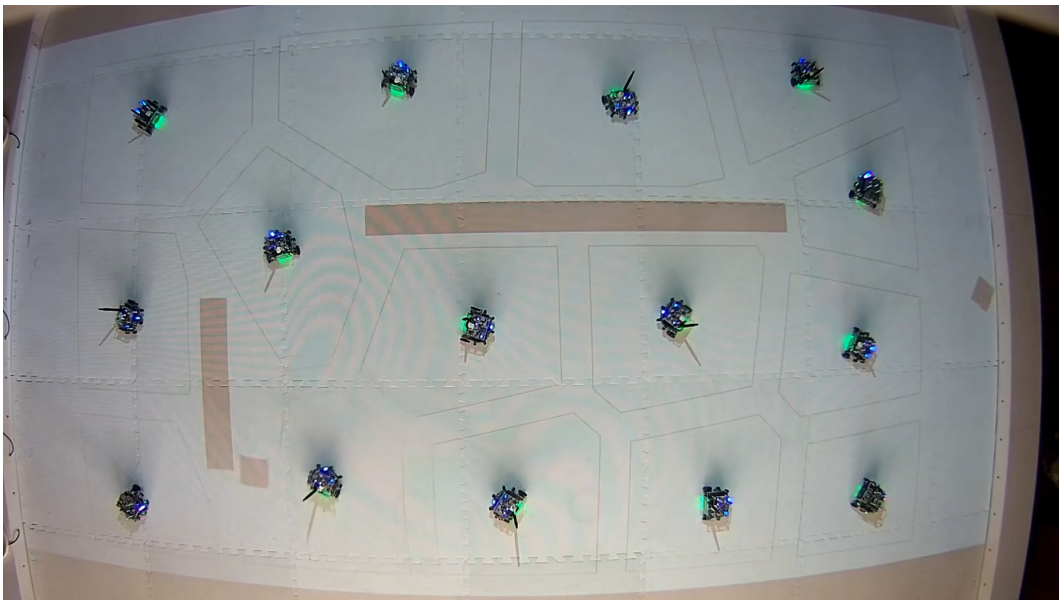
Figure 3.10: Performance on the simulator. a) Voronoid starting position. All robots positioned towards the bottom right corner. b) Start of movement. External robots move faster as the Voronoi region grows towards an empty space. c) Final equilibrium pose. The robots remain spaced out, covering the entire terrain. The simulation can be found at https://youtu.be/u8JXHs6fdwQ.

In the end, the experiment never reaches that equilibrium position due to the moving obstacles that force the robots next to them to be in constant motion due to the

alteration of their voronoi region, causing their neighbouring robots to move to a lesser extent as well. Figure 3.10c



(a)



(b)

Figure 3.11: Performance on the Robotarium. a) Robotarium voronoi distribution. Robots are moving and spreading out on the scene. b) Final voronoi equilibrium pose. The distribution is different as in the simulator due to physical limitations of the robots. The video can be found at https://youtu.be/9kjvGFiEnP8.

After obtaining positive results in the simulator, the experiment was sent to the Robotarium. In this case, the results obtained were slightly different to those seen in the simulator. The first is that the distance at which a robot is considered to have reached its destination had to be modified. This distance has had to be reduced from 0.05 to

0.02 because the laboratory is not as accurate as the simulator. With a distance of 0.05, the robots barely moved forward a little before coming to a standstill, considering the initial position as the equilibrium position. This is because the recalculation of the voronoi region did not deviate the centroid too much, so the robot already considered that it had reached its destination and therefore did not move. By not moving, the calculated voronoi region remained static and the centroid remained in its position.

The other different factor observed is the final equilibrium position. If moving obstacles that can change this result are not taken into account, the robots in the simulator end up in a position that resembles three horizontal rows of five robots each. In this case, this distribution is not completed due also to the tolerance value of the end position distance Figure 3.11b. This distance can be reduced even further, achieving the same simulator layout, but with even smaller values the robots never stay still and end up in perpetual motion on the site trying to adjust to that position.

3.4 Affine Formation Maneuver Control

3.4.1 Theoretical background

In this final experiment, the state-of-the-art algorithm from *Affine Formation Manoeuvrer Control of Multiagent Systems* [18] is tested, addressing formation control in multi-agent systems, focusing on two main tasks:

- Formation of a desired geometric pattern: Guiding agents to adopt a specific configuration.
- Collective manoeuvres: Allowing the formation to dynamically change parameters such as centroid, orientation and scale.

To achieve these goals simultaneously, it proposes a novel approach based on stress matrices, which are generalisations of Laplacian matrices of graphs with positive, negative and zero edge weights. This method allows agents to follow a target formation that is an affine transformation (including translations, rotations and scalings) of a nominal configuration. Moreover, the proposed control is distributed: only a small number of agents, called leaders, know the desired manoeuvres, while the rest, the followers, adjust their behaviour accordingly.

To satisfy this, the following control law is used for the followers:

$$\dot{\hat{p}}_i(t) = -k \sum_{j \in \mathcal{N}_i} w_{ij} \big(\hat{p}_i(t) - \hat{p}_j(t)\big) \tag{3.9}$$

Where:

- $\hat{p}_i(t), \hat{p}_j(t)$: Position x, y of robot i, j at time t.

- $\mathcal{N}_i$: Neighbours of node i in the network (robots with which i can communicate with)
- w_{ij}: Weight of the connection between the robots i, j.
- k: Multiplicative constant to control the strength of the stress.

3.4.2 Implementation over DREAM

This experiment will consist of the control of formation by follower robots and movements with leader robots that force the followers to adapt to the new formation. For this purpose, a structure of seven robots will be used, with three of them as leaders and the remaining four as followers. The distribution has been extracted from *Affine Formation Maneuver Control of Multiagent Systems* [18] as well as the weights needed to maintain that formation. It is going to be a formation with the three front robots as leaders represented in the Figure 3.12 with the colour orange. These will be in charge of performing the formation control calculations without taking into account the followers marked in green.

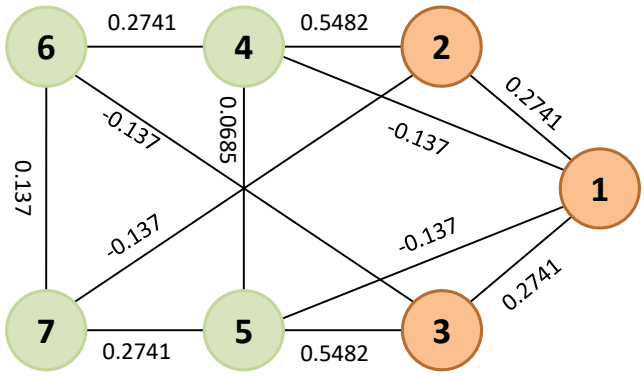


Figure 3.12: Formation to be used in the simulation. The orange dots represent the leaders, while the green dots are the followers. The lines represent the connections between robots and their weights to maintain stress equilibrium. Weight values are obtained from [18]'s chapter VII.

The sequence will be as follows. The leaders will stand still in position and the followers will start at the corners of the field. The followers should move towards their positions in the formation. Then, black bars will appear in a horizontal sweep to simulate obstacles. The leaders will try to avoid them by deforming the structure with the intention that the followers adapt to the new formation and avoid the obstacles as

well.

The following classes have been used in the code:

- **Radio**: There will be a different type of radio for each type of robot inherited from DREAM's main radio. Leaders will have a radio that is responsible for publishing their current position, but they do not receive that kind of information back. Followers will have a different radio that will publish their position and receive their position based on connections with other bots.

- **Sensor**: They will use two types of sensors each. DREAM's odometry sensor to know their own position and DREAM's movement sensor to check all nearby robots.

- **Obstacle**: This extra class is responsible for placing visible obstacles in the terrain to be avoided by the formation. The obstacles are rectangle-shaped and sweep from right to left.

- **Robot**: Here there will be two distinct robot classes that will inherit from DREAM's robot_hrvo class. One class will be the class for the leaders that will be in charge of maintaining the formation and dodging obstacles, and the other class will be the class for the followers that will be in charge of calculating their position based on the control law 3.9.

Moreover, (3.9) was implemented as its direct forward Euler discretization with a Δt nominally given by the DREAM time gap of 33 milliseconds, which can be configured different between agents.

Goals of the experiment:

- Make the formation not only rigid, but also dynamic, capable of performing related transformations such as translations, rotations and scaling in response to planned manoeuvres or changes in the environment.

- Manage several different types of robots and radios at the same time in the same experiment.

The robots shall be communicated as seen in the Figure 3.12, following the following weight matrix:

$$w = \begin{bmatrix} 0 & 0.2741 & 0.2741 & -0.137 & -0.137 & 0 & 0 \\ 0.2741 & 0 & 0 & 0.5482 & 0 & 0 & -0.137 \\ 0.2741 & 0 & 0 & 0 & 0.5482 & -0.137 & 0 \\ -0.137 & 0.5482 & 0 & 0 & 0.0685 & 0.2741 & 0 \\ -0.137 & 0 & 0.5482 & 0.0685 & 0 & 0 & 0.2741 \\ 0 & 0 & -0.137 & 0.2741 & 0 & 0 & 0.137 \\ 0 & -0.137 & 0 & 0 & 0.2741 & 0.137 & 0 \end{bmatrix}$$

3.4.3 Experimental results

When the simulation starts, the three leaders appear in the centre of the field in the shape of a triangle while the followers are in the corners, Figure 3.13a. The leaders remain completely static while the followers advance to their calculated positions, Figure 3.13b. It is the task of the followers not to bump into the leaders on their way to their position as the leaders will not take them into account at all.

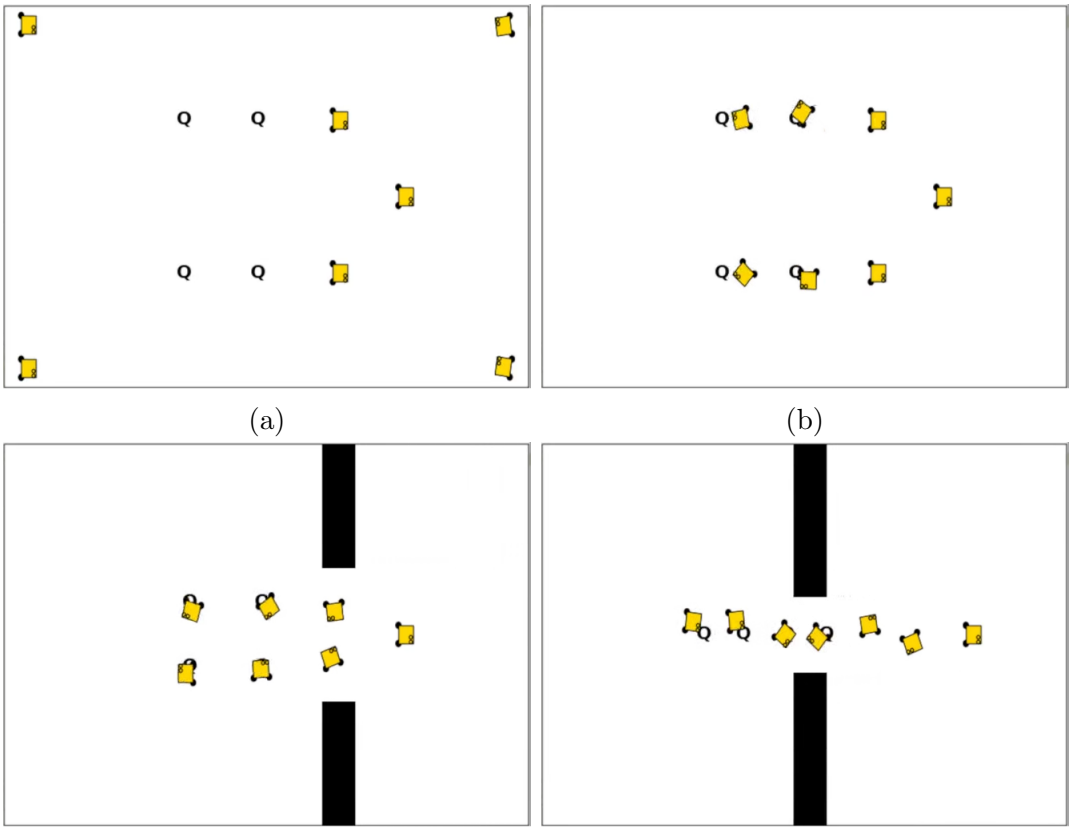
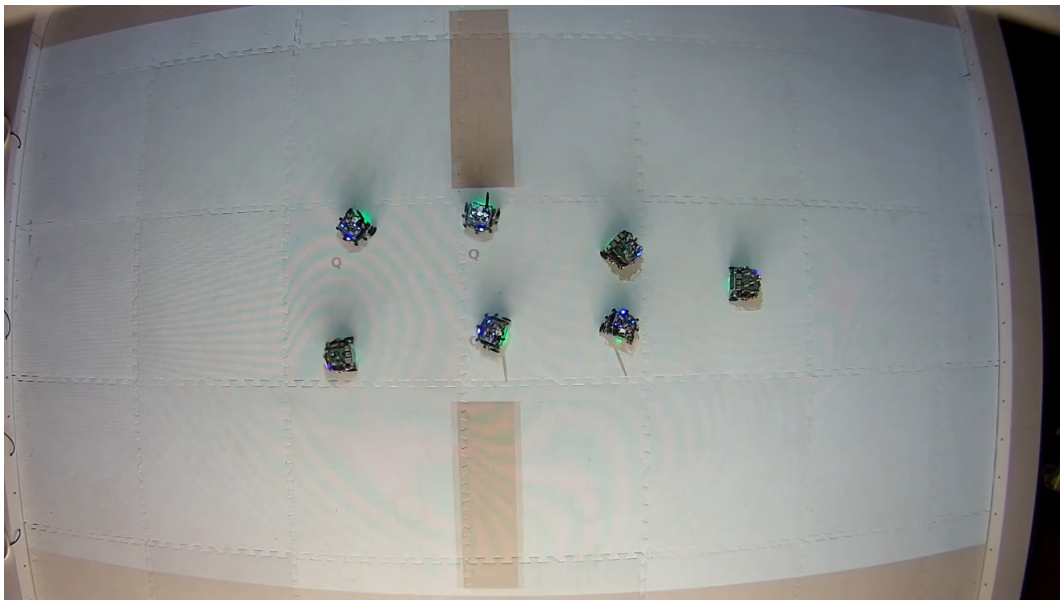


Figure 3.13: Performance on the simulator. a) Starting position. The three leaders stand in their triangular formation while the followers start in the corners. Marks on the ground mark the position where the followers should go. b) Los robots han avanzado desde las esquinas hasta sus posiciones calculadas. c) Formation flattened to avoid the first obstacle. d) Formation in fully stretched form to avoid the second obstacle. The simulation can be found at https://youtu.be/HrFs4DRcx7s.

Once they have reached their positions, an obstacle appears on the right side. The obstacle consists of two black rectangles with a gap in the middle to pass through. The leaders in order to avoid it, come together causing the entire formation to tighten enough for the obstacle to pass without touching any of the robots. The followers recalculate their new positions based on the leaders' new positions and successfully avoid the obstacle, Figure 3.13c.



(a)

(b)

Figure 3.14: Performance on the Robotarium. a) Robots avoiding the first obstacle by forming a compressed formation, like in the simulator. b) Final formation of the robots after avoiding both sets of obstacles. The formation remains as shown in Figure 3.12. The video can be found at https://youtu.be/l-g1QARxXYU.

The robots return to their original positions and then a second obstacle appears. In this case, the gap between the two rectangles is much smaller, so the leaders align themselves, causing the followers to adapt as well, forming a line that passes precisely through the gap without causing any collision with the obstacle Figure 3.13d.

After obtaining positive results in the sequence, the test was conducted on the Robotarium. The results with the real robots were successful, though slightly worse than in the simulator. The robots found it more challenging to reach their intended positions, and it was necessary to adjust the constant k from the control law differently from how it was adjusted in the simulation to achieve a positive result. The value had to be decreased because a higher value leads to more aggressive dynamics, which, while potentially converging faster, can cause oscillations around the desired position. A lower value results in slower convergence but also greater robustness.

Chapter 4

Conclusions and future work

This project has proposed DREAM, a library designed for the execution of distributed multi-robot algorithms on virtual robotics simulators. In order to test its effectiveness, it has been tested on GeorgiaTech's Robotarium simulator, since in addition to providing a Python simulator, it also gave the opportunity to test it remotely on real robots in its own laboratory.

The division of the robot into different parts that can be controlled independently allows simulations to be performed with great control over all the elements separately thanks to the Timer, managing the activation times of each one and allowing easy communication between them through the system of callback methods.

A total of four experiments have been developed using DREAM that have touched on different multi-robot strategies, demonstrating that the library is flexible and adaptable to any type of application need and even the inclusion of completely new classes if required. The simulated experiments in the Robotarium, with favourable results for both the simulator and the robots in the laboratory, have served both to compare the possible differences in behaviour between the simulator and the physical world and to show the compatibility of DREAM with real robots.

From this project, several areas for improvement and future exploration were identified. The possibility of extending the library by creating subclasses from existing ones to add more specific functionalities is possible, but these new subclasses will be more dependent on the characteristics of each simulator where they will be used, such as the incorporation of acceleration or gravity sensors for adapted drone simulations, the addition of multiple actuators to add robotic arms, or cameras for computer vision.

Bibliography

[1] N. AbuJabal et al. "Path Planning Techniques for Real-Time Multi-Robot Systems: A Systematic Review". In: *Electronics* 13.12 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13122239.

[2] C.E. Aguero et al. "Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response". In: *Automation Science and Engineering, IEEE Transactions on* 12.2 (Apr. 2015), pp. 494–506. ISSN: 1545-5955. DOI: 10.1109/TASE.2014.2368997.

[3] J. Cortes et al. "Coverage control for mobile sensing networks". In: *IEEE Transactions on Robotics and Automation* 20.2 (2004), pp. 243–255. DOI: 10.1109/TRA.2004.824698.

[4] Marco Dorigo, Guy Theraulaz, and Vito Trianni. "Reflections on the future of swarm robotics". In: *Science Robotics* 5.49 (2020), eabe4385. DOI: 10.1126/scirobotics.abe4385.

[5] A. S. Morse J. Lin and B. D. O. Anderson. "The multi-agent rendezvous problem". In: *Proc. 42nd IEEE Conf. Decision and Control* (Dec. 2003), pp. 1508–1513.

[6] A.C. Jiménez, V. García-Díaz, and S. Bolaños. "A Decentralized Framework for Multi-Agent Robotic Systems". In: *Sensors* 18.2 (2018). DOI: 10.3390/s18020417.

[7] Nathan K. and Andrew H. "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, Sept. 2004, pp. 2149–2154.

[8] I. Mas and C. Kitts. "Centralized and decentralized multi-robot control methods using the cluster space control framework". In: *2010 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. 2010, pp. 115–122. DOI: 10.1109/AIM.2010.5695768.

[9] O. Michel. "Webots: Professional Mobile Robot Simulation". In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42.

[10] E.s Mohamed et al. "Smart farming for improving agricultural management". In: *The Egyptian Journal of Remote Sensing and Space Science* 24 (Sept. 2021). DOI: 10.1016/j.ejrs.2021.08.007.

[11] R. Olfati-Saber, J.A. Fax, and R.M. Murray. "Consensus and Cooperation in Networked Multi-Agent Systems". In: *Proceedings of the IEEE* 95.1 (2007), pp. 215–233. DOI: 10.1109/JPROC.2006.887293.

[12] C. Pinciroli et al. “ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics”. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2011), pp. 5027–5034. DOI: 10.1109/IROS.2011.6094829.

[13] C. Pinciroli et al. “ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems”. In: *Swarm Intelligence* 6 (Dec. 2012), pp. 271–295. DOI: 10.1007/s11721-012-0072-5.

[14] J. Snape et al. “The Hybrid Reciprocal Velocity Obstacle”. In: *IEEE Transactions on Robotics* 27.4 (2011), pp. 696–706. DOI: 10.1109/TRO.2011.2120810.

[15] S. Wilson and M. Egerstedt. “The Robotarium: A Remotely-Accessible, Multi-Robot Testbed for Control Research and Education”. In: *IEEE Open Journal of Control Systems* PP (Jan. 2022), pp. 1–13. DOI: 10.1109/OJCSYS.2022.3231523.

[16] Sean Wilson. *Robotarium Pyhton Simulator Github Repository*. October 2023. URL: https://github.com/robotarium/robotarium_python_simulator/tree/134a1f769b082bf0a8cd84c9138886e80e7c4e90 (visited on 10/15/2024).

[17] Zheng Zhang et al. “Collision-Free Route Planning for Multiple AGVs in an Automated Warehouse Based on Collision Classification”. In: *IEEE Access* 6 (2018), pp. 26022–26035. DOI: 10.1109/ACCESS.2018.2819199.

[18] Shiyu Zhao. “Affine Formation Maneuver Control of Multiagent Systems”. In: *IEEE Transactions on Automatic Control* 63.12 (2018), pp. 4140–4155. DOI: 10.1109/TAC.2018.2798805.

[19] Xin Zhou et al. “Swarm of micro flying robots in the wild”. In: *Science Robotics* 7.66 (2022), eabm5954. DOI: 10.1126/scirobotics.abm5954.

Appendices

Appendix A

DREAM's source code

This appendix contains all the source code for the library used, DREAM. It consists of five scripts.

- **dream.py** contains the Robot class.
- **dream_sensor.py** contains the Sensor class.
- **dream_radio.py** which contains the Radio class.
- **dream_timer.py** which contains the Timer class.
- **dream_specs.py** which serves as a configurable file to easily modify the parameters of the robots and the terrain.

A.1 dream.py

```
import numpy as np

from dream_sensor import *
from dream_radio import *
from dream_timer import *
from dream_specs import *


class Robot:
    def __init__ (self, robot_id, tag = 'robot', vel_gain =
    ↪ 0.8, ang_gain = 3.0):
        """
        Main robot class.

        robot_id : int   -> id of robot
        tag      : str   -> name tag of the robot
        vel_gain : float -> velocity gain
        ang_gain : float -> angular gain

        """
        self.robot_id = robot_id
```

```
        self.tag = tag

        self.pos_x = 0
        self.pos_y = 0
        self.rot_w = 0

        self.vel_gain = vel_gain
        self.ang_gain = ang_gain

        # Robot velocities and dynamics
        self.dxi = np.array((0.0, 0.0))
        self.dxu = np.array((0.0, 0.0))


        # Real robot specs
        self.v_max = R_V_MAX        # max linear velocity
        self.w_max = R_W_MAX        # max angular velocity
        self.ac_max = R_AC_MAX      # max actuator velocity
        self.r_wheel = R_R_WHEEL    # robot wheel radius
        self.r_length = R_R_LENGTH # robot length

        self.rr = 1 / self.r_wheel
        self.ll = self.r_length / (2 * self.r_wheel)
        self.rr_ = self.ac_max / self.rr
        self.ll_ = self.ll / self.rr


    def actuator_limit(self):
        """
        Call this function to reduce the linear velocity and
         ↪ avoid exceeding the actuator limits.
        Angular velocity is maintained.

        """

        self.v_ = max(-self.v_max, min(self.v_max, self.dxu[0])
         ↪ )
        self.w_ = max(-self.w_max, min(self.w_max, self.dxu[1])
         ↪ )
        if self.rr * abs(self.v_) + self.ll * abs(self.w_) >
         ↪ self.ac_max:
            self.v_ = (self.rr_ - self.ll_ * abs(self.w_)) * np
             ↪ .sign(self.dxu[0])
        self.dxu[0] = self.v_
        self.dxu[1] = self.w_


    def dxi_to_dxu(self):
        """
        Transform self.dxi velocities into self.dxu dynamics
        """
```

```
        a = np.cos(self.rot_w)
        b = np.sin(self.rot_w)

        self.dxu[0] = self.vel_gain*(a * self.dxi[0] + b * self
         ↪ .dxi[1])
        self.dxu[1] = self.ang_gain*np.arctan2(-b * self.dxi[0]
         ↪  + a * self.dxi[1], self.dxu[0])/(np.pi/2)



class Robot_hrvo(Robot):
    def __init__ (self, robot_id, tag, vel_gain = 0.8, ang_gain
     ↪  = 3.0, safety = 0.24, check_boundaries = True):
        """
        Robot class with HRVO function to be used to avoid
         ↪ other robots

        robot_id         : int   -> id of robot
        tag              : str   -> name tag of the robot
        vel_gain         : float -> velocity gain
        ang_gain         : float -> angular gain
        safety           : float -> safety radius of the robots
        check_boundaries : bool  -> check limits of the
         ↪ environment

        """

        self.robot_id = robot_id
        self.tag = tag
        self.safety = safety #if safety > 0.24 else 0.24  #
         ↪ safety radius

        self.detections = [[30, 0, 0, 0]]

        self.check_boundaries = check_boundaries
        self.bound_x = [T_MIN_WIDTH, T_MAX_WIDTH]
        self.bound_y = [T_MIN_HEIGHT, T_MAX_HEIGHT]

        super().__init__(self.robot_id, self.tag, vel_gain,
         ↪ ang_gain)


    def HRVO(self, detections, desired_velocity):
        """
        Apply HRVO to the current desired velocity in order to
         ↪ modify it's path to avoid other robots

        Args:
            detections       : np.array[float, float, float,
             ↪ float] ->
```

```
            Nx4 array as N = number of detected robots. If
             ↪ A is this robot and B the detected robot:
            [Distance from A to B, angle from A to B, B
             ↪ x_velocity, B y_velocity]

        desired_velocity : np.array[float, float]
         ↪                 ->
            1x2 array with the desired velocity [
             ↪ desired_x_velocity, desired_y_velocity]

    Return:
        None, the super().dxi parameter is updated with the
         ↪  resulting value
    """

    detections = np.asarray(detections)
    n_detections = len(detections)
    hrvo_mat = np.empty((n_detections * 2, 2))
    hrvo_vec = np.empty((n_detections * 2))

    for i in range(n_detections):

        obst_dist = detections[i, 0]
        obst_angl = detections[i, 1]
        obst_velx = detections[i, 2]
        obst_vely = detections[i, 3]

        obst_dist = obst_dist if obst_dist > self.safety
         ↪ else self.safety
        obst_phi = np.arcsin(self.safety / obst_dist)
        obst_phi_l = obst_angl + obst_phi
        obst_phi_r = obst_angl - obst_phi


        # Create VO triangle
        temp_mat_ = np.array(([1., 0., 0.], [0., 1., 0.],
         ↪ [-obst_velx, -obst_vely, 1.]))

        temp_lin_l = np.cross(np.array([0, 0, 1]), np.array
         ↪ ([np.cos(obst_phi_l), np.sin(obst_phi_l)]))
        temp_lin_l = temp_mat_ @ temp_lin_l

        hrvo_mat[i*2, :] = -temp_lin_l[:2]
        hrvo_vec[i*2] = temp_lin_l[2]

        temp_lin_r = np.cross(np.array([0, 0, 1]), np.array
         ↪ ([np.cos(obst_phi_r), np.sin(obst_phi_r)]))
        temp_lin_r = temp_mat_ @ temp_lin_r

        hrvo_mat[i*2 + 1, :] = temp_lin_r[:2]
        hrvo_vec[i*2 + 1] = -temp_lin_r[2]
```

```
            # RVO
            temp_mat_ = np.array(([1., 0., 0.], [0., 1., 0.],
             ↪ [-(obst_velx+self.dxi[0])/2, -(obst_vely+self.
             ↪ dxi[1])/2, 1.]))

            # HRVO
            temp_lin_l = np.cross(np.array([0, 0, 1]), np.array
             ↪ ([np.cos(obst_phi_l), np.sin(obst_phi_l)]))
            temp_lin_l = temp_mat_ @ temp_lin_l
            dist_l = abs(temp_lin_l[0]*self.dxi[0]+temp_lin_l
             ↪ [1]*self.dxi[1]+temp_lin_l[2])/np.sqrt(
             ↪ temp_lin_l[0]**2 + temp_lin_l[1]**2)

            temp_lin_r = np.cross(np.array([0, 0, 1]), np.array
             ↪ ([np.cos(obst_phi_r), np.sin(obst_phi_r)]))
            temp_lin_r = temp_mat_ @ temp_lin_r
            dist_r = abs(temp_lin_r[0]*self.dxi[0]+temp_lin_r
             ↪ [1]*self.dxi[1]+temp_lin_r[2])/np.sqrt(
             ↪ temp_lin_r[0]**2 + temp_lin_r[1]**2)

            if dist_l < dist_r:
                hrvo_mat[i*2, :] = -temp_lin_l[:2]
                hrvo_vec[i*2] = temp_lin_l[2]

            if dist_l > dist_r:
                hrvo_mat[i*2 + 1, :] = temp_lin_r[:2]
                hrvo_vec[i*2 + 1] = -temp_lin_r[2]


        # Search for valid velocities
        th = np.linspace(0, 2 * np.pi, 20)
        vel = np.linspace(0.0, self.v_max, 10)

        vv, thth = np.meshgrid(vel, th)

        vx_sample = np.append((vv * np.cos(thth)).flatten(), 0)
        vy_sample = np.append((vv * np.sin(thth)).flatten(), 0)

        v_sample = np.stack((vx_sample, vy_sample))


        for i in range(n_detections):
            hrvo_mat_ = hrvo_mat[2*i:2*i+2, :]
            hrvo_vec_ = hrvo_vec[2*i:2*i+2]

            v_out = []
            for i in range(np.shape(v_sample)[1]):
                if not ((hrvo_mat_ @ v_sample[:, i] < hrvo_vec_
                 ↪ ).all()):
                    v_out.append(v_sample[:, i])
```

```
            v_sample = np.array(v_out).T

            if np.shape(v_sample)[0] == 0:
                v_sample=np.array(([0.0], [0.0]))


        if self.check_boundaries:

            v_out = []
            for i in range(np.shape(v_sample)[1]):
                if self.bound_x[0] < v_sample[0, i] + self.
                  ↪ pos_x < self.bound_x[1] and self.bound_y[0]
                  ↪  < v_sample[1, i] + self.pos_y < self.
                  ↪ bound_y[1]:
                    v_out.append(v_sample[:, i])
            v_sample = np.array(v_out).T

            if np.shape(v_sample)[0] == 0:
                v_sample=np.array(([0.0], [0.0]))


        size = np.shape(v_sample)[1]
        diffs = v_sample - ((desired_velocity).reshape(2, 1) @
          ↪ np.ones(size).reshape(1, size))
        norm = np.linalg.norm(diffs, axis=0)
        min_index = np.where(norm == np.amin(norm))[0][0]
        self.dxi = (v_sample[:, min_index])

        limit_norm = np.linalg.norm(self.dxi)
        if limit_norm > self.v_max:
            self.dxi *= self.v_max/limit_norm



if __name__ == "__main__":
    print("DREAM Robot class")
```

A.2 dream_sensor.py

```
import numpy as np

class Sensor:
    def __init__(self, robot_id, tag = 'Sensor'):
        """
        Sensor base class for a robot

        robot_id : int -> id of robot which it is attached
        tag      : str -> name tag of the sensor

        """

        self.robot_id = robot_id
        self.tag = tag

        self.pos_x = 0
        self.pos_y = 0
        self.rot_w = 0

        self.ret = 0   # return value


    def add_noise(self, values, noise):
        """
        Add random noise to the values measured

        Args:
            values : (np.array, list) -> values to be altered
            noise  : float            -> float value

        Return:
            np.array -> Same values altered by a random normal
             ↪ distribution
        """
        return np.add(values, np.random.normal(0, noise, np.
         ↪ shape(values)))


    def update(self, x):
        """
        Get current position of the sensor

        Args:
            x : np.array -> Positions of all robots
        """
        self.pos_x = x[0, self.robot_id]
        self.pos_y = x[1, self.robot_id]
        self.rot_w = x[2, self.robot_id]
```

```
class Movement_sensor(Sensor):
    def __init__(self, robot_id, tag, callback, range = 1.0,
     ↪ direction = 0.0, wide = 2*np.pi, noise = 0.0):
        """
        Movement sensor gets current position and velocity of
         ↪ all robots that can be detected

        robot_id  : int      -> id of robot which it is
         ↪ attached
        tag       : str      -> name tag of the sensor
        callback  : callable -> callback function
        range     : float    -> maximum distance of detection
        direction : float    -> counter-clockwise angle offset
         ↪ of the sensor respect the robot front
        wide      : float    -> aperture angle of the sensor
        noise     : float    -> noise of the sensor
        """

        self.callback = callback   # callback function

        self.range = range         # sensor maximum range
        self.direction = direction # sensor pointing direction
        self.wide = wide / 2       # sensor aperture angle
        self.noise = noise         # sensor noise

        super().__init__(robot_id, tag)


    def update(self, x, v):
        """
        Get position and velocity of all other robot entities
         ↪ in range

        Args:
            x : np.array -> 3xN positions array of all robots
            v : np.array -> 2xN dynamics array of all robots
             ↪ from previous loop

        """

        super().update(x)

        self.ret = []

        for i in range(np.shape(x)[1]):

            dist = np.sqrt((self.pos_x - x[0, i]) ** 2 + (self.
             ↪ pos_y - x[1, i]) ** 2)
```

```
            angle = np.arctan2(self.pos_y - x[1, i], self.pos_x
             ↪  - x[0, i])

            angle_s = angle - (self.direction + self.rot_w) -
             ↪ np.pi
            angle_s = (angle_s + np.pi) % (2 * np.pi) - np.pi


            if dist > 0 and dist <= self.range and abs(angle_s)
             ↪  <= self.wide:

                cs = np.cos(x[2, i])
                ss = np.sin(x[2, i])

                vx = (cs*v[0, i] - 0.05*ss*v[1, i])
                vy = (ss*v[0, i] + 0.05*cs*v[1, i])

                self.ret.append([dist, angle, vx, vy])

        self.ret = self.add_noise(self.ret, self.noise)
        self.callback(self.ret)


class Odometry_sensor(Sensor):
    def __init__(self, robot_id, tag, callback, noise = 0.0):
        """
        Odometry sensor measures the current position of the
         ↪ attached robot

        robot_id  : int      -> id of robot which it is
         ↪ attached
        tag       : str      -> name tag of the sensor
        callback  : callable -> callback function
        noise     : float    -> noise of the sensor

        """

        self.callback = callback
        self.noise = noise


        super().__init__(robot_id, tag)


    def update(self, x):
        """
        Get current position of the robot attached

        Args:
            x : np.array-> 3xN positions array of all robot
             ↪ poses
```

```
        """

        super().update(x)

        self.ret = [self.pos_x, self.pos_y, self.rot_w]
        self.ret = self.add_noise(self.ret, self.noise)
        self.callback(self.ret)


if __name__ =="__main__":
    print("DREAM Sensor class")
```

A.3 dream_radio.py

```
import numpy as np

# Dictionary where all messages will be stored
messages = {}

class Radio:
    def __init__ (self, robot_id, tag = 'Radio'):
        """
        Radio base class for a robot

        robot_id : int -> id of robot which it is attached
        tag      : str -> name tag of the radio

        """

        self.robot_id = robot_id
        self.tag = tag

        self.pos_x = 0
        self.pos_y = 0
        self.rot_w = 0


    def send_data(self, data) -> None:
        """
        Publish data in global dictionary of messages. Previous
         ↪  data is overwritten.

        Args:
            data : any -> values to be altered

        """
        messages[self.robot_id] = data


    def update(self, x):
        """
        Get current position of the radio

        Args:
            x : np.array -> Positions of all robots
        """

        self.pos_x = x[0, self.robot_id]
        self.pos_y = x[1, self.robot_id]
        self.rot_w = x[2, self.robot_id]


class Radio_range(Radio):
```

```
    def __init__ (self, robot_id, tag, callback, range=1.0):
        """
        Radio gets current position and velocity of all robots
         ↪ that can be detected

        robot_id   : int      -> id of robot which it is
         ↪ attached
        tag        : str      -> name tag of the sensor
        callback   : callable -> callback function
        range      : float    -> noise of the sensor
        """

        self.robot_id = robot_id
        self.tag = tag

        self.callback = callback
        self.range = range


        super().__init__(self.robot_id, self.tag)

    def update(self, x):
        """
        Get current position of the robot attached

        Args:
            x : np.array-> Positions of all robots

        """

        super().update(x)

        data_list = []
        for i in range(np.shape(x)[1]):

            if i in messages:
                dist = np.sqrt((self.pos_x - x[0, i]) ** 2 + (
                 ↪ self.pos_y - x[1, i]) ** 2)

                if dist > 0 and dist <= self.range:
                    data_list.append(messages[i])


        robot_data = self.callback(data_list)
        self.send_data(robot_data)



if __name__ == "__main__":
    print("DREAM Radio class")
```

A.4 dream_timer.py

```
import numpy as np

# List where all timers will be stored
timer_list = []

class Timer:
    def __init__(self, robot_id, timer_delay, list_of_timers):
        """
        Timer class to control when the different elements are
         ↪ to be activated

        robot_id       : int                    -> id of robot
         ↪ which it is attached
        timer_delay    : int                    -> number of
         ↪ steps until the robot is activated
        list_of_timers : list[dict{str, int | float | str |
         ↪ callable}] ->
                         list of timers in the form of [{'delay
                          ↪ ': int | float, 'tag': str, 'call
                          ↪ ': callable}, ...]
        """

        self.robot_id = robot_id        # id of robot which it
         ↪ is attached
        self.timer_delay = timer_delay  # robot start-up delay
        self.list_of_timers = [{'ptime': 0, 'delay': tim['delay
         ↪ '], 'tag': tim['tag'], 'call': tim['call']} for tim
         ↪  in list_of_timers]

        self.current_time = 0


    def update(self, current_time_, param):

        self.current_time = current_time_

        if self.current_time <= self.timer_delay:
            return

        for timer_ in self.list_of_timers:
            if self.current_time - timer_['ptime'] >= timer_['
             ↪ delay']:
                timer_['ptime'] = self.current_time
                timer_['call'](*param[timer_['tag']])


def update_timers(current_time, param) -> None:
    """
    Call this to update all timers
```

```
    Args:
        current_time -> current time of the simulation loop

    Return:
        None
    """

    for timer_unit in timer_list:
        timer_unit.update(current_time, param)



def create_timer(robot_id: int, delay: int, list_of_timers:
 ↪ list) -> Timer:
    """
    Call this to create a timer attached to a robot
    Timer created is stored in timer_list

    Args:
        robot_id : int  -> Id of the robot to which it is
         ↪ attached
        delay    : int  -> Number of iterations before starting
        list_of_timers : list[dict{str, int | float | str |
         ↪ callable}] ->
                         list of timers in the form of [{'delay
                          ↪ ': int | float, 'tag': str, 'call
                          ↪ ': callable}, ...]

    Return:
        Timer -> Timer created
    """

    # Create timer class and append to timer_list
    ti = Timer(robot_id, delay, list_of_timers)
    timer_list.append(ti)

    return ti



if __name__ == "__main__":
    print("DREAM Timer class")
```

A.5 dream_specs.py

```
"""
Robot values
"""
R_V_MAX     : float = 0.2       # max linear velocity
R_W_MAX     : float = 3.636363  # max angular velocity
R_AC_MAX    : float = 12.5      # max actuator velocity
R_R_WHEEL   : float = 0.016     # robot wheel radius
R_R_LENGTH  : float = 0.105     # robot length


"""
Terrain values
"""
T_WIDTH     : float = 3.2
T_HEIGHT    : float = 2.0
T_ORIGIN_X  : float = 0.0
T_ORIGIN_Y  : float = 0.0

T_MAX_WIDTH = T_ORIGIN_X + T_WIDTH / 2
T_MIN_WIDTH = T_ORIGIN_X - T_WIDTH / 2
T_MAX_HEIGHT = T_ORIGIN_Y + T_HEIGHT / 2
T_MIN_HEIGHT = T_ORIGIN_Y - T_HEIGHT / 2


if __name__ == '__main__':
    print("DREAM robot and terrain parameters")
```

Appendix B

Application Example: HRVO experiment

This appendix contains the commented code used for the execution of the first of the experiments carried out. It consists of a new Robot class that inherits from `Robot_hrvo` and uses the DREAM classes. Then, a Robotarium environment is created on which the ten robots with sensors, radios and timers are created, with its corresponding timers list to define the time gaps between updates. During the execution loop at the end, the `update_timers` function is called and passed as a parameter the dictionary containing, identified by a tag, all the input parameters of all the callback methods that are included in the list of timers previously created.

B.1 HRVO experiment code

```
# Import Robotarium
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *

# Import numpy for array manipulation and patches for extra
  ↪ plotting
import numpy as np
import matplotlib.patches as patches

# Import DREAM
from dream import *


class Robot_c(Robot_hrvo):
    def __init__ (self, robot_id, tag, goal_points = np.array
      ↪ (([0.0],[0.0])), vel_gain = 0.8, ang_gain = 3, safety =
```

```
    ↪  0.24, color='k'):
        """
        Robot circle class.
        When its goal position has been reached, state += 1
        Send and receive robot_states by radio communications
        When all robot_states are the same, go to next goal
         ↪ point
        """

        self.robot_id = robot_id
        self.tag = tag
        self.goal_points = goal_points
        self.current_goal = self.goal_points[:, 0]

        self.vel_gain = vel_gain
        self.ang_gain = ang_gain

        self.delta_x = 0
        self.delta_y = 0

        self.desired_velocity = np.array((0.0, 0.0))

        self.robot_states = [0,0,0,0,0,0,0,0,0,0]
        self.state = 0
        self.change_state = True

        # Plot goal point and current velocity over the terrain
        self.color = color
        self.finalgoal = None
        self.target = None
        self.line1 = None
        self.line2 = None


        super().__init__(self.robot_id, self.tag, vel_gain=
         ↪ vel_gain, ang_gain=ang_gain, safety=safety)



    def update_radio(self, radio_data):
        self.radio_data = radio_data

        # Update variable with current robot states
        for i in range(len(self.radio_data)):
            for j in range(len(self.robot_states)):
                if self.radio_data[i][j] > self.robot_states[j
                 ↪ ]:
                    self.robot_states[j] = self.radio_data[i][j
                     ↪ ]

        self.robot_states[self.robot_id] = self.state
```

```
        # Check if all robots has reached its goal
        if all(flag == self.state for flag in self.robot_states
          ↪ ):
            if self.state < 3:
                self.current_goal = self.goal_points[:, self.
                  ↪ state]
                self.change_state = True

        # Send message to radio
        return self.robot_states


    def update_sensor(self, detections):
        self.detections = detections


    def update_odom(self, pos):
        self.pos_x = pos[0]
        self.pos_y = pos[1]
        self.rot_w = pos[2]



    def update(self, dxu_):

        # Compute desired velocity to reach the goal
        self.delta_x = self.current_goal[0] - self.pos_x
        self.delta_y = self.current_goal[1] - self.pos_y

        dist_error = np.sqrt(self.delta_x ** 2 + self.delta_y
          ↪ ** 2)
        angle_error = np.arctan2(self.delta_y, self.delta_x)

        if dist_error < 0.05:
            dist_error = 0
            self.desired_velocity = np.array((0.0, 0.0))

            if self.change_state:
                self.state += 1
                self.change_state = False

        else:
            self.desired_velocity[0] = self.v_max * self.
              ↪ delta_x / dist_error
            self.desired_velocity[1] = self.v_max * self.
              ↪ delta_y / dist_error


        # Compute HRVO to avoid obstacles
        self.HRVO(self.detections, self.desired_velocity)
```

```
        # Convert dxi velocities from HRVO to dxu
        self.dxi_to_dxu()

        # Limit velocity to not exceed actuator limits
        self.actuator_limit()



        # Return self.dxu
        dxu_[0][self.robot_id] = self.dxu[0]
        dxu_[1][self.robot_id] = self.dxu[1]


        # Plot goal points and hrvo points
        if self.finalgoal is not None:
            self.finalgoal.remove()
            self.target.remove()
            self.line1.remove()
            self.line2.remove()

        self.finalgoal = plt.Circle((self.current_goal[0], self
          ↪ .current_goal[1]), 0.04, color=self.color)
        r.axes.add_patch(self.finalgoal)
        self.target = plt.Circle((self.dxi[0]+self.pos_x, self.
          ↪ dxi[1]+self.pos_y), 0.02, color=self.color)
        r.axes.add_patch(self.target)
        vertices=[(self.pos_x, self.pos_y), (self.dxi[0]+self.
          ↪ pos_x, self.dxi[1]+self.pos_y), (self.dxi[0]+self.
          ↪ pos_x, self.dxi[1]+self.pos_y), (self.pos_x, self.
          ↪ pos_y)]
        self.line1 = patches.Polygon(vertices, fill=False, lw
          ↪ =0.5, ls='-', color=self.color)
        r.axes.add_patch(self.line1)
        vertices=[(self.dxi[0]+self.pos_x, self.dxi[1]+self.
          ↪ pos_y), (self.current_goal[0], self.current_goal
          ↪ [1]), (self.current_goal[0], self.current_goal[1]),
          ↪  (self.dxi[0]+self.pos_x, self.dxi[1]+self.pos_y)]
        self.line2 = patches.Polygon(vertices, fill=False, lw
          ↪ =0.5, ls='-', color=self.color)
        r.axes.add_patch(self.line2)




"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
```

```
# Define number of robots and number of iterations of the
  ↪ simulation
N = 10
iterations = 1000

# Create a circlular shape for the N robots
radius = 0.85
p_theta = 2 * np.pi * (np.arange(0, 2 * N, 2)/(2 * N))
p_circ = np.vstack([
            np.hstack([radius * np.cos(p_theta), radius * np.
              ↪ cos(p_theta+np.pi)]),
            np.hstack([radius * np.sin(p_theta), radius * np.
              ↪ sin(p_theta+np.pi)])
            ])

x_goal_1 = p_circ[:, :N]
x_goal_2 = p_circ[:, N:]

goal_points = np.vstack((x_goal_1[:, 0], x_goal_2[:, 0],
  ↪ x_goal_1[:, 0]))
for i in range(1, N):
    goal_points = np.vstack((goal_points, x_goal_1[:, i],
      ↪ x_goal_2[:, i], x_goal_1[:, i]))
goal_points = goal_points.T


# Create a Robotarium instance
r = robotarium.Robotarium(number_of_robots = N, show_figure =
  ↪ True, sim_in_real_time = False)

# Create tags for callback methods
robot_tag = 'robot'
movement_tag = 'movement'
odometry_tag = 'odom'
radio_tag = 'radio'

colors = ['black', 'red', 'gold', 'olive', 'yellow', 'lawngreen
  ↪ ', 'lime', 'cyan', 'navy', 'magenta']

# Create N robots
for i in range(N):

    # Create a robot, a movement sensor, an odometry sensor and
      ↪  a radio
    rob = Robot_c(i, f'{robot_tag}_{i}', goal_points =
      ↪ goal_points[:, 3*i:3*i+3], color = colors[i])
    mov = Movement_sensor(i, f'{movement_tag}_{i}', rob.
      ↪ update_sensor)
    odo = Odometry_sensor(i, f'{odometry_tag}_{i}', rob.
      ↪ update_odom)
```

```
    rad = Radio_range(i, f'{radio_tag}_{i}', rob.update_radio)

    # Create list of timers
    list_timers = [
        {'delay': 4 + (i%2), 'tag': robot_tag, 'call': rob.
          ↪ update},
        {'delay': 3 + (i%2), 'tag': movement_tag, 'call': mov.
          ↪ update},
        {'delay': 2 + (i%2), 'tag': odometry_tag, 'call': odo.
          ↪ update},
        {'delay': 30, 'tag': radio_tag, 'call': rad.update}
    ]

    # Create Timer
    create_timer(i, 20 + i, list_timers)



# Simulation first step
dxu = np.zeros((2,N))
x = r.get_poses()
r.step()

# Rest of simulation steps
for i in range(iterations):

    # Get current poses from Robotarium
    x = r.get_poses()

    # Create dictionary of parameters for callback functions
    params = {movement_tag: (x, dxu), odometry_tag: (x,),
      ↪ radio_tag: (x,), robot_tag: (dxu,)}

    # Update Timer
    update_timers(i, params)

    # Send velocities to Robotarium
    r.set_velocities(np.arange(N), dxu)
    r.step()


r.call_at_scripts_end()
```