Formation-centric software interface

by

Rohan Walia

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Masters of Science

in

Robotics Engineering

by

July 2022

APPROVED:

Prof. Carlo Pinciroli, Advisor

Prof. William Michalson

Prof. Markus Nemitz

Abstract

Multi-robot systems are often deployed as formations to accomplish tasks where a single robot has limited potential, such as collective transport, search and rescue missions and choreographed robot motions such as drone light shows. Due to the nature of these applications, retaining the formation shape plays an integral role in the success of the task. Extensive research has been conducted on how to effectively control and navigate robot formations to complete these tasks while retaining the underlying shape. However, in most cases, robots are manually assigned to create a formation shape and require specific control laws to maintain that shape throughout the task. Programming each robot manually becomes cumbersome when number of robots increases or when the formation shape changes frequently.

To tackle these challenges, we develop an intuitive and concise software interface for creating and managing robot formations. As part of this interface, we introduce data types that enable users to easily define robot formations. We also introduce functions that 1) quickly assign robots to a desired formation shape, 2) change shapes of existing formations and 3) navigate a rigid formation in a mapped environment. We demonstrate the use of this interface through sample programming scripts accompanied by simulated and real robot experiments. We also describe the underlying software architecture and navigation techniques that support this work.

Acknowledgements

I want to express my gratitude towards my thesis advisor - Professor Carlo Pinciroli. He gave me invaluable inputs with immense interest and patience despite his numerous responsibilities inside and outside the classroom. He fostered a wonderful culture of respect and curiosity at the NEST lab. His efforts to facilitate state of the art software and hardware enables quality research.

I would like to thank Professor William Michalson and Professor Markus Nemitz for serving on my thesis committee and providing their feedback on how to improve my work.

I greatly appreciate my peers at the Nest Lab. This work was made possible because of Stephen Powers' contributions to develop the underlying software architecture. He supported me during the design and debugging phases of this work. I also thoroughly enjoyed my interactions with Ashay Aswale, Khai Yi Chin, Davis Catherman, Arsalan Akhter and Josh Bloom.

I received unbelievable support through WPI in different forms. The Robotics Engineering Faculty and Staff helped me learn robotics - a field I took a chance on two years ago. WPI International House made it extremely easy to be a student during a global pandemic. My manager, Kristin Marengo, helped keep my work schedule flexible while I took graduate coursework. Lori Kendall-Taylor, my student success manager, ensured I had the support to keep going. Lastly, my friends made it possible to not make this journey alone.

Most importantly, I would like to thank to my family. Their interest in supporting everything I do is the reason I am here. How they do it is beyond my understanding, but I am glad they do.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Robot swarms have demonstrated greater performance and reliability in certain tasks as compared to their single-robot counterparts. For example, Sauter *et al.* [29] have observed that coordinated unmanned aerial and ground vehicles can be used to improve human operated target acquisition military systems. By adding semi-autonomous swarms, the operations can be scaled by eliminating the need for assigning one human controller per vehicle. Additionally, these semi-autonomous swarms are designed to compensate for individual robots failing during the operation.

In many swarm robotics applications, the main strength of a swarm is decentralized coordination, which leads to scalability and robustness for the overall system [11] [18]. However, the shape of a swarm is also crucial for a variety of tasks. When robot swarms are arranged in a desired shape, a robot formation is created. In a recent study on aerial package delivery, Alkouz and Bouguettaya [10] observed how energy consumption changes for a drone formation based on the formation shape and wind conditions. They experimented with two formation types: fixed and adaptive.

Fixed formations were not allowed to change their shape while adaptive formations were allowed to change between five flight formation shapes: Column Formation, Front, Echelon, Vee and Diamond (shown in Figure 1.1). After subjecting fixed and adaptive formations to different wind conditions, they concluded that adaptive formations performed better in terms of energy consumption over different trip distances and wind speeds. Therefore, the ability of the drones to acquire different formations played a key role in this scenario.

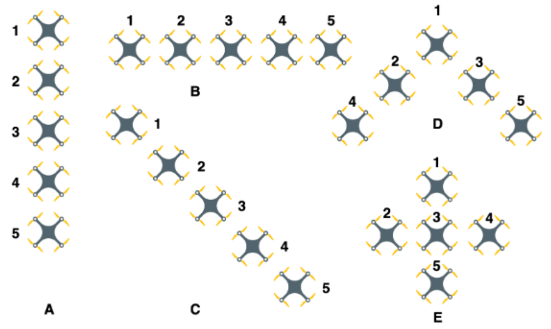


Figure 1.1: Five different flight formations used by Alkouz and Bouguettaya. A: Column, B: Front, C: Echelon, D: Vee, E: Diamond

Besides package delivery, drones have also been used in light shows in the entertainment industry [19]. Such applications where robots need to swiftly transition from one formation to another and execute coordinated motion patterns fall under the umbrella of robot choreography. The ability to express robot motion through different formation shapes lies at the heart of robot choreography.

Collective transport is another application where shape of the formation plays a central role. In collective transport, a swarm of mobile robots is used to transport a rigid object in a given environment. Needless to say, the formation shape needs to match the contour of the object to be transported. Collective transport has applications in warehouse management, where ARMs (autonomous mobile robots) can be used to transport oversized or heavy objects. Research efforts for collective transport scenarios have primarily been concentrated towards control and navigation of formations [27] [15] [16].

Despite the benefits of arranging robots into specific formation shapes, there is no software which abstracts low-level details of arranging robots into formations and easily changing existing formation shapes. Having access to such software would allow more expressive and faster software development when it comes to programming robot formations. Successful robotics software applications such as ROS were not designed for robot swarms, let alone formations shapes [25]. Although some swarm-robotics software packages for ROS touch on this subject, they were primarily designed to provide a library for commonly observed swarm behavior [17] and decentralized coordination [31]. Moreover, even software specifically designed for swarm robotics is not intended for providing out-of-box solutions for programming robot formations [25].

1.2 Problem Statement

The first objective of this thesis is to introduce a software interface for programming robots from the perspective of formation shapes. To accomplish this objective, we intend to provide user-friendly methods for creating formations and performing formation assignment and manipulation. Formation assignment should enable a programmer to rearrange a given distribution of robots into a desired shape. Formation manipulation should allow a programmer to apply a set of simple motion operations to a formation. These operations, or primitives, should be applied by treating the formation as either a rigid or a deformable body. Motion primitives that respect the rigidity of a formation include translation and rotation. Examples of motion primitives that treat the formation as a deformable body include scaling (increasing or decreasing the formation size) or shearing (skewing the formation shape horizontally or vertically).

The second objective of this thesis is to empirically demonstrate the efficacy of this software interface through simulated and real robot experiments. This requires adopting a navigation technique that enables robots to reach their goals while avoiding collisions with each other.

1.3 Contribution

For the first objective of this thesis, we introduce data types and functions that describe the structure and functionality of our software interface. We demonstrate the use of these data types and functions by sharing sample programming scripts. We discuss our problem formulation for formation assignment through MILP (Mixed-Integer Linear Programming) and its implementation through MATLAB's `intlinprog` interface. Additionally, we share our implementation for a set of four motion primitives that constitute formation manipulation.

For the second objective of this thesis, we describe a potential field based approach for navigation and demonstrate its use through pairs of simulated and real-robot experiments. We also demonstrate a path planning application by performing collective transport in simulation and with real robots. Finally, we analyze the performance of our MILP-based algorithm for formation assignment and discuss the future direction for this work.

Chapter 2

Related Work

Software development for swarm robotics mainly focuses on enabling programming for scalable centralized and decentralized systems. Usually the intended use of such software is to deploy algorithms that result in emergent behavior. Due to this reason, only a handful of software deals with robot formations and their shapes. Swarm robotics research, as compared to swarm robotics software, has paid more attention to robot formations. Nevertheless, majority of the research efforts have been invested towards developing robust navigation and control strategies for robot formations.

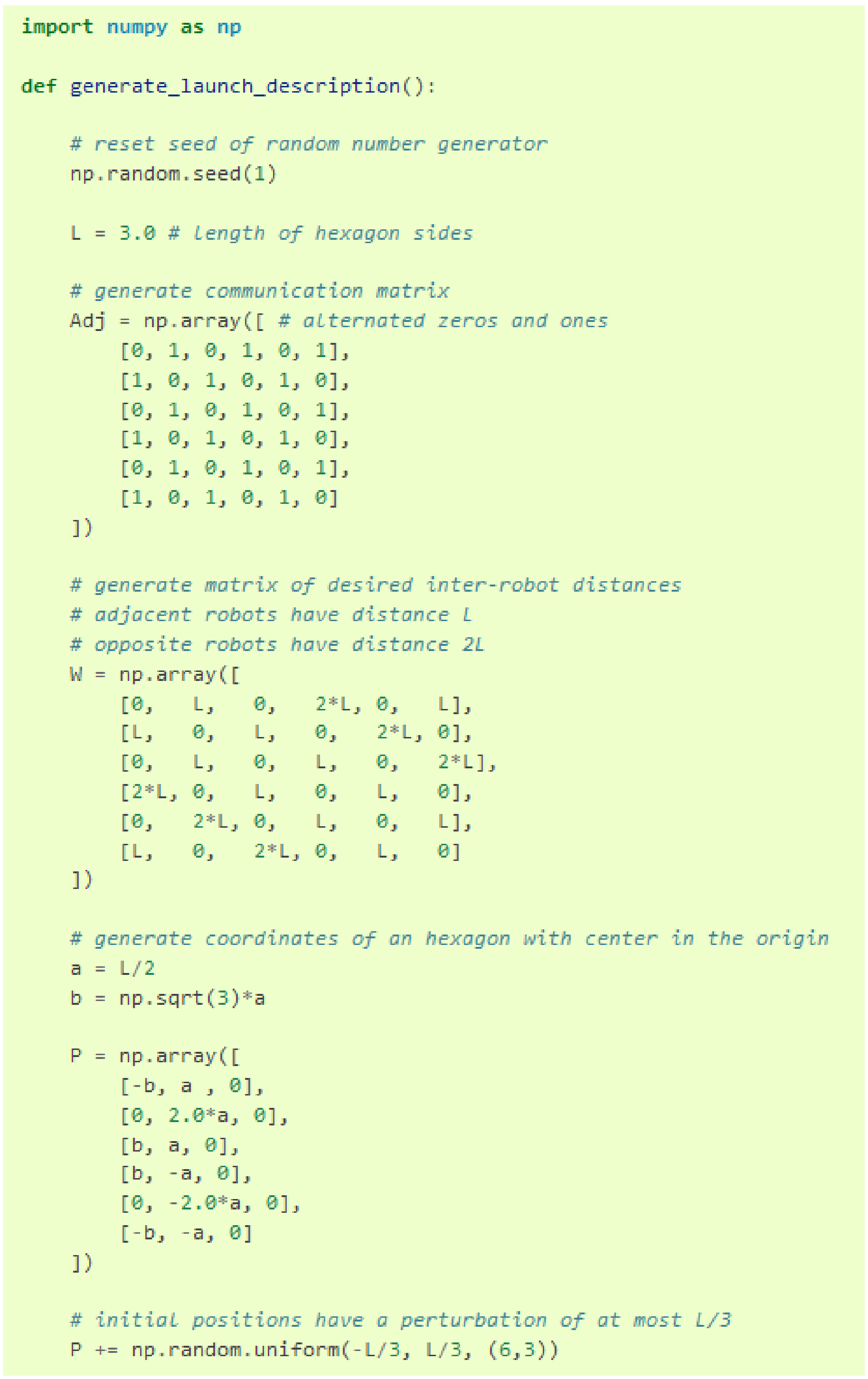
Therefore, the purpose of this section is to highlight what gaps these software development and research efforts leave when it comes to programming robot formations, and how we fill those gaps. To do that, we look at software applications which cater to robot formations in any capacity. We also look at research efforts that touch on the idea of formation management, which includes arranging robots into a desired formation and defining simple motion operations for existing formations. Finally, we discuss how a minimalistic robot motion design language can lead towards programming for robot formations.

2.1 Swarm Robotics Software

Standalone swarm-robotics software such as the Buzz programming language [25] provides out-of-box solutions for swarm management. This includes support for generating motion patterns commonly found in the field of swarm robotics such as aggregation, dispersion and flocking. Out of these motion patterns, flocking is considered coordinated motion of a swarm. However, this coordinated motion is not guaranteed to retain a desired shape.

Several swarm robotics software packages also exist for the widely adopted middleware ROS (Robot Operating System). These packages make swarm robotics programming accessible to a larger audience. Because of this reason, they are designed to fulfil mainstream swarm robotics requirements such as virtual stigmergy and neighborhood management. For example, `micros_swarm_framework` for ROS1 [12] follows in Buzz's footsteps to provide data structures that enable swarm management and communication mechanisms. Although it lists applications in "motion and spatial coordination" of a swarm and "splitting [a swarm] into multiple swarms", there is no mention of what level of control a user has over the shape of the swarm.

More recent packages (geared towards ROS2) branch off from generic coordinated motion such as flocking and focus on formations. The `ChoirBot` package [31] allows users to create rigid formations by specifying inter-agent distances and coordinates of the formation shape as matrices. Even though a user can specify these matrices explicitly, they have to be recreated and the entire program needs to rerun every time the shape is changed. In other words, there is no provision for programmatically reassigning shape to a formation once it has been created. Figure 2.1 shows a code snippet that portrays the setup process for creating a formation shape:



```
import numpy as np

def generate_launch_description():

    # reset seed of random number generator
    np.random.seed(1)

    L = 3.0 # length of hexagon sides

    # generate communication matrix
    Adj = np.array([ # alternated zeros and ones
        [0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0]
    ])

    # generate matrix of desired inter-robot distances
    # adjacent robots have distance L
    # opposite robots have distance 2L
    W = np.array([
        [0,   L,   0,   2*L, 0,   L],
        [L,   0,   L,   0,   2*L, 0],
        [0,   L,   0,   L,   0,   2*L],
        [2*L, 0,   L,   0,   L,   0],
        [0,   2*L, 0,   L,   0,   L],
        [L,   0,   2*L, 0,   L,   0]
    ])

    # generate coordinates of an hexagon with center in the origin
    a = L/2
    b = np.sqrt(3)*a

    P = np.array([
        [-b, a , 0],
        [0, 2.0*a, 0],
        [b, a, 0],
        [b, -a, 0],
        [0, -2.0*a, 0],
        [-b, -a, 0]
    ])

    # initial positions have a perturbation of at most L/3
    P += np.random.uniform(-L/3, L/3, (6,3))
```

Figure 2.1: Specifying formation shape parameters in ChoirBot. Matrices W and P need to be changed every time a new formation shape is required, and the program is rerun to create a new shape.

Swarm robotics software at large also fails to provide an interface for manipulating existing formations. 'Manipulation' here refers to the ability to perform basic motion operations on a formation. These include rigid body operations such as translation and rotation, or changing the shape of the formation such as scaling the formation size up or down. `ROS2Swarm` [17] is a package for ROS2 that introduces the concept of motion patterns. Regardless, each motion pattern emulates emergent behavior and only the 'drive' motion pattern is designed to manipulate a formation.

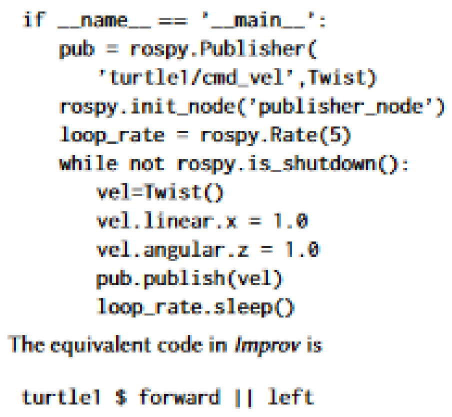
2.2 Formation Management

In order to change the shape of a formation, it is crucial to decide how robots will be arranged to form the desired shape. This is the essentially a goal assignment problem. Goal assignment maps a set of robots to a formation shape in the most optimal manner. To solve goal assignment, one needs to define a cost function (with constraints) and a method for solving the cost function (a solver). Turpin *et al.*'s cost function is the collective distance travelled by robots to form a given shape [33]. They calculate this distance using an optimal path planner in a discretized environment represented as a roadmap. Then, they use the Hungarian algorithm as their solver to find the most optimal assignment (corresponding to the least collective distance travelled). However, this method requires knowing the robot dynamics beforehand. This is because this method also calculates trajectories for each robot based on the goal assignment. In our work, we take a similar approach to solve goal assignment by minimizing the total distance travelled by the swarm. Yet, our method does not require knowing robot dynamics beforehand because we separate the processes of goal assignment and navigation.

The second challenge in formation management is changing a formation's position/orientation or size/structure. This can be achieved by defining a set of motion primitives for the entire formation. The intention here is for all the robots to execute a motion primitive in unison. Du *et al.* [13] define a set of motion primitives (including rigid body rotation), but each motion primitive is designed to be periodic. This requires the motion to be executed repeatedly. In our work, we define a set of motion primitives that are executed one at a time, without the need for repeated execution.

2.3 Minimalistic Robot Motion Design

As discussed earlier, `ChoirBot` [31] already lists an interface for creating a robot formation. Similarly, [33] provides a goal assignment algorithm for creating robot formations and [13] introduces periodic motion primitives that can be used with existing formations. However, these features are not accessible to programmers in a user-friendly manner. Abstracting the low-level implementation details of creating and manipulating a formation would allow users to program from a formation's point of view, instead of individual robots' point of view. The benefit of this approach has already been demonstrated for single robots by Nilles *et al.* in [21]. In this work, Nilles *et al.* introduce *Improv*, a language which deals with "a high-level description for robot motion." Figure 2.2 shows a comparison between two code samples from ROS and *Improv* that achieve the same purpose: executing a turn maneuver.



```
if __name__ == '__main__':
    pub = rospy.Publisher(
        'turtle1/cmd_vel',Twist)
    rospy.init_node('publisher_node')
    loop_rate = rospy.Rate(5)
    while not rospy.is_shutdown():
        vel=Twist()
        vel.linear.x = 1.0
        vel.angular.z = 1.0
        pub.publish(vel)
        loop_rate.sleep()
```

The equivalent code in *Improv* is

```
turtle1 $ forward || left
```

Figure 2.2: Comparison of python-ROS and *Improv* code to execute a turn maneuver on a Turtlebot.

It can be clearly seen that *Improv's* code does not require the setup procedure, which includes creating a publisher, initializing a ROS node, defining a rate for the communication loop, and specifying the motion termination condition explicitly. This difference in code complexity will be even more profound between *Improv* and a lower-level language such as C++. Niles *et al.* argue that the low level of abstraction required by ROS does not allow users to easily "translate their mental model of movement" for planning and visualizing motion. One of the motivations behind developing *Improv* is to reduce the difference between a user's "mental model of movement" and the code. We take inspiration from *Improv* to develop a software interface which is minimalistic and robot agnostic to draw the focus away from individual robots and bring it towards the shape of an entire formation.

Chapter 3

Methodology

3.1 Problem Formulation

The objective of this thesis is to develop a software interface which allows users to interact with robot formations programmatically. This interface must enable programming from the perspective of a formation while abstracting the low level implementation details. We identify the following three requirements to achieve this objective:

1. **Formation creation:** The user should be able to create a formation in an intuitive and concise manner. This requires establishing what constitutes a formation, and how the user provides that information.

2. **Formation manipulation:** Formation manipulation is the ability to describe a formation's movement or changes to its shape. It enables a user to navigate or adapt a formation to its environment. Formation manipulation should not require a user to explicitly coordinate motion for each individual robot.

3. **Formation assignment:** Throughout the course of a task, a swarm of robots

needs to achieve different formations. To change the shape of a formation, the user should only be required to describe the new shape. They should not be asked to manually rearrange robots.

Our formulations for each of these requirements are explained in the following subsections:

3.1.1 Formation Creation

To represent a geometric shape using robots, one needs to come up with the spatial distribution that shows how robots form this shape. This can be achieved by providing a set of points in space that constitute the geometric shape. We call this set of points a `cloud`.

A `Cloud` is an unordered distribution of agents in space, akin to the concept of a point cloud [20]. 'Unordered' here means that any agent could occupy any position within the shape, as long as it belongs to the given set of positions. For example, the Figure 3.1 shows a rectangular cloud of four agents in a 2D environment, where each agent can be placed on any one corner of the rectangle:

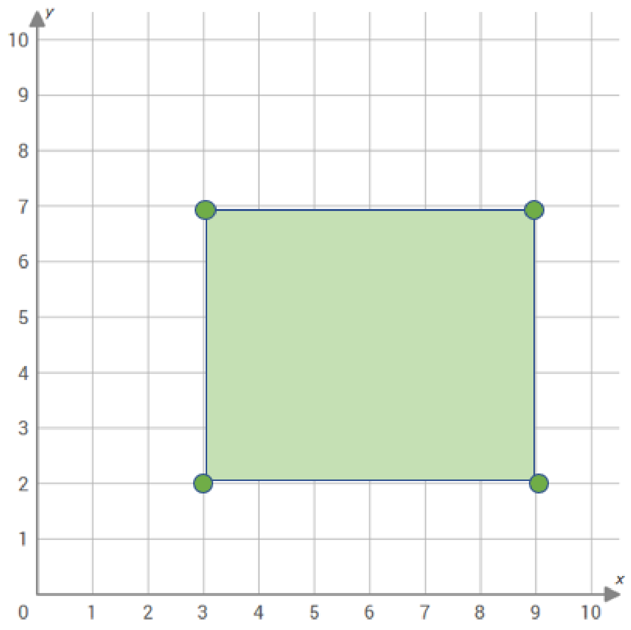


Figure 3.1: Example of a 2D rectangular cloud. Agents are allowed to be placed at (3, 2), (3, 7), (9, 7) and (9, 2).

A cloud is sufficient to describe the geometry of a shape. However, it does not describe how robots are allocated to a shape. For example, in order to fill a cloud of 4 robots in an environment of 10 robots, the user needs to specify which 4 robots would be used to fill the cloud.

To solve this problem, we establish the concept of a `Formation`. A `Formation` is an ordered group of agents, where each agent is assigned an ID to keep track of its position within the a shape. A formation can be described by the following attributes:

1. **Cloud:** A cloud describes the shape of the formation as a set of robot poses in space.

2. **Agent Order:** Agent order is an ordered list (array) of robot IDs that correspond to each robot pose in the cloud attribute.

3. **Orientation:** Since a formation represents a geometric shape, a user cannot infer where the formation points through visual cues. Therefore, an orientation is established when the formation is created to keep track of the 'head' of a formation with respect to the world coordinates of its environment. This is particularly important for aligning the formation in a specific direction.

4. **Agent Diameter:** To avoid collisions, each agent's geometry needs to be known. An agent's diameter describes its circular or spherical footprint. This information is required to ensure that given list of robots do not collide with each other to create the formation.

Section 3.4 describes how we incorporate these concepts into a programming interface.

3.1.2 Formation Manipulation

As mentioned earlier, formation manipulation affects the position or shape of a formation. This requires coordinating motion of all robots in the entire formation. To abstract the implementation details of such coordinated motion, we define a set of primitive operations that only require a single parameter to describe the operation's affect on the entire formation.

These primitive operations, or motion primitives, ultimately describe how position of each robot changes in the formation. Therefore, they can be defined as linear operations on the coordinates for each robot in a formation. We provide a set of four motion primitives: translate, rotate, scale and shear. Translate and rotate treat a formation as a rigid body and change its 'pose'. Scale and shear change the shape of the formation by treating the formation as a deformable body. Section 3.2 contains a detailed description and formulation of these primitives.

3.1.3 Formation Assignment

Formation Assignment is the problem of calculating how to rearrange robots in a formation to create a new shape. Formations represent geometric shapes using a set of robot positions. Therefore, this rearrangement problem boils down to determining what robot in the starting formation occupies which position in the given shape. We call this a 'mapping' of robots between the two shapes.

One way to select the best mapping is to reduce the time taken by the swarm to transition between these shapes. The total time taken by a swarm to transition between the shapes is the sum of time taken by all individual agents to move from their respective start position to goal positions. Mathematically, this can be expressed as:

$$T = \sum_{1}^{n} (\text{speed} \cdot d_r) \tag{3.1}$$

where n is the swarm size, d_r is agent r's displacement, and 'speed' is the magnitude of velocity shared by all agents in the swarm. If velocity magnitude of all agents is the same, then the time taken by each individual agent to move from its start location to its goal location is proportional to the displacement d_r ($time = \frac{|displacement|}{|velocity|}$). Here, a d_r value is calculated for every mapping of start position in the formation to a goal position in the given shape. Therefore, an optimization algorithm would reach at the best possible goal assignment by choosing the appropriate d_r value for each agent.

We formulate a rudimentary solution to this problem by assuming that the swarm is homogeneous, i.e., all agents have identical geometry, as well as kinematic and dynamic characteristics. Essentially, this requirement indicates that all agents have the same geometric footprint and acceleration profiles. Additionally, we assume 1-1 mapping to ensure that every agent is assigned a unique position in the given shape. A detailed formulation of goal assignment is covered in Section 3.3.

3.2 Motion Primitives

To understand how the default primitives affect a formation, assume matrices F_x and F_y represent the x and y positions of agents in a 2D formation of four agents arranged in a square shape. The formulation for each motion primitive can then be described as follows:

1. **Translate**: The translate primitive displaces a rigid formation in space using a translation vector. The translation vector determines the length and direction of the displacement. For a point (x, y) in 2D formation, the translate operation changes the point as follows:

$$\begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.2}$$

   where $[t_x \;\; t_y]'$ is the translation vector, and $[x_{\text{new}} \;\; y_{\text{new}}]'$ is the new position vector for the point.

   For example, if:

$$F_x = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

   then a translation vector [1 1]' changes F_x and F_y as follows:

$$F_x = \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}$$

This process is shown in Figure 3.2 below. x and y positions of each agent in F_x and F_y correspond to their actual position in the formation shown in the figure (starting clock-wise from top left in F_x and F_y, agent 1 corresponds to F_{x11} and F_{y11}).

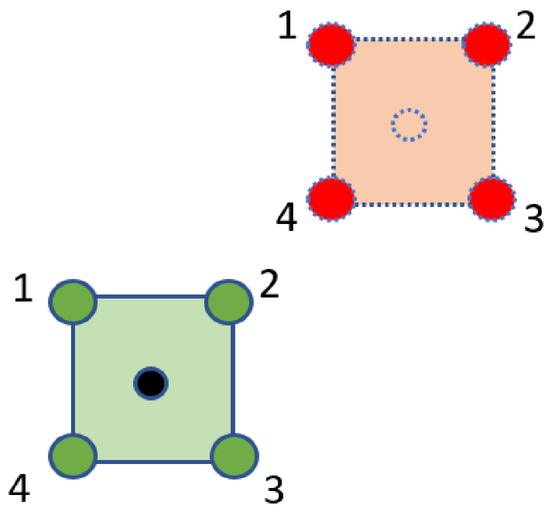


Figure 3.2: Translation process for four agents in a square formation. Green formation is the initial formation, orange formation is the final formation.

2. **Rotate (rotation angle)**: This primitive rotates a rigid formation about its center of mass. This is done by applying the same rotation (with respect to the center of mass of the formation) to x and y positions of each agent in the formation.

   For a point (x, y) in a 2D formation, the rotate operation changes the point as follows:

$$\begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - \text{com}_x \\ y - \text{com}_y \end{bmatrix} + \begin{bmatrix} \text{com}_x \\ \text{com}_y \end{bmatrix} \tag{3.3}$$

   where θ is the counter-clockwise angle of rotation measured with respect to the local frame of the formation. com_x and com_y are the x and y components of the center of mass of the formation. $[x_{\mathrm{new}}\ y_{\mathrm{new}}]'$ is the new (rotated) point. The local frame of the formation always has y axis pointing up, and the x axis pointing to the right in a 2D plane.

For example, if:

$$F_x = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

then $\theta = \frac{\pi}{4}$ changes F_x and F_y as follows:

$$F_x = \begin{bmatrix} -0.2071 & 0.5000 \\ 0.5000 & 1.2071 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 0.5000 & 1.2701 \\ 0.5000 & -0.2071 \end{bmatrix}$$

This process is shown in Figure 3.3.

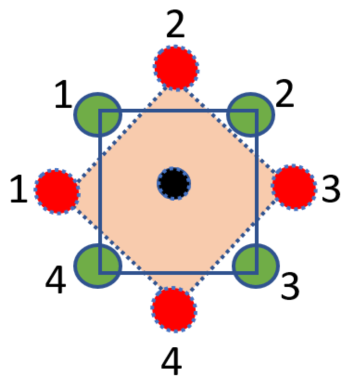


Figure 3.3: Rotation process for four agents in a square formation. Green formation is the initial formation, orange formation is the final formation. Notice that the center of mass of the initial and final formations coincide, since each agent is rotated about the center of mass of the initial formation.

3. **Scale:** The scaling primitive scales the radial distance of each agent in the formation from the center of mass of the formation, effectively scaling the formation diameter. Formation diameter is calculated based on Euclidean

distance between the centers of the two most diametrically opposite agents in the formation. The scaling primitive preserves the inter-agent angle, but it does not preserve the inter-agent distance. Note that the center of mass of the formation remains the same.

For a point (x, y) in a 2D formation, the scale operation changes the point as follows:

$$\begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} = s \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.4}$$

$$\begin{bmatrix} x'_{\text{adjusted}} \\ y'_{\text{adjusted}} \end{bmatrix} = \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} - \begin{bmatrix} \text{com}_{\text{diff}x} \\ \text{com}_{\text{diff}y} \end{bmatrix}$$

where s is the scale factor and $[x_{\text{new}} \;\; y_{\text{new}}]'$ is the vector obtained through the scaling operation. $[x'_{\text{adjusted}} \;\; y'_{\text{adjusted}}]'$ is the final point calculated after adjusting for the change in the center of mass due to scaling.

For example, if:

$$F_x = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

then a scale factor of 2 changes F_x and F_y as follows:

$$F_x = \begin{bmatrix} -0.5 & 1.5 \\ -0.5 & 1.5 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 1.5 & 1.5 \\ -0.5 & -0.5 \end{bmatrix}$$

This process is shown in Figure 3.4 below:

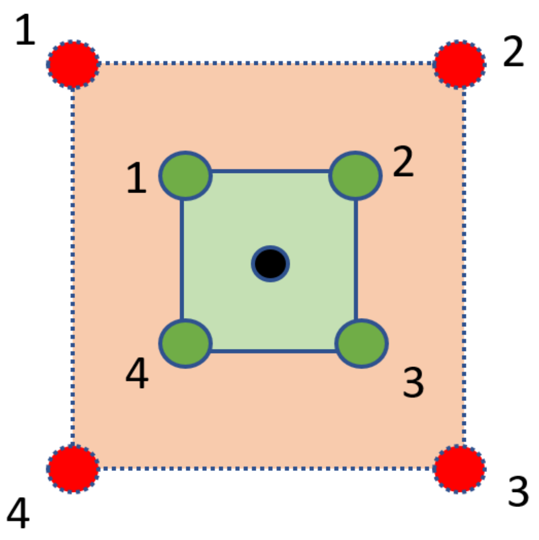


Figure 3.4: Scaling process for four agents in a square formation. Green formation is the initial formation, orange formation is the final formation. Notice that the center of mass of the initial and final formations coincide, since position of each agent is scaled about the center of mass of the initial formation.

4. **Horizontal shear**: Horizontal shear preserves the vertical height of the formation, but skews the formation horizontally. Each agent is displaced in proportion to how far it is from the bottom-most point of the formation.

   For a point (x, y) in a 2D formation, the shearing operation changes the point as follows:

$$\begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.5}$$

   where s is the shear factor, and $[x_{\text{new}}\ y_{\text{new}}]'$ is the new vector obtained through shearing.

   For example, if:

$$F_x = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

then a shear factor of 2 changes F_x and F_y as follows:

$$F_x = \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

As shown in F_x and F_y above and in Figure 3.5 below, only the top two agents are shifted to the right. The bottom two agents are co-linear with the bottom-most point of the formation, and therefore are not affected by the shearing operation.

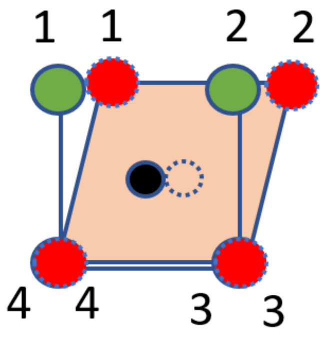


Figure 3.5: Horizontal shearing process for four agents in a square formation. Green formation is the initial formation, orange formation is the final formation. Notice that the center of mass of the initial and final formations do not coincide in this case.

3.3 Goal Assignment Formulation

3.3.1 1-1 Mapping

While transitioning from a formation to a new shape, each agent in the formation has to be 'mapped' to a position in the shape. 1-1 mapping ensures that undesirable or impossible goal assignments do not occur, such as two agents in the starting formation having the same goal in the goal shape, or one agent in the goal shape having two starting positions in the starting formation. It is assumed that the number of agents in the starting formations and number of positions in the goal shape should match, otherwise the assignment would not be possible. Figure 3.6 shows visualization of a mapping for a swarm of 5 agents.

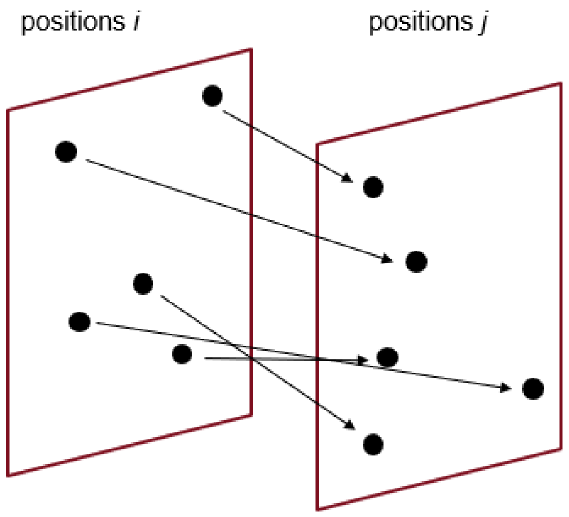


Figure 3.6: Initial (i) and final (j) robot distributions

To design a solution to the goal assignment problem, we model agents as point masses. Let the configuration space of all agents be discretized into a grid of size MxN, represented as $C \in R^2$. Let C_{free} be the obstacle-free subset of C. Let $C_s \in C_{\text{free}}$ and $C_g \in C_{\text{free}}$ be the given set of start and goal states of the agents. The goal assignment problem with the 1-1 mapping constraints can now be mathematically expressed as:

$$\forall i \sum_j m_{ij} = 1 \tag{3.6}$$

such that:

$$m_{ij} = \begin{cases} 1 & \text{if (i, j) are connected} \\ 0 & \text{otherwise} \end{cases}$$

where m_{ij} represents the mapping from start location C_i to goal location C_j.

3.3.2 Mapping and Cost matrices

To optimize goal assignment, each mapping pair m_{ij} needs to have a cost associated with it. This cost can be calculated based on the problem description. In this work, we calculate the cost of a mapping pair m_{ij} as the straight line Euclidean distance form C_i to C_j.

The mapping information is stored in a 'mapping matrix' M, while the cost of each individual mapping pair is stored in a 'cost matrix' H. Mathematically, the mapping and cost matrices for s start locations and g goal locations can be written as:

$$M = \begin{bmatrix} m_{1,1} & \cdots & m_{1,g} \\ \cdots & \ddots & \cdots \\ m_{s,1} & \cdots & m_{s,g} \end{bmatrix} \tag{3.7}$$

$$H = \begin{bmatrix} h_{1,1} & \cdots & h_{1,g} \\ \cdots & \ddots & \cdots \\ h_{s,1} & \cdots & h_{s,g} \end{bmatrix} \tag{3.8}$$

Each element $h_{i,j}$ corresponds to the cost associated with mapping $m_{i,j}$.

3.3.3 Encoding Goal Assignment In MATLAB

This section illustrates how MATLAB's `intlinprog` interface is used to select the best mappings $m_{i,j}$ based on cost matrix values $h_{i,j}$. `intlinprog` interface uses Multi-Integer Linear Programming (MILP) to solve for problems of the form:

$$\min_x f^T x \text{ subject to } \begin{cases} x(\text{intcon}) \text{ are integers} \\ A \cdot x \leq b \\ A_{\text{eq}} \cdot x = b_{\text{eq}} \\ lb \leq x \leq ub \end{cases} \tag{3.9}$$

Here $f^T \cdot x$ is the cost function, f is the vector of of linear coefficients in the cost function, and x is the solution vector. MATLAB minimizes the cost function by default. Matrix A_{eq} and vector b_{eq} store linear equality constraints for the problem, whereas matrix A and vector b store the linear inequality constraints for the problem. Note that A and b can be empty if there are no inequality constraints in the problem. Vector `intcon` specifies which elements of solution vector x are integer-valued. Finally, vectors lb and ub specify the lower and upper bounds for the elements of the solution vector. The challenges now are:

1. Express the optimization function (3.9) using matrices M (mapping matrix) and H (cost matrix)
2. Encode Equation 3.6 into matrix A_{eq} and vector B_{eq} to enforce the linear equality problem constraints

Note that our problem does not have any linear inequality constraints, therefore A and b are empty for our case.

Linear Equality Constraints

To understand how we solve the challenges highlighted in the previous section, let us start with an example of a swarm of 2 agents.

The mapping matrix for this swarm would look like:

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

such that $\forall i, j \; m_{i,j} \in \{0, 1\}$. This means that $m_{i,j}$ should only hold binary values. If $m_{ij} = 1$, then the agent at i^{th} starting location would end up at the j^{th} goal location. MATLAB's `intlinprog` would determine these values, therefore the elements of the mapping matrix M would serve as the elements for the solution vector x in Equation 3.9.

The cost matrix for this swarm would look like:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

$h_{i,j}$ values will be calculated based on the problem description. In our case, each h_{ij} value is the Euclidean distance between points associated with cells C_i and C_j.

To solve the first challenge, the cost expression shown in Equation 3.9 can be written as:

$$f^T x = \forall i \sum_j m_{ij} \cdot h_{ij} = m_{11} \cdot h_{11} + m_{12} \cdot h_{12} + m_{21} \cdot h_{21} + m_{22} \cdot h_{22} \tag{3.10}$$

MATLAB's MILP optimizer would try to minimize the result of Equation 3.10. From this equation, we can explicitly write f^T and x as:

$$f = \begin{bmatrix} h_{11} \\ h_{21} \\ h_{12} \\ h_{22} \end{bmatrix} \tag{3.11}$$

$$x = \begin{bmatrix} m_{11} \\ m_{21} \\ m_{12} \\ m_{22} \end{bmatrix} \tag{3.12}$$

For a swarm with 's' start locations and 'g' goal locations, the generic relation between f^T and H, and x and M can be written as:

$$f = \begin{bmatrix} h_{11} \\ \vdots \\ h_{1g} \\ h_{21} \\ \vdots \\ h_{2g} \\ \vdots \\ \vdots \\ \vdots \\ h_{s1} \\ \vdots \\ h_{sg} \end{bmatrix} \tag{3.13}$$

$$x = \begin{bmatrix} m_{11} \\ \vdots \\ m_{1g} \\ m_{21} \\ \vdots \\ m_{2g} \\ \vdots \\ \vdots \\ \vdots \\ m_{s1} \\ \vdots \\ m_{sg} \end{bmatrix} \tag{3.14}$$

Equation 3.13 shows that f is constructed by concatenating the transpose of all rows of matrix H. Similarly, equation 3.14 shows that x is constructed by concatenating the transpose of all rows of matrix M.

To tackle the second challenge, 1-1 mapping should be maintained by ensuring that there should only be a single "1" in each row and column of matrix M. For example, if $m_{11} = 1$, then $m_{12} = 0$ and $m_{21} = 0$. Similarly, if $m_{12} = 1$, then $m_{11} = 0$ and $m_{22} = 0$ and so on. Essentially, all rows and columns of M should be linearly independent. This criteria can be ensured through the following linear equation:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} m_{11} \\ m_{12} \\ m_{21} \\ m_{22} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{3.15}$$

Equation 3.15 shows that for any two pairs of mappings in a row or a column of M, only one of them can hold a value of 1. Note that Equation 3.15 can be rewritten as:

$$A_{\text{eq}} \cdot x = b_{\text{eq}}$$

where:

$$A_{\text{eq}} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \text{ and } b_{\text{eq}} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{3.16}$$

Similarly, A_{eq} matrix and b_{eq} vector for a swarm of size 3 (three starting and goal locations each) would be:

$$A_{\text{eq}} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \text{ and } b_{\text{eq}} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{3.17}$$

Comparing 3.16 and 3.17 reveals a pattern for automatically creating A_{eq} and b_{eq} based on the swarm size. b_{eq} is just a column vector of ones with length of n^2, where n is the swarm size. A_{eq} is a matrix of size $2n \times n^2$. The upper half portion of A_{eq} can be constructed using row vectors of ones of length n, where the row vector in row i starts at column $j = i$. The lower half portion of A_{eq} is constructed from n horizontally concatenated identity matrices of order n (size $n \times n$). In general, the following algorithm can be used to create A_{eq} for a swarm of size n:

```
function Aeq = generateAeq(n)
    Aeq = zeros(n*2, n^2);
    for i = 1:n
        cols = 1 + n*(i-1) : n*i;
        Aeq(i, cols) = ones(1, n);
        bottomRows = n+1 : *2; % for bottom half
        Aeq(bottomRows, cols) = eye(n); % identity matrix
           of order 'n'
    end
end
```

Adding Value Constraints

Since our problem is integer-valued, `intcon` will be a vector of ones of length n, where n is the swarm size. Moreover, solution vector x can only take on binary values 0 and 1 for our problem. Therefore, lb will be a vector of zeros of length n, and ub will be a vector of ones of length n. Using MATLAB's syntax, these value constraints can be summarized up as follows:

$$\text{intcon} = \texttt{1:n}$$

$$u_b = \texttt{zeros(1, n)}$$

$$l_b = \texttt{ones(1, n)}$$

3.4 Programming Interface

In this section, we introduce a programming interface that models the problem formulation for formation creation, formation manipulation and formation assignment. We introduce data types that define how to construct a formation and functions that enable creating motion for an entire formation.

3.4.1 Data Types

Position, Orientation and Pose

In order to construct a cloud and a formation, a user should be able to locate each robot in space. The `Pose` data type helps define where the robot is located in world coordinates. `Pose` can be further decomposed into `Position` and `Orientation` data types. The `Position` data type has 3 components: x, y and z. It helps determine the position of an agent in $\mathbb{R}^3$. Each component is a value of type `double` [5]. Similarly, an `Orientation` data type has three components of type `double` - roll, pitch and yaw which determine the orientation of an agent in $\mathbb{R}^3$. Tables A.4, A.2 and A.3 provide an overview of the `Pose`, `Position` and `Orientation` classes respectively.

Cloud, Formation and Path

A `Cloud` data type is defined as an array of `Pose`s, where each `Pose` locates an agent in the shape. A `Cloud` can be created using any number of techniques as long as the `Pose` of the agents can be determined. For example, the following script shows how the square cloud in Figure 3.1 is created:

```
%% create a square shaped cloud
pose1 = Pose(Position(3, 2, 0), Orientation(0, 0, 0));
pose2 = Pose(Position(3, 7, 0), Orientation(0, 0, 0));
```

```
pose3 = Pose(Position(9, 7, 0), Orientation(0, 0, 0));
pose4 = Pose(Position(9, 2, 0), Orientation(0, 0, 0));

square_cloud = Cloud([pose1 pose2 pose3 pose4]);
```

Alternatively, a cloud can be created graphically by using the `graphicalPositionInput` data type (discussed below). In addition to these two methods, users can use one of many MATLAB toolboxes to determine agent poses, including the computer vision toolbox to determine agent locations within a shape, or the robotics toolbox to determine ground truth values of poses in a real experiment. Table A.5 provides an overview of the `Cloud` class.

A `Formation` data type helps create a formation by storing all attributes described in Section 3.1. Following are all the attributes of a formation, of which only the first four are user provided:

1. **Cloud:** A `Cloud` determines the shape of a formation.

2. **Agent IDs:** Agent IDs help in identifying agents in a cloud, and correspond to each `Pose` in the `Cloud`.

3. **Agent diameter:** Agent diameter specifies the circular or spherical footprint of all agents in a formation (assuming a homogeneous formation).

4. **Orientation:** This attribute determines the orientation of the formation shape in global coordinates. It helps in defining where a formation 'points', and is useful to align the formation in a particular direction.

5. **Diameter:** A formation diameter is the Euclidean distance between the centers of two agents in the formation which are farthest away from each other. It is calculated from the `Cloud` attribute. A formation diameter describes the

'size' of a formation, which is helpful in describing the circular footprint of the entire formation for path planning (see Subsection 3.4.2).

6. **Centroid:** The formation centroid is the center of mass of a formation and is calculated using the `Cloud` attribute. The centroid is essential to describe how motion primitives are applied to a formation (see Equations 3.2, 3.3 and 3.4), as well as for determining the goal pose for a formation in path planning (see Section 3.4.2 and Figure 4.17).

An overview of the `Formation` class is given in Table A.6. Figure 3.7 shows a formation of four agents. The orientation of the formation is shown in red with respect to the world coordinates.

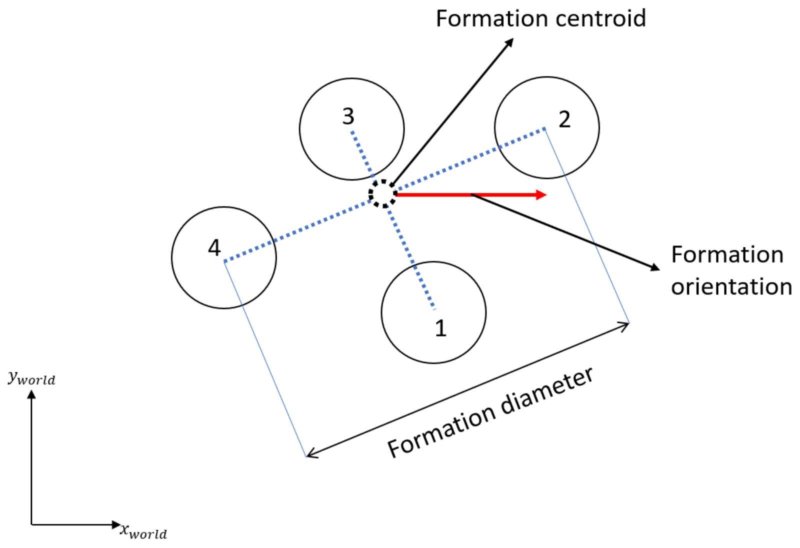


Figure 3.7: Example of a 2D formation of 4 agents.

The following scripts shows how a formation can be created for a square-shaped cloud:

```
%% create a formation from a cloud
```

```
orientation = Orientation(0, 0, 0);
agent_diameter = 0.14;
id_order = [2 5 1 6];
square_formation = Formation(square_cloud, orientation,
   agent_diameter, id_order);
```

The four default motion primitives are implemented as functions of the `Formation` class. Continuing the square formation example above, the following script shows how the default motion primitives can be applied to the square formation:

```
%% use motion primitives with a formation
map = buildMap(false); % custom function to create empty 5x5 map

% translate
translation_vector = [1, 1, 0];
translated_formation = square_formation.translate(
   translation_vector, map);

% rotation
angle = pi/4;
rotated_formation = square_formation.rotate(angle, map);

% scaling
scale_factor = 1.1;
scaled_formation = square_formation.scale(scale_factor, map);

% shearing
shear_factor = 1.1;
sheared_formation = square_formation.shear(shear_factor, map);
```

The results of this script are represented in Figures 3.2, 3.3, 3.4, and 3.5. Note that `map` is a variable of type `binaryOccupancyMap` [6]. `binaryOccupancyMap` is part of the MATLAB's navigation toolbox [7]. This `map` allows a `Formation` to ensure that applying a motion primitive would not render any agent in an occupied portion of the environment. As shown in the code sample above, we use a custom function to create a map that models the environment we used for experimental evaluation. A user can specify their own `binaryOccupancyMap` that represents occupancy information of their environment. Table A.6 summarizes the `Formation` API.

While a user can create new formations by applying motion primitives, they need to express how a robot moves from one formation to another. The `Path` data type describes motion of robots through different `Formations`. It is defined as an array of `Formations` where each `Formation` is constructed using the same group of robots. Figure 3.8 shows an example of a path for a group of 5 robots starting in a pentagon formation, transitioning to a square formation with a center, and finally ending up in a 'vee' formation:

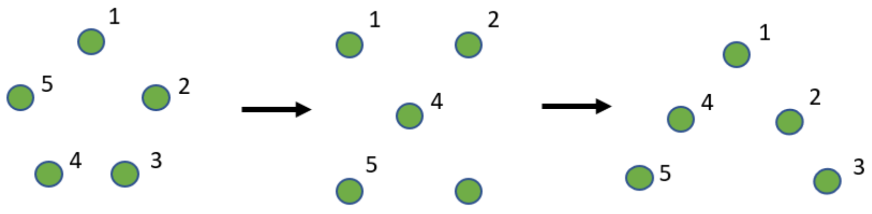


Figure 3.8: A path of 3 formations for a group of 5 robots

The following code sample shows how a path can be created for the scenario shown in Figure 3.8:

```
path = Path([pentagon_formation, square_formation, vee_formation]);
```

A path can also be created using the `planPath` function which is discussed in the next section. Table A.7 gives an overview of the `Path` data type.

Graphically selecting positions: graphicalPositionInput

As an alternative to creating clouds programmatically, the `graphicalPositionInput` data type can be used to select points on a given `binaryOccupancyMap`. In addition to creating clouds, it allows a user to choose a goal `Pose` for a formation to move to (see 4.1.4). The `graphicalPositionInput` constructor requires a map of the environment, the number of agents in the new cloud to be created, and (optionally) a cloud representing where the agents currently are in the map. The following code snippet shows how a `graphicalPositionInput` is used to first obtain a cloud:

```
% Define where robots currently are
pose1 = Pose(Position(2, 1, 0), Orientation(0, 0, 0));
pose2 = Pose(Position(1, 2, 0), Orientation(0, 0, 0));
pose3 = Pose(Position(1, 1, 0), Orientation(0, 0, 0));
pose4 = Pose(Position(2, 2, 0), Orientation(0, 0, 0));
current_cloud = Cloud([pose1 pose2 pose3 pose4]);

% create gpi to select 4 points
gpi = graphicalPositionInput(map, 4, square_cloud);

% call start method to create a cloud of 4 agents graphically
new_cloud = gpi.start();
```

Figure 3.9 shows the the graphical interface brought up using `start` method. The user is selecting a new cloud based on where the robots are currently are in the map.

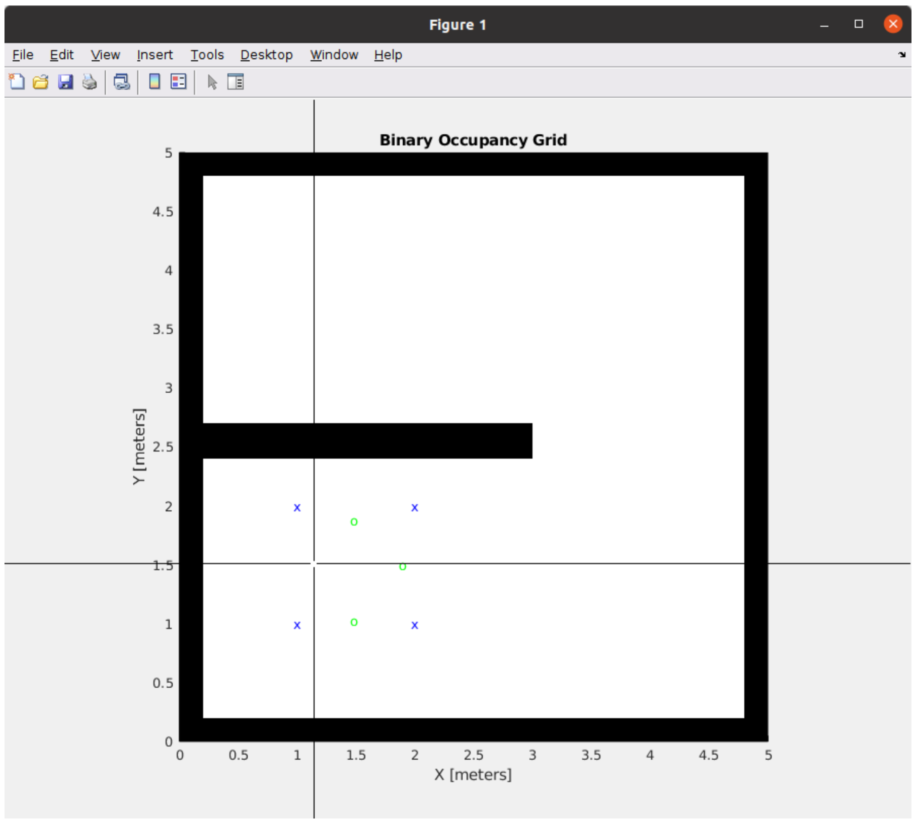


Figure 3.9: Selecting agent positions for a cloud. The blue ’x’s are current positions of the robots (if known), and the green ’o’s are the newly selected agent positions for the cloud. Dark portions of the binary occupancy grid represent obstacles (walls).

The newly obtained cloud can now be used to create a formation and then select a goal pose for it to move to. The objective here is for the centroid of the formation to align with the goal. The following code sample shows this process:

```
% use new_cloud to create a new formation
agent_diameter = 0.14;
id_order = [1 2 3 4];
new_formation = Formation(new_cloud, Orientation(0, 0, 0),
    agent_diameter, id_order);

% call goal method to get a goal pose for the newly created
    formation
```

```
goal_pose = gpi.goal(new_formation);
```

Figure 3.10 shows a user selecting a goal pose for a formation using the `goal` method. Notice that the map has been 'inflated': the occupancy of the walls is exaggerated to help the user select a 'safe' goal for the formation. This is done by increasing the dimensions of each wall by the radius of the formation (shown in green circles in Figure 3.10).

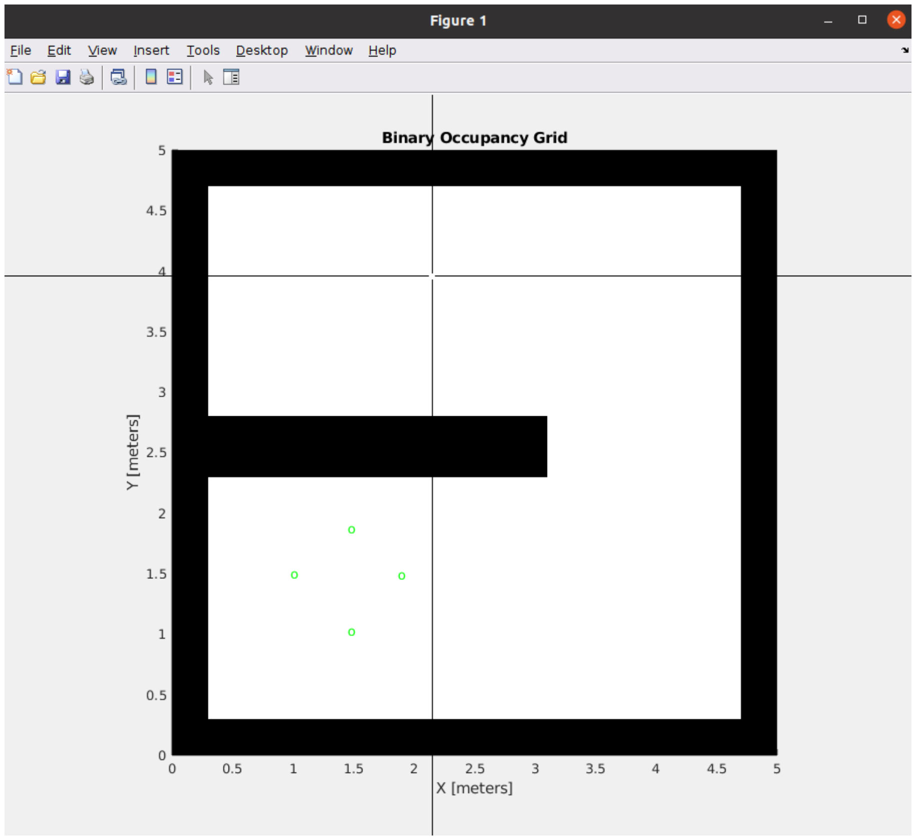


Figure 3.10: Selecting goal for a formation. The green 'o's show the current position of agents in the formation.

After obtaining a goal, the `planPath` function can be used to create a collision-free path for the formation. Table A.8 lists the API for the `graphicalPositionInput` class.

3.4.2 Functions

assign

The `assign` function enables a user to change the shape of a formation. The user provides a starting formation which represents how the robots are currently arranged, and a goal cloud which represents the new shape for the robots to form. The assign function uses the goal assignment problem formulation described in Section 3.3. The script below describes how a user would change a square formation to a 'line' formation, as shown in Figure 3.11.

```
line_cloud = Cloud([pose1 pose2 pose3 pose4]);
line_formation = assign(map, square_formation, line_cloud);
```

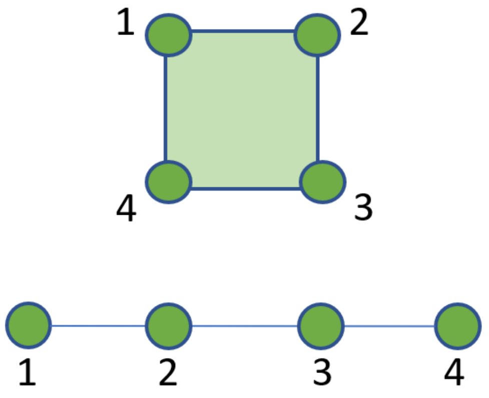


Figure 3.11: A square formation and line formation for the same group of robots.

planPath

The `planPath` function calculates a collision-free path for a given formation and goal pose in a `binaryOccupancyMap`. It assumes that the formation is rigid. A user can provide a function handle to the planner of their choice. This planner should accept a start `Pose`, goal `Pose`, and a `binaryOccupancyMap`. This planner should return a nx2 double matrix, where n is the swarm size and the first and second columns represent x and y components of the waypoints along the path respectively. A high level description of the planning process is described below:

1. **Map inflation:** The formation is assumed to be rigid, and can therefore be treated as a large mobile robot. This large robot would share the same circular footprint as the formation, which can be determined using the formation diameter. The given `binaryOccupancyMap` can then be inflated by the formation radius, which allows a path planner to calculate a collision-free path using only the centroid of the formation.

2. **Call planning algorithm:** The user-provided planning algorithm uses the start pose, the goal pose and the inflated map to calculate the waypoints. The start pose is the centroid of the given formation. Following the scenario depicted in Figure 3.9 and 3.10, Figure 3.12 shows an inflated map with frontier nodes and edges (blue) and the most optimal path (red) from start to goal. In this particular case, we use MATLAB's PRM planner [8] which is part of the Navigation Toolbox.

3. **Generate translated formations:** The waypoints returned by the planning algorithm represent the path that the formation's centroid should follow. To determine the formation at each successive waypoint, `planPath` simply calls the translate primitive to get a new formation at the next waypoint. Here, the translation vector is generated by subtracting the position vector of the current waypoint from that of the next waypoint in global coordinates. The translation operation is shown in Equation 3.2.

4. **Return Path:** The formations calculated in the previous step are stored in an array. This array is used to create a `Path`.

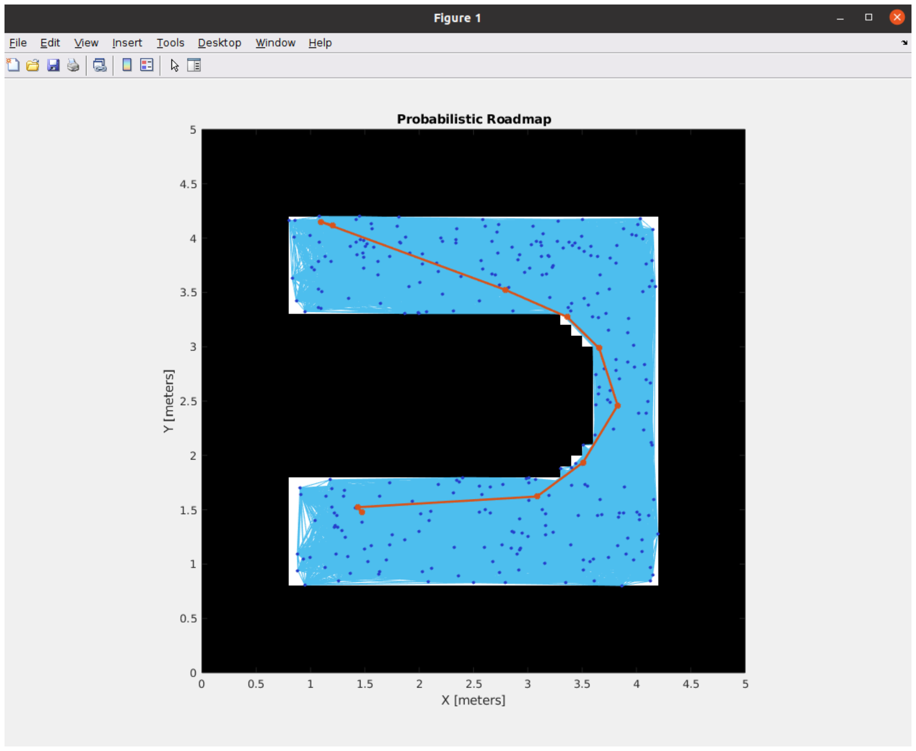


Figure 3.12: Output of planPath function. The blue nodes and edges show sample space created by exploring the given binaryOccupancyMap. The optimal path is shown in red.

follow

The `follow` function enables robots to execute motion specified through a `path` object. Figure 3.13 shows how the follow function executes robot motion for a `path`.

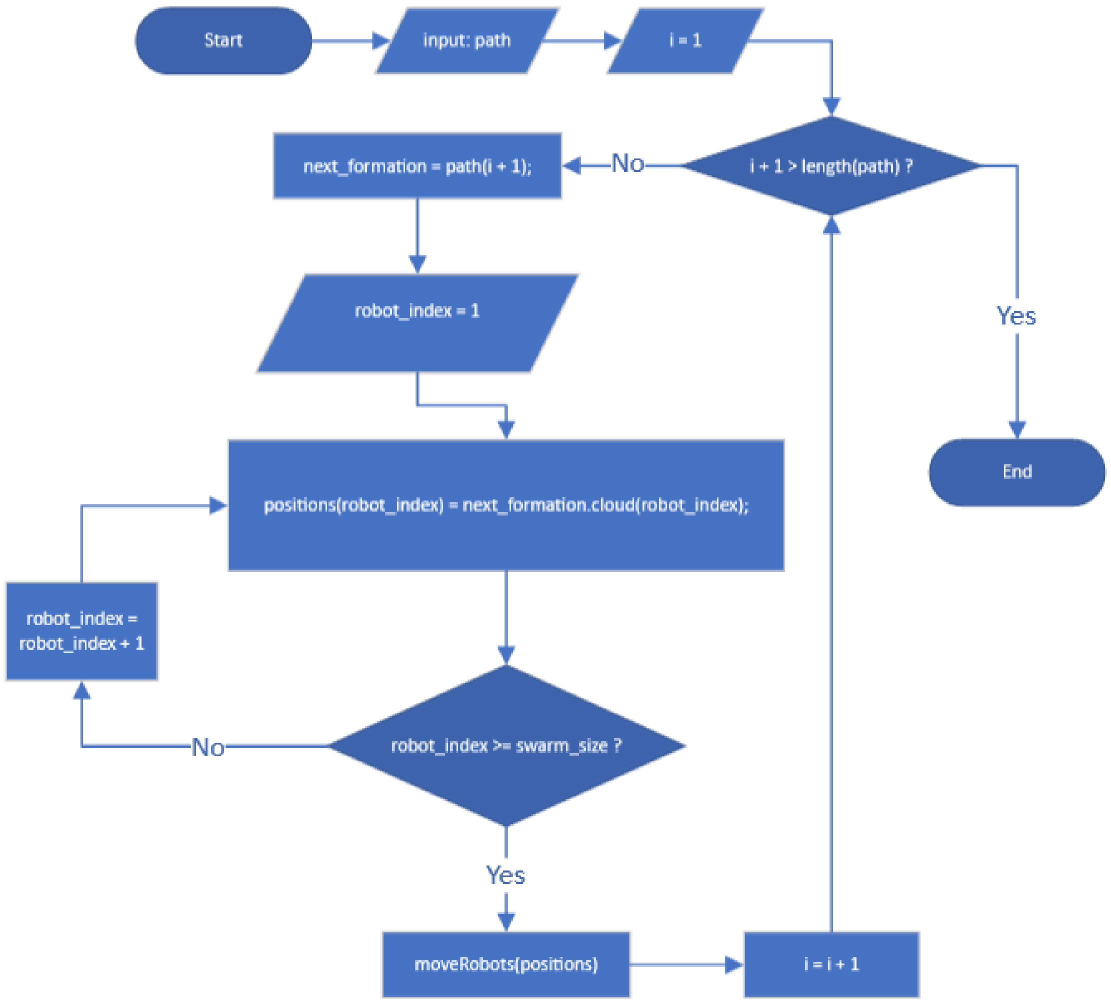

Start
input: path
i = 1
i + 1 > length(path) ?
No
Yes
next_formation = path(i + 1);
robot_index = 1
End
positions(robot_index) = next_formation.cloud(robot_index);
robot_index = robot_index + 1
No
robot_index >= swarm_size ?
Yes
moveRobots(positions)
i = i + 1

Figure 3.13: Overview of how follow function executes robot motion

Starting with the first formation in the `path`, `follow` extracts what position each robot should navigate to in order to form the next formation. These positions are are stored in the '`positions`' matrix. The `moveRobots` function uses the navigation strategy described in Section 3.6 to move each robot to the next formation. This process is repeated until the robots reach the last formation in `path`. Source code for `follow` and `moveRobots` is provided in Appendix B.

Table 3.1 gives an overview of the three functions.

function	**Description**
`assign(Formation, Cloud, binaryOccupancyMap):Formation`	Assigns positions to agents in starting formation to positions in cloud based on a given binary occupancy map. Returns the assignment in the form of a new formation.
`planPath(Formation, Pose, binaryOccupancyMap, handle):Path`	Returns path from a starting Formation to a goal Pose for a given binaryOccupancyMap. It uses user-provided function handle.
`follow(path):int`	Executes robot motion to follow given path. Returns 1 if all robots reached their goal. Returns 0 otherwise.

Table 3.1: Overview of functions

3.5 Software Platforms

The programming interface introduced in Section 3.4 is implemented on top of two existing software platforms: MATLAB and Buzz. Additionally, we use the ARGoS multi-physics simulator for experimental evaluation, which is further explained in section 4.1.1. Figure 3.14 shows an overview of the software platforms used in this work.

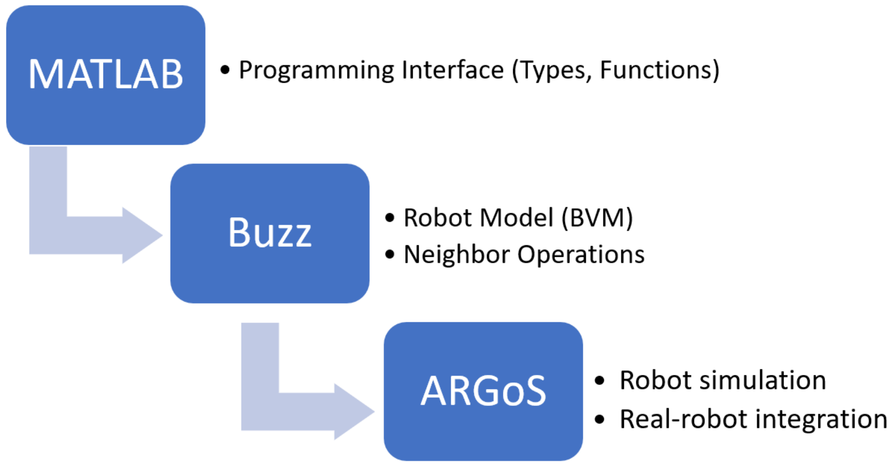


Figure 3.14: Overview of the software platforms used in this work

MATLAB's ability to manage data through matrices allows a user to intuitively represent many physical quantities often used in robotics, such as position (x, y, z), orientation (roll, pitch, yaw), pose (position, orientation) and rotation/transformation matrices. We take advantage of this ability to express and modify robot swarm properties. MATLAB also provides a broad range of engineering toolboxes which offer solutions for multidisciplinary engineering problems [7] [9]. We harness MATLAB's Navigation toolbox to enable occupancy checking and path planning.

Buzz is a dynamically typed domain-specific programming language for heterogeneous swarms [25]. It abstracts low-level setup and implementation details required for modelling robot swarms. We rely on Buzz to handle neighbor data management. Additionally, Buzz is well integrated with ARGoS which offers excellent support for simulated and real-robot experiments. The following sections cover features of Buzz relevant to our work. We also briefly touch on integration of MATLAB and Buzz that allows us to access Buzz features through a MATLAB programming environment.

3.5.1 Buzz Features

Robots in Buzz are modelled as BVMs (Buzz Virtual Machines). A BVM processes sensor readings and actuator values for robots in a simulation or real-robot experiment. BVMs also store neighbor data that includes information such as pose of each neighbor with respect to a robot. The two main features of Buzz that make it useful for our software architecture are:

1. **Situated communication and neighborhood operations:** Situated communication [30] allows each robot to talk to other robots in its vicinity using line-of-sight communication. BuzzVMs store neighborhood data received through situated communication in the form of a dictionary. This data includes `distance`, `azimuth`, and `elevation`. Each data value is identified by the id of the corresponding neighbor. Here `distance` is the line-of-site distance, `azimuth` is the azimuth angle and `elevation` is z-direction distance between a robot and its neighbor. As of this writing, our work only requires `distance` and `azimuth` to determine relative pose of each neighbor from a robot's coordinate frame.

2. **Extendibility:** BuzzVM is implemented in C and can be controlled through Buzz functions (native closures) or external C functions (C closures) [25]. This allows developers to integrate it with external environments, such as MATLAB C++ MEX interface [4] or ROS.

3.5.2 MATLAB-Buzz integration

Integration of MATLAB with Buzz allows our programming interface to exploit the neighborhood operations provided through Buzz. This is accomplished through MATLAB's C++ MEX functions. MEX stands for MATLAB executable, and is essentially a MATLAB function that can be called from a MATLAB script. MEX functions are compatible with C++ 11.

In the MATLAB-Buzz interface, BuzzVMs run in parallel and communicate with MATLAB through MEX functions. This communication topology is represented Figure 3.15. The extensible nature of Buzz allows registering external functions to a BuzzVM through C closures (Section III-A in [25]). Using MATLAB MEX functions as C closures, the custom command '`buzz_do`' (created as part of the MATLAB-Buzz interface) allows a user to handle BuzzVMs from a MATLAB script.

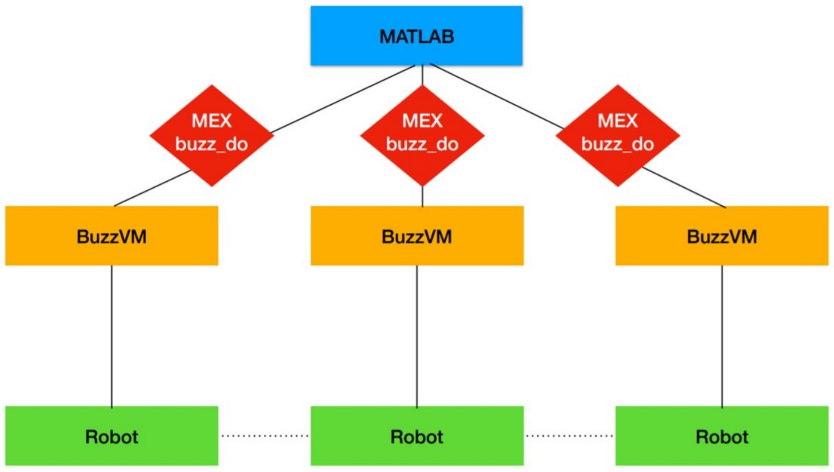


Figure 3.15: Communication topology of the MATLAB-Buzz interface

Having access to BuzzVMs allows users to determine the robot's current state, as well as access its neighborhood data. The MATLAB-BUZZ interface functions that allow our interface to access this data are listed in Table 3.2 below:

Function	Description
`set(key, val)`	set the value of the given `key` in the BuzzVM table to `val`
`get(key)`	return the value of `key` from the BuzzVM table
`set_leds(r, g, b)`	set the color of the LEDs on the Khepera robot to (r, g, b)
`set_wheels(left, right)`	set the wheel speeds on the Khepera robot to given speeds `left` and `right`
`goto(x, y)`	generate low-level propulsion/steering commands to navigate to (x, y)

Table 3.2: MATLAB-Buzz interface functions used to access BVM data

3.6 Navigation and control

Until now, we have described how a user can programmatically define motion for formations. In order to successfully execute this motion, each robot in a formation needs to drive to it's assigned goal while avoiding collisions with other robots. A successful navigation function takes this into account to produce a heading for the robot. This heading should feed into a control strategy to provide low-level inputs to steer and drive the robot.

In the following section, we describe implementation of a potential field navigation strategy that generates a control input for a unicycle model of a robot. This input is used by the `goto` function in buzz to generate low-level control commands for a differential drive robot. This process is shown in Figure 3.16.

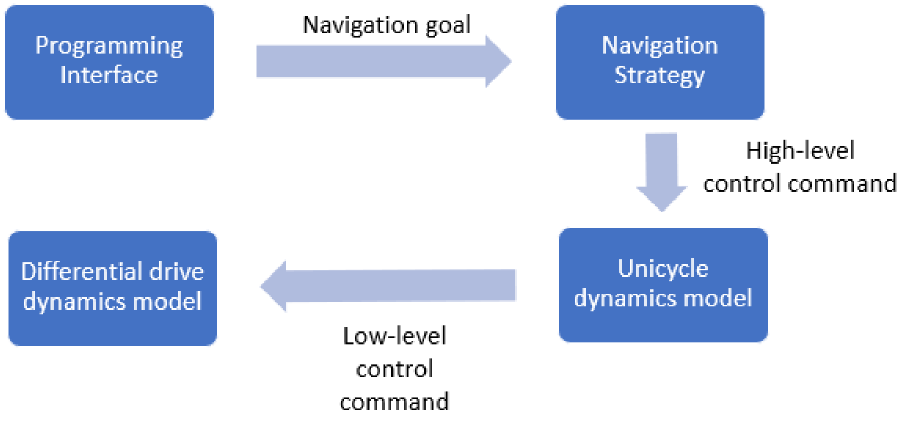

Programming Interface
Navigation goal
Navigation Strategy
High-level control command
Unicycle dynamics model
Low-level control command
Differential drive dynamics model

Figure 3.16: Overview of navigation and control

3.6.1 Potential Field Navigation

The potential field navigation approach models attractive and repulsive forces to drive each robot to its respective goal while avoiding other robots. In this approach, entities with opposite polarities attract each other while those with same polarities repulse each other. Robot-goal pairs are modelled as entities with opposite polarities, while robot-robot pairs are modelled with same polarity.

We model attractive and repulsive forces as functions of distances between robot-goal and robot-robot pairs respectively, where each robot's pose is determined using a global positioning method. The combined effect of attraction and repulsion can be used to produce a heading for each robot. The following sections explain how net attractive and repulsive forces are calculated for each robot.

Net Repulsive Force

Repulsive force is a function of the distance between a robot and an entity. In order to repel an entity, repulsive force should increase inversely with respect to this distance. We compared two candidate functions for modelling this behavior: $\frac{1}{x^2}$ and $\frac{1}{x}$, where x is the distance between the robots. Figure 3.17 compares what the force looks like for each function over an identical range of agent-entity distance:

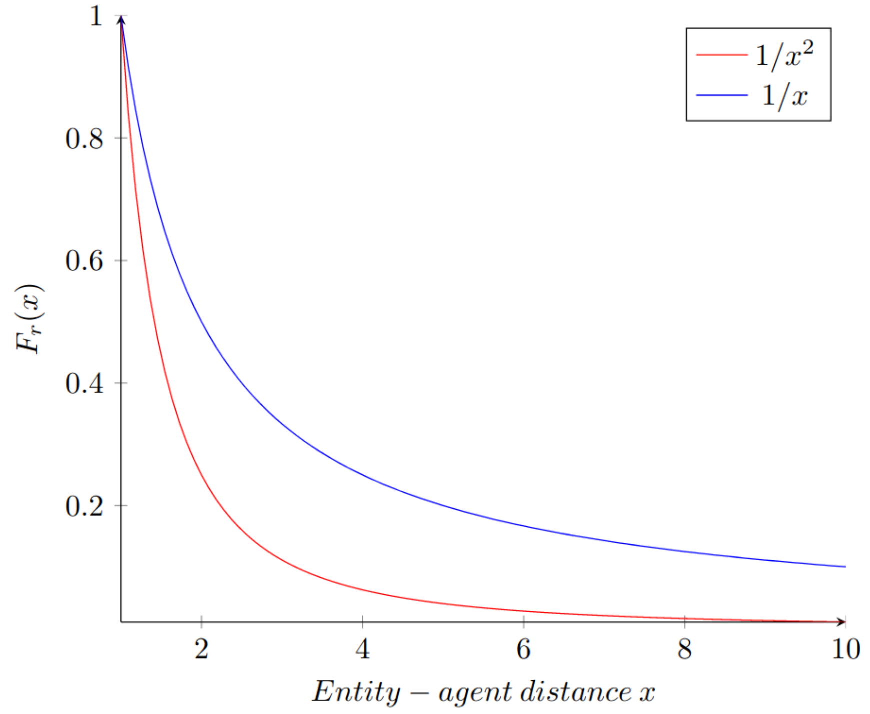


Figure 3.17: Comparison of two candidate functions for representing repulsive force

As seen in 3.17, the response of $\frac{1}{dist_r^2}$ model is more localized than that of $\frac{1}{dist_r}$. We chose $\frac{1}{dist_r^2}$ to ensure that robots only react to entities that are in their immediate vicinity. Equation 3.18 mathematically describes a repulsive force using this function candidate:

$$F_r(\text{dist}) = -\min(F_{\max}, F_{\max} \cdot \frac{1}{\text{cappedDistance}(\text{dist}_r, r_{\text{robot}}, \text{dist}_{\text{safe}})^2}) \tag{3.18}$$

Here F_r is the net repulsive force acting on a robot, $F_{\max}$ is the maximum allowable force magnitude to be applied to any robot, r_{robot} is the radius of each robot, dist_r is the distance between the robot and repelling entity, and $\text{dist}_{\text{safe}}$ is the 'safe distance' for the robots. Safe distance is the minimum inter-agent distance that the robots are allowed to achieve at any point of time. Safe distance is measured as the line-of-sight distance between centers of each pair of robots. Figure 3.18 shows Robot_2 applying a repulsive force on Robot_1. Note that $\text{dist}_{\text{safe}}$ is marked with a blue boundary around Robot_1. The center of Robot_2 is not allowed to enter this boundary at any time during the experiment.

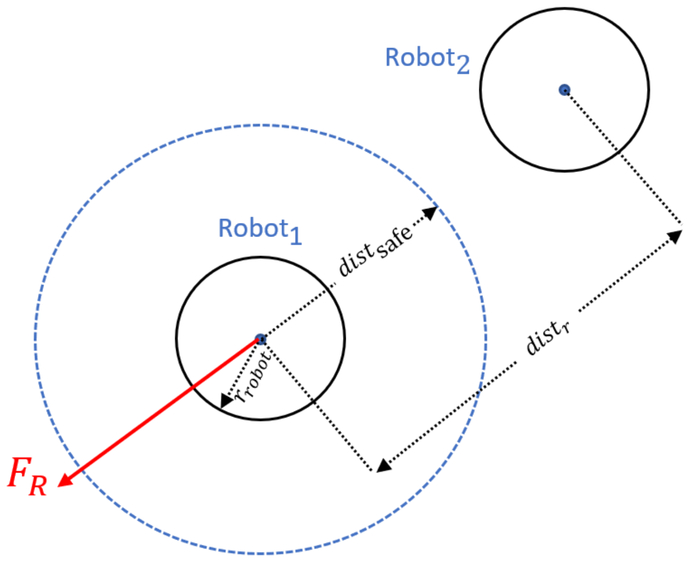


Figure 3.18: Robot2 applying a repulsive force on Robot1

Function `cappedDistance` in Equation 3.18 accounts for r_{robot} and $\text{dist}_{\text{safe}}$ while ensuring that the resulting distance is always positive. `cappedDistance` is implemented programmatically as follows:

```
function final_dist = cappedDistance(dist_r, r_robot, dist_safe)
    final_dist = dist_r - 2*r_robot - dist_safe;
    if final_dist < min_dist
        final_dist = min_dist;
    end
    final_dist = single(final_dist);
end
```

Note that the variable `min_dist` is a parameter which ensures that the `final_dist` value remains positive, otherwise it would lead to a division-by-zero error in Equation 3.18. For this purpose, `min_dist` > 0.

Equation 3.18 calculates the repulsive force for a single entity. To calculate the net repulsive force for each robot, repulsive forces associated with each entity in the robot's vicinity are calculated and converted to Cartesian coordinates. Note that the neighborhood management feature in Buzz [25] tracks neighbor locations in a robot's local coordinates. Therefore, neighbor distance values obtained through Buzz need not be explicitly converted to the robot's local coordinates. The vector sum of these forces in the Cartesian coordinate form is then averaged to obtain the net repulsive force. This process is summarized through the following code block:

```
    %% Repulsive force

    % account for robot diameter while calculating distances wrt
       other
    % robots
    r_robot = 7; % radius of kheperaIV robots (cm)

```

```
  % account for 'safe distance' between two robots to avoid
     collision
  dist_safe = 10; % tunable parameter (cm)

  % Function handle for repuslive force - force should decrease
     with increase in distance
  F_r = @(dist_r) -min(max_force, max_force*(1/cappedDistance(
     dist_r, r_robot, dist_safe)^2));

  % get information about neighbors of this robot
  bvm_state = robots(ridMap(rid)).bvm_state;

% Calculate repulsive forces based on neighbor distance
  if ~ isempty(bvm_state.neighbors)
      % get orientation of all neighbors with respect to this
         robot
      thetas = vertcat(bvm_state.neighbors.azimuth);
      % get line-of-sight distances of this robot from its
         neighbors
      neighborDistances = vertcat(bvm_state.neighbors.distance);
      % calculate position vectors from this robot to neighbors
         in polar form
      rhos = arrayfun(F_r, neighborDistances);

      % convert position vectors to neighbors to Cartesian form
      [r_x, r_y] = pol2cart(thetas, rhos);
      d_x = sum(r_x);
      d_y = sum(r_y);
  end
```

Net Attractive Force

Attractive force exerted on a robot should be directly proportional to the distance between the robot and its goal. We compared two candidate functions to model this force: x and x^2, where x is the distance between the robot's center and its goal. As shown in Figure 3.19, the force value increases more aggressively for x^2. To avoid this issue, We chose to model attractive force with the candidate function x.

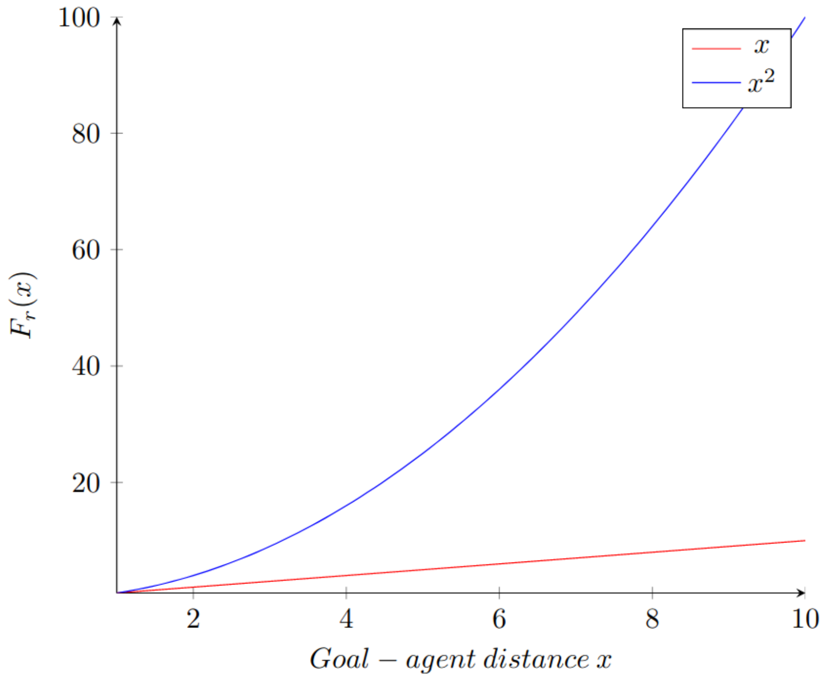


Figure 3.19: Comparison of two models for representing attractive force

Equation 3.19 models the attractive force exerted on a robot by its goal:

$$F_a = \min(F_{\max}, F_{\max} \cdot \mathrm{dist}_a) \tag{3.19}$$

Here, F_a is the force of attraction between the robot and its goal and $dist_a$ is the distance between the agent and the goal. Figure 3.20 shows this scenario.

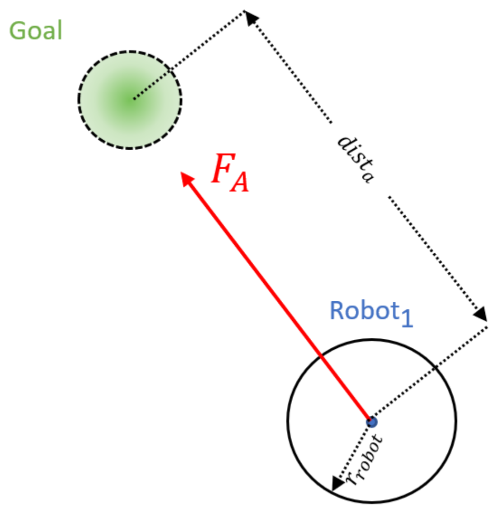


Figure 3.20: Robot1 experiencing attractive force exerted by its goal

Note that the goal pose is known to the system in global coordinates. dist$_a$ is obtained by converting the goal from world frame to local frame as shown in Figure 3.21 below:

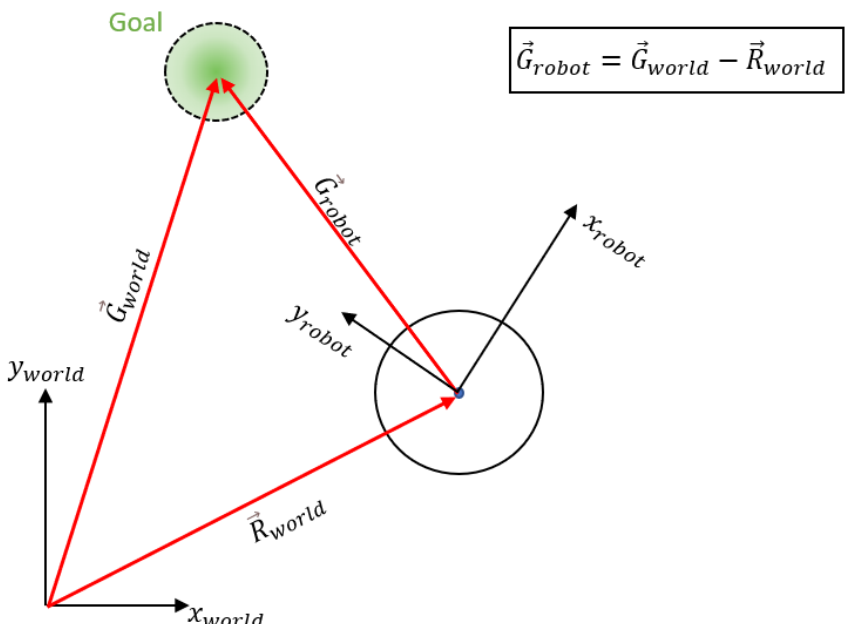


Figure 3.21: Converting a 2D navigation goal from global coordinates to the robot's local coordinates

Assume that the goal location is stored in the `goal` variable with respect to world coordinates. Then, the attractive force can be programmatically determined as follows:

```
% convert goal to robot frame
goal = world2RobotFrame(goal(1), goal(2)).';

% find goal angle with respect to robot x frame
goalAngle = findAngle([1, 0], goal);

% obtain line-of-sight distance between robot and goal
goalDist = pdist([0 0; goal], 'euclidean');

% calculate force of attraction based on line-of-sight distance
    between robot and goal
F_a = min(max_force, max_force*dist_a);
```

```
% convert force vector to Cartesian form
[g_x, g_y] = pol2cart(goalAngle, F_a);
```

The next section illustrates how the repulsive and attractive forces are combined to obtain the net force acting on a robot.

Net Force

The net force on acting on the robot determines the combined effect of repulsion and attraction induced on it by obstacles and the goal, as shown in Figure 3.22.

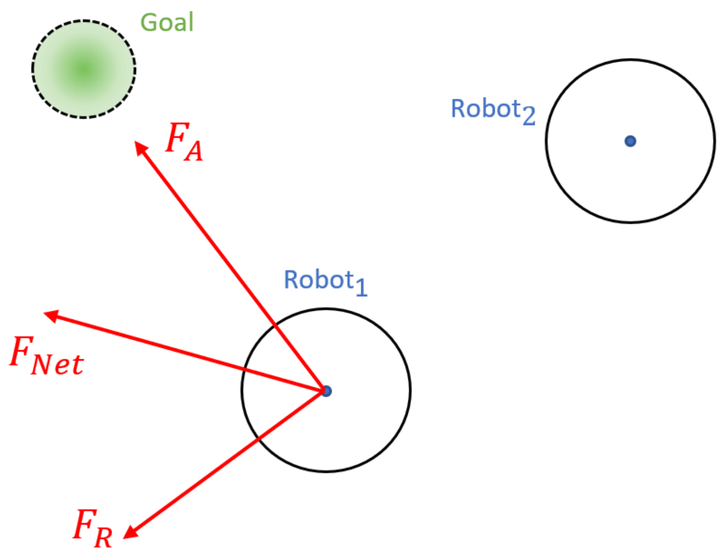


Figure 3.22: Net force acting on Robot1

The net force is simply the vector sum of F_r and F_a. Based on the code samples presented in the previous two sections, the net force is calculated as follows:

```
% Vector addition of attractive and repulsive forces
f_x = d_x + g_x;
```

```
    f_y = d_y + g_y;


    total_force = [f_x f_y];
```

The `total_force` variable is then returned from the parent function and passed along to the low-level controller (`goto` function) in order to steer and propel the robot accordingly. A complete force navigation function which calculates repulsive, attractive and net force is listed in appendix C.

3.6.2 Buzz goto Function

By default, Buzz does not provide an implementation for a low-level control function which converts navigation goals to propulsion commands for the robot. This is done to ensure that Buzz does not conform to a specific robot type. In Buzz, users are expected to call the `goto` function which takes in a 2D heading in the form of x and y values. The function then converts this heading to propulsion commands tailored to the robot in question (Section V-A of [25]). Users can implement their own version of `goto` that confirms to this pattern of input/output. Alternatively, if Buzz is being used in conjuction with ARGoS, a user can exploit one of the ARGoS extensions that have `goto` implemented for robots such as Kilobot, Footbot and KheperaIV [1].

For our work, we used the Khepera extension for ARGoS [2]. One integral part of the goto() function defined in this implementation is the `SetWheelsSpeedsFromVector` function [3]. This function essentially models a proportional controller for differential drive robots [28].

The `goto` function accepts a 2D navigation goal in the robot's local coordinate frame. This goal is expressed as a heading vector from the origin of the robot's local frame to the goal (see Figure 3.21). The `SetWheelSpeedsFromVector` function then calculates a counter-clockwise heading angle starting from the robot's orientation (x-axis) and leading to the heading vector. Then, a wheel speed magnitude V is calculated based on the length of $\vec{G_{robot}}$ or a maximum wheel speed value V_{max}, whichever is lower. V is proportionally split between the left and the right wheel speeds (V_L and V_R respectively) based on the heading angle. Figure 3.23 depicts this scenario, where $\vec{G_{robot}}$ is the heading vector. There are three cases for how V_L and V_R are split:

1. **Soft turn:** A soft turn condition is executed when both wheel speeds are positive but one wheel speed has a higher magnitude in order to make the turn. Figure 3.23 depicts this case. The wheel speeds are set in proportion to the angle, where $V_L < V_R$ as the robot needs to turn left to align itself with $\vec{G_{robot}}$.

2. **Hard turn:** A hard turn condition is executed when one wheel speed is positive while the other one is negative. Both wheel speeds are set to V (one negative, one positive) in this case for the turn to complete as quickly as possible.

3. **No turn:** A no turn condition arises when the goal is approximately straight ahead of the robot. In this condition, both wheel speeds are set to positive V.

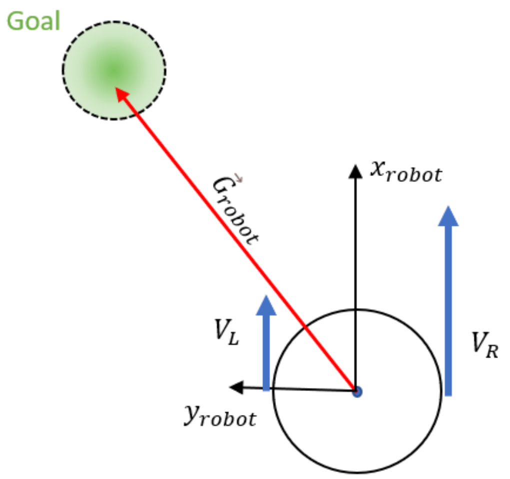


Figure 3.23: Proportional control on a differential drive robot for a 2D navigation goal

Chapter 4

Experimental Evaluation

The purpose of this chapter is to analyze the effectiveness of the interface introduced in Chapter 3. We include demonstrations to show similarity of code execution in simulation and on real robots for key features of this interface. Additionally, we show how our MILP-basd problem formulation for formation assignment performs as the swarm size increases.

4.1 Demonstrations

In this section, we include demonstrations for formation assignment, motion primitives (translate, rotate, scale, shear) and navigation (collision avoidance, collective transport) in simulation and on real robots. For each pair of simulated and real robot demonstration, we first share a programming script that describes the underlying experiment. Then, we share screenshots of simulated and real robot demonstrations that show execution of the script.

We use the ARGoS multi-physics robot simulator to perform demonstrations in simulation. For real robot demonstrations, we use ARGoS in combination with a Vicon motion capture system to track the pose of all robots. ARGoS allows us to instantly switch between simulated and real-robot experiments due to its seamless integration with Buzz and the Vicon motion capture system.

Figure 4.1 shows a high-level diagram of the communication loop between the robots, Vicon motion capture system, ARGoS, and the server that hosts ARGoS. This setup is used for real-robot experiments. ARGoS, Vicon and the KheperaIV robot are further described in Subsection 4.1.1.

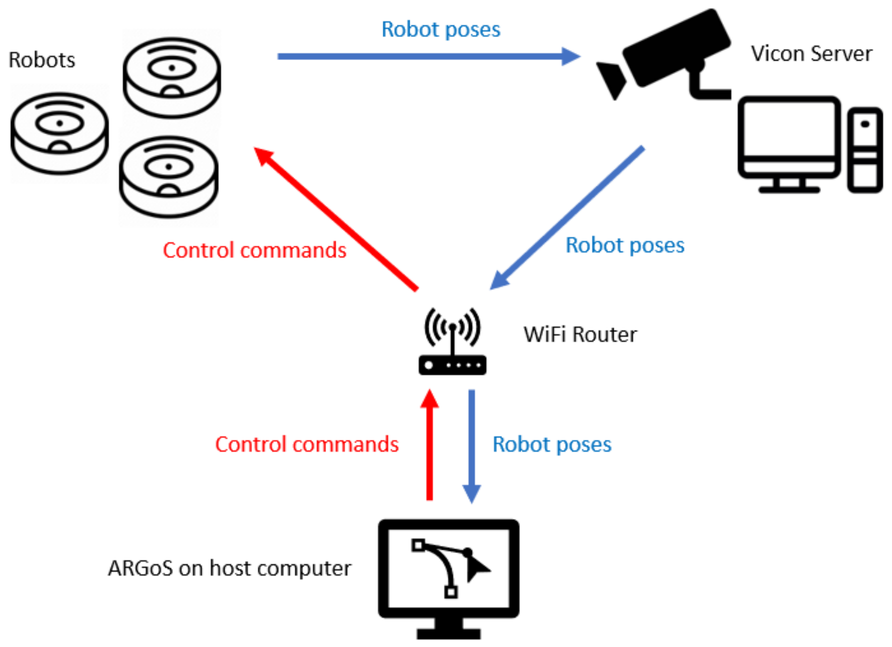


Figure 4.1: High-level communication loop between the robots, Vicon motion capture system and ARGoS

4.1.1 Setup

ARGoS

ARGoS is responsible for reporting pose and sensor data from robots to the MATLAB-Buzz interface, and forwarding control commands from the MATLAB-Buzz interface back to the robots. In simulation, each experiment is conducted in an arena of 5m x 5m in size bordered by walls 0.2m x 0.2m in size (each), as shown in Figure 4.2.

ARGoS configurations for simulated and real robot experiments are almost identical with the only major difference being the physics engine. A physics engine in ARGoS is responsible for updating the pose of all robots in an experiment (see embodied entities in [26]). For simulations, we use the `dyn2d` physics engine. For real robot experiments, we switch to the Vicon physics engine which uses the Vicon motion capture system to provide real time information about robot poses in an experiment arena.

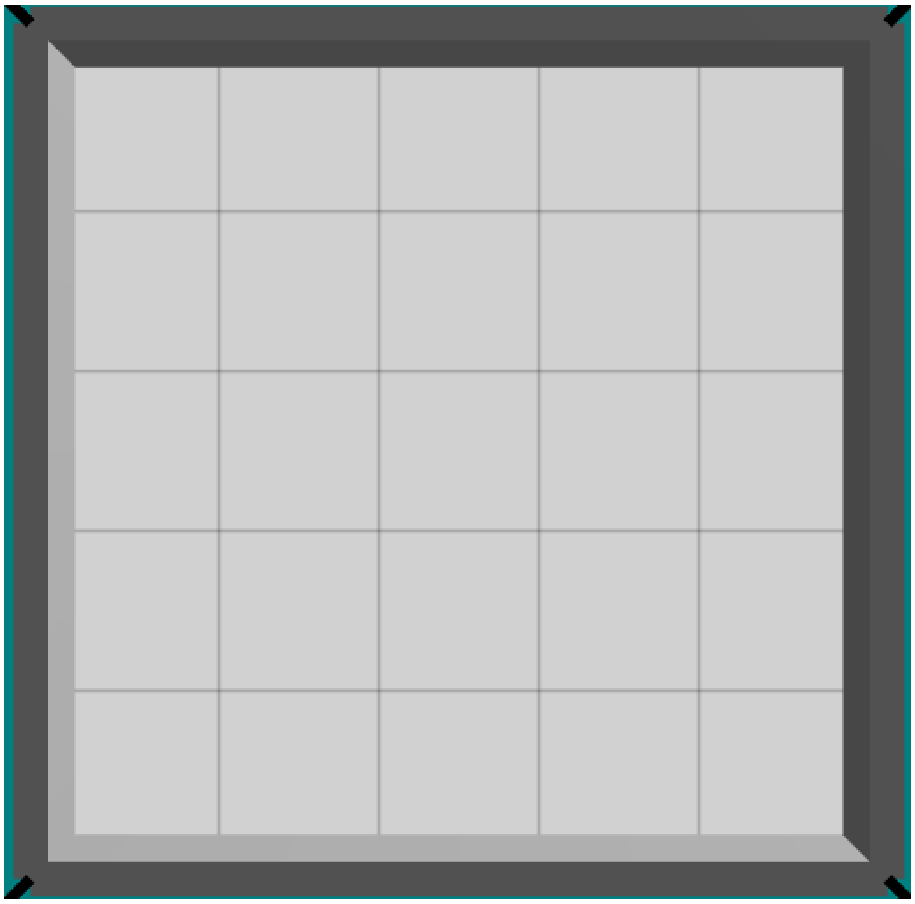


Figure 4.2: Empty 5m x 5m arena in ARGoS with 0.2m x 0.2m walls as borders

Vicon Motion Capture System

The Vicon Motion Capture System allows real-time tracking of selected objects in an experiment arena. Real robot experiments are conducted in a 160 cm x 119 cm (roughly 5.25 ft x 3.90 ft) arena. As part of previous work done at Nest Lab, the Vicon SDK (Software Development Kit) was exploited to be integrated as a plugin for ARGoS. This plugin includes a physics engine which updates spatial information of robots. This information is visualized in real-time through ARGoS' graphical user interface. Our experimental setup includes 10 vicon motion capture cameras monitoring the experimental arena.

KheperaIV Robot

KheperaIV is a differential drive mobile robot with a circular footprint of diameter 14 cm. It includes a variety of sensors for obstacle detection, including proximity (infrared) sensors, ultrasonic sensors, an optional LIDAR module as well as an RGB camera. The robots run on Yocto linux distribution meant for embedded applications. Each robot also has a Wi-Fi module on board, which is used to communicate with ARGoS. The Khepera plugin for ARGoS [2] models the ultrasonic and proximity sensors and includes a controller for steering and propulsion of the robot.

Figure 4.3: KheperaIV mobile robot [1]

1 Source: https://www.k-team.com/khepera-iv

4.1.2 Formation Assignment

The purpose of formation assignment is to rearrange a given distribution of robots into a desired shape. To demonstrate formation assignment, we start with a given formation of robots and assign them to a new formation. The original formation is created before the experiment begins. For simulation, robot positions for this formation are specified through the ARGoS configuration file. For real robot experiments, the robots are manually placed in the arena.

The script below lists the experiment procedure:

```
% 1. declare IDs for robots used in the experiment
robotIDs = [1 2 4 5];

% 2. build map with walls only at corners
map = buildMap(false);

% 3. initialize robots through MATLAB-Buzz interface
initializeRobots(robotIDs);

% 4. create formation from current distribution of robots
current_cloud = getCurrentPositions(robotIDs);
current_formation = Formation(current_cloud, Orientation(0, 0, 0),
   0.14, robotIDs);

% 5. obtain new shape (cloud) graphically from user
gpi = graphicalPositionInput(map, length(robotIDs), current_cloud);
new_cloud = gpi.start();

% 6. assign new formation to current formation of robots
new_formation = assign(map, current_formation, new_cloud);
```

```
% 7. create path to move robots new formation
path = Path([current_formation new_formation]);

% 8. execute robot motion
follow(path);
```

This procedure is explained as follows:

1. The user first declares robot IDs being used for this experiment. These IDs are defined in the ARGoS configuration file for simulation or determined through the IP addresses for real robots.
2. The user provides a map to be used for checking occupancy during formation assignment. The `buildMap()` function returns a `binaryOccupancyMap` of the environment shown in Figures 4.5 and 4.6 below.
3. `initialzeRobots` establishes the communication between MATLAB-Buzz and ARGoS for each robot ID.
4. The `getCurrentPositions()` function queries the Buzz-MATLAB interface to receive robot positions and returns a `Cloud` representing the shape that the robots form. This cloud is used to create an initial formation of the robots.
5. The `graphicalPositionInput` class is used to graphically select a new cloud for the target formation, as shown in Figure 4.4.
6. The `assign` function is used to return a new `Formation` based on the given starting formation and target cloud.
7. A `Path` is created using `current_formation` and `new_formation`.
8. The `follow` function is used to execute robot motion along the `Path` created in the previous step.

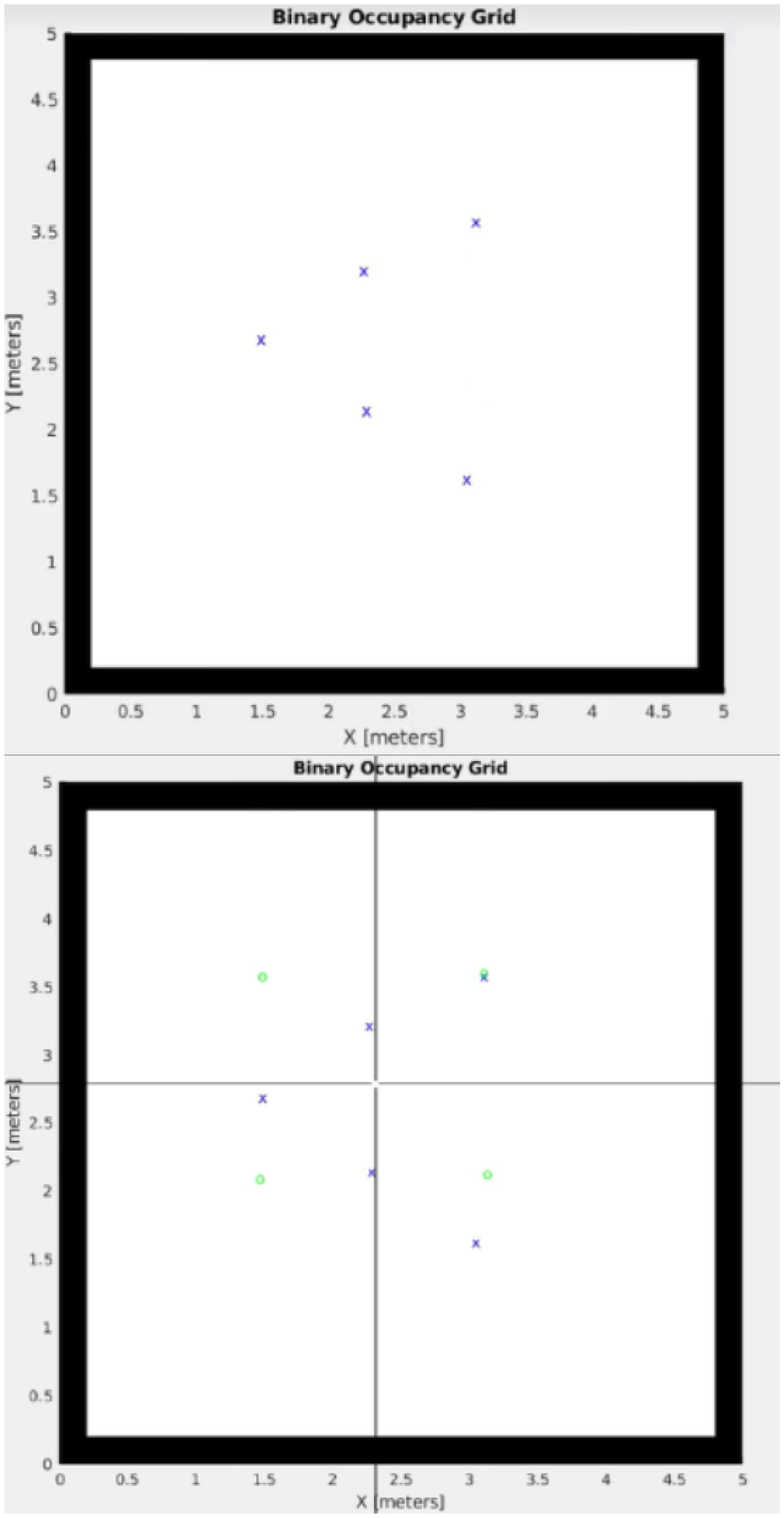


Figure 4.4: GUI for selecting new formation based on where the current formation lies. The blue 'x's represent the current formation, whereas the green circles represent the new formation being selected.

Figures 4.5 and 4.6 show the execution of this script in simulated and real experiments respectively. In Figure 4.6, the dashed blue lines in Frame 1 depict the initial formation of the robots, which are arranged in a '<' shape. Each robot is marked with an ID in Figure 4.6 to track its movement through the frames. Frames 2 and 3 show how each robot travels to form the target shape. Frame 4 shows the target formation, which represents four robots arranged at the corners of a quadrilateral and one robot in the center of the quadrilateral.

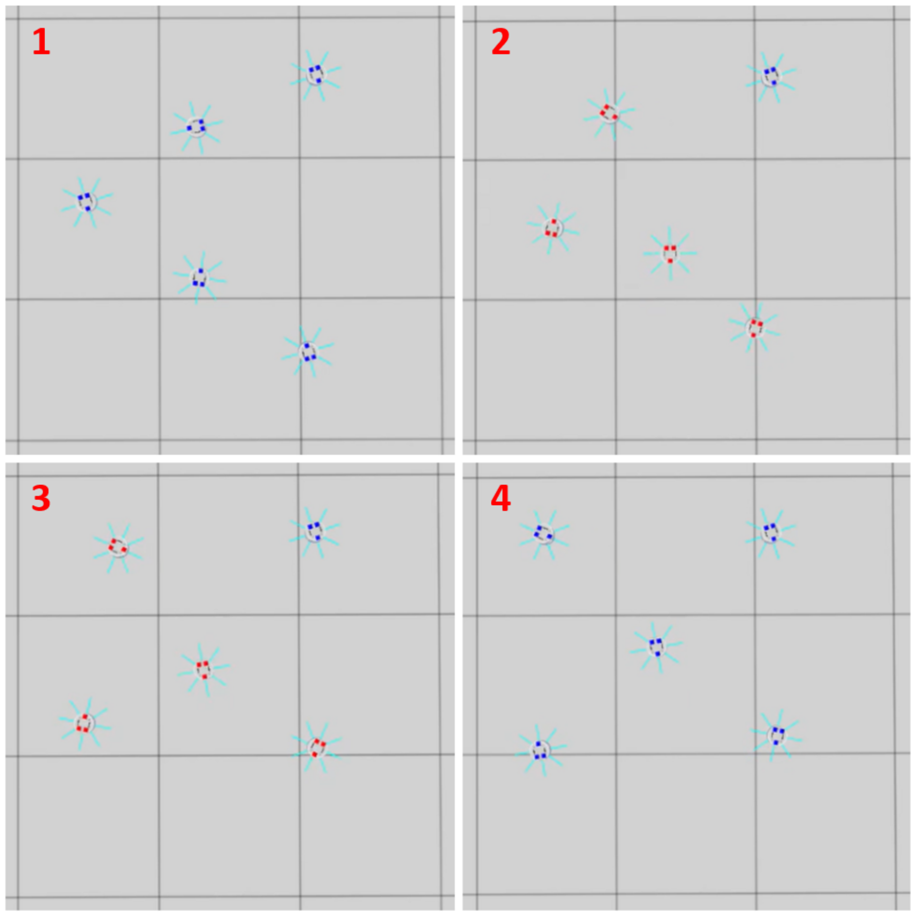


Figure 4.5: Formation assignment in simulation

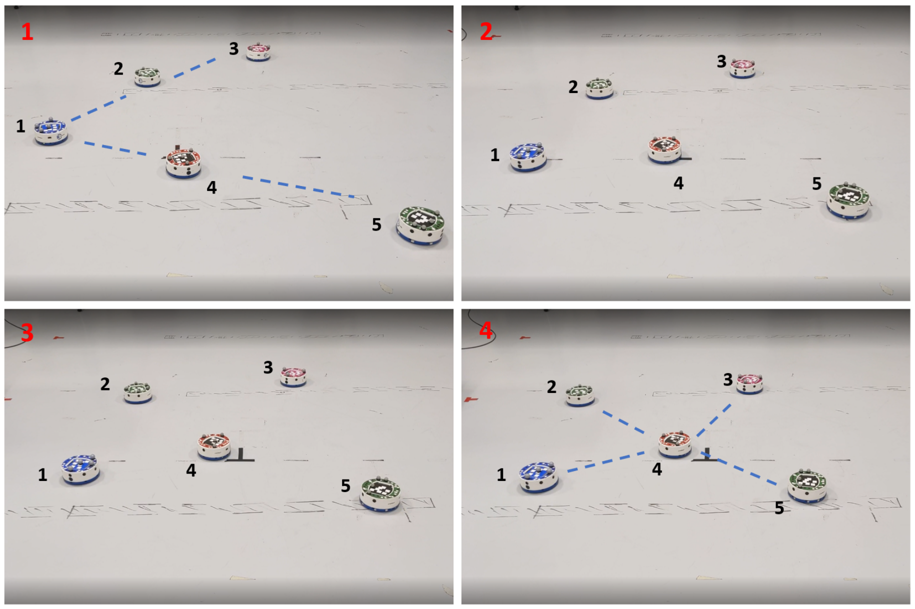


Figure 4.6: Formation assignment with real robots

Notice that each robot moves to the point in the final formation which is closest to its position in the starting formation. This is because formation assignment is optimized for least total distance travelled by the swarm. Although this behavior is expected for majority of the cases, it is not guaranteed as the actual assignment depends on the initial and final shapes. Regardless, the total distance travelled by the swarm will always be minimized.

4.1.3 Motion Primitives

Each motion primitive is designed to change the formation in a defined manner. The `translate` and `rotate` primitives respect the rigidity of a formation, whereas `scale` and `shear` are designed to change the formation shape. For each motion primitive demonstration, a set of robots is arranged in a starting formation in the arena. A new formation is obtained by applying one of the motion primitives to this starting formation. A detailed experimental procedure is described below:

1. **Obtain initial formation of robots:** This step is similar to steps 1-4 from the experimental procedure for Formation Assignment. We obtain the initial formation of robots as follows: create a map of the environment, initialize the robots, obtain a cloud representation of the robots from simulated or real-robot environment, and finally create a formation from the cloud representation.

```
% build map with walls only at corners
map = buildMap(false);

% initialize robots through MATLAB-Buzz interface
initializeRobots(robotIDs);

% create formation from current distribution of robots
current_cloud = getCurrentPositions(robotIDs);
current_formation = Formation(current_cloud, Orientation(0,
    0, 0), 0.14, robotIDs);
```

2. **Apply a motion primitive to create a new formation:** Once the initial formation is obtained, a user can simply apply any of the four motion primitives as follows:

```
new_formation = current_formation.translate(
    translation_vector, map) % meters
new_formation = current_formation.rotate(rotation_angle,
    map) % radians
new_formation = current_formation.scale(scale_factor, map)
new_formation = current_formation.shear(shear_factor, map)
```

Each motion primitive returns a new formation. In addition to an input for each primitive, a map is required to ensure that all robots lie in free space after the motion primitive is executed.

3. **Execute motion:** Once a new formation is obtained, a `Path` is created using the starting and final formations. The `follow` function uses this path to execute the robot motion.

```
path = Path([current_formation new_formation]);
follow(path);
```

The following subsections show results for each motion primitive:

Translate

For translate primitive, robots are arranged in a square formation in the free space and translated 1 meter to the right in simulation (translation vector: [1 0]) and 0.5 m to the right for real-robot experiments (translation vector: [0.5 0]). The first frame in Figure 4.7 shows the direction of the vector being applied at the centroid of the formation.

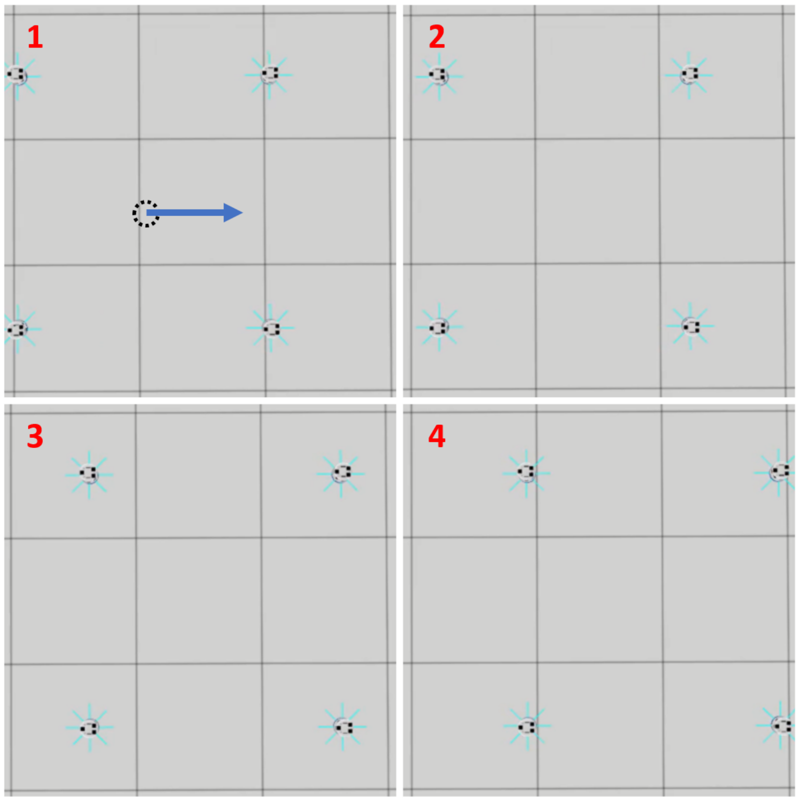


Figure 4.7: Simulated demonstration for Translate primitive

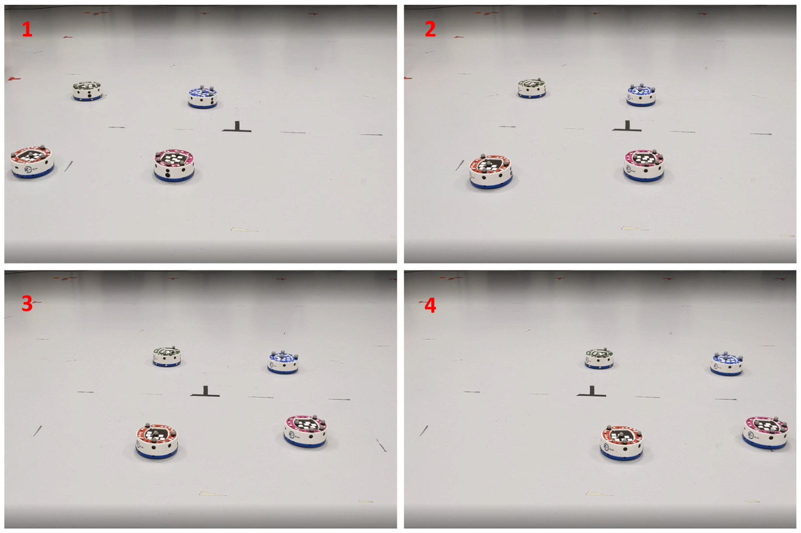


Figure 4.8: Real-robot demonstration for Translate primitive

Rotation

For the rotate primitive, a square formation of robots is rotated counterclockwise by an angle of $\frac{\pi}{4}$. This square formation is centered in the middle of the arena at the start of the experiment. The first frame in Figure 4.9 shows the direction of rotation. This rotation is applied at the center of the formation. A negative angle of rotation would reverse the direction of rotation.

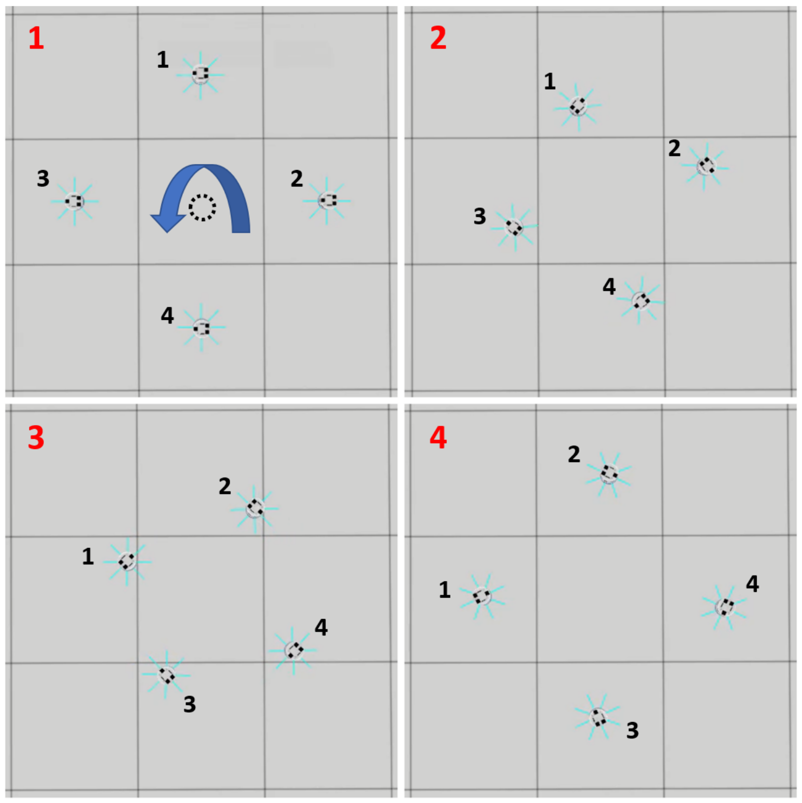


Figure 4.9: Simulated demonstration for Rotate primitive

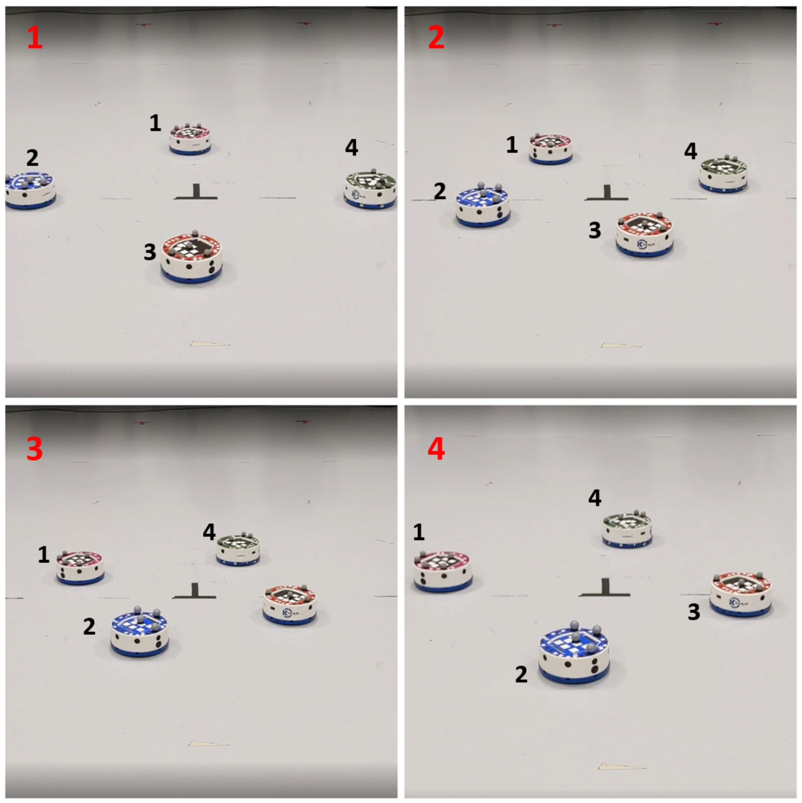


Figure 4.10: Real-robot demonstration for Rotate primitive

Scale

A formation of 7 robots was used for the scale primitive, with one robot placed at the center of the arena. As shown in Figures 4.11 and 4.12, the robot at the center remains in place as the formation is scaled about its centroid. The formation shown here is scaled by a factor of 1.2. A negative scale factor would reduce the size of the formation.

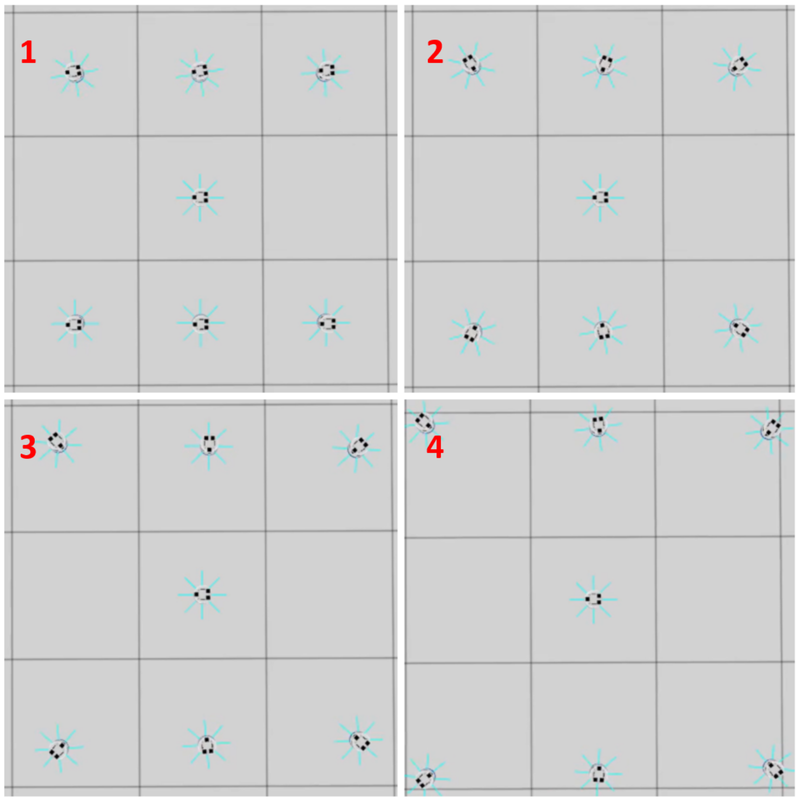


Figure 4.11: Simulated demonstration for Scale primitive

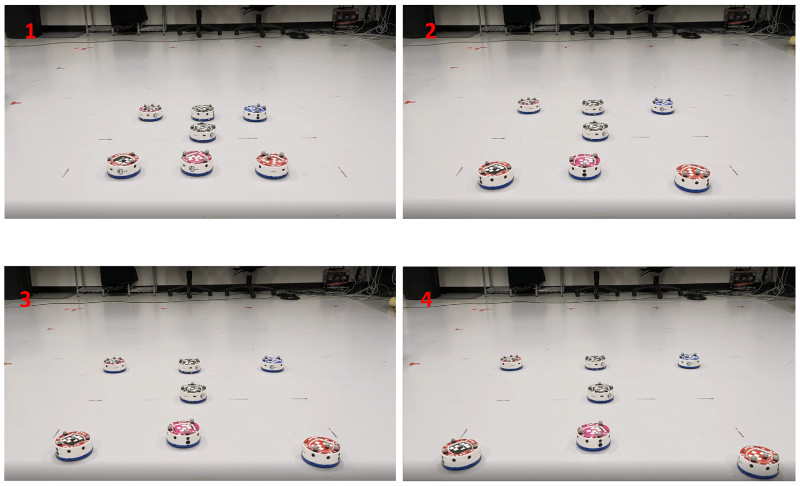


Figure 4.12: Real-robot demonstration for Scale primitive

Shear

Similar to the `scale` primitive, a formation of 7 robots was placed at the center of the arena. Since the `shear` primitive is applied around formation centroid, the robot at the center does not move while the top row moves to the right and the bottom row moves to the left. A negative `shear` factor would reverse the direction of movements of the robots (top row in this formation would move to the left while the bottom row would move to the right).

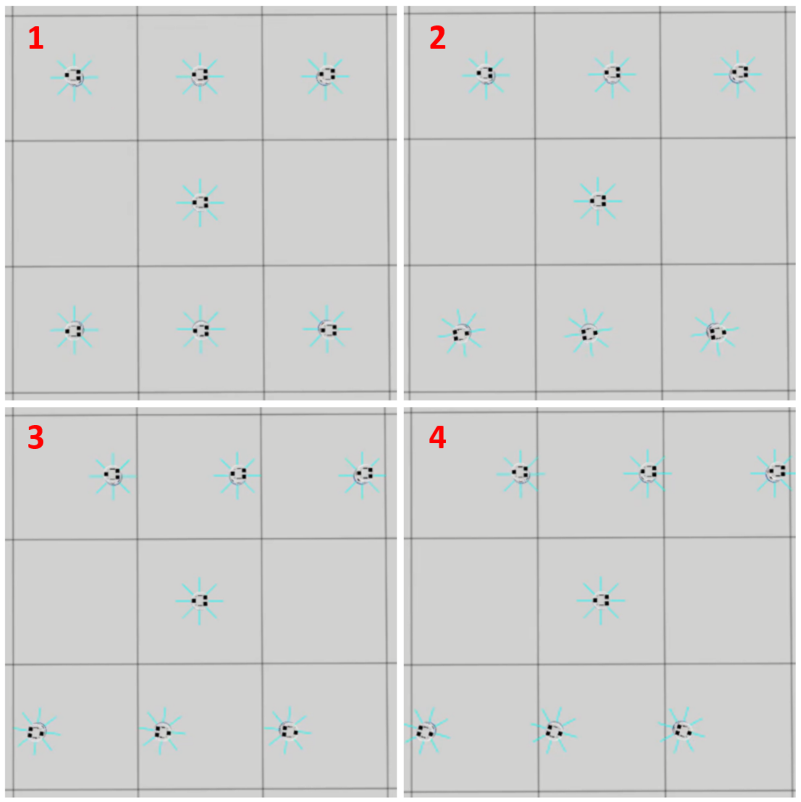


Figure 4.13: Simulated demonstration for Shear primitive

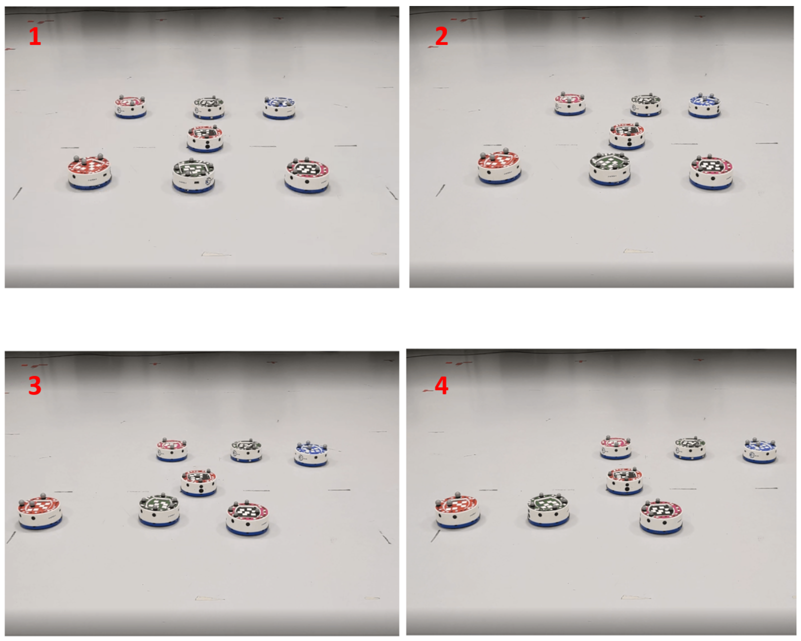


Figure 4.14: Real-robot demonstration for Shear primitive

4.1.4 Navigation

Collision Avoidance

A potential field navigation approach allows robots to reach their goals by following the line-of-sight path between their start and goal positions. Real-time collision avoidance becomes crucial for robots to successfully create formations if no path-deconfliction algorithm is used to resolve expected collisions.

In this section, we demonstrate collision avoidance using a square formation of 4 robots. Each robot starts at one corner of the square and moves to it's diametrically opposite corner in the square. These diametrically opposite pairings of start-goal positions force the robots to steer in each other's vicinity at the center of the square. As shown in frames 2 and 3 of Figure 4.16, the robots avoid collision by maintaining a 'safe distance' (set to 10 cm in this experiment). The repulsive force dominates the attractive force in this scenario, and allows the robots to stop and steer away from one-another.

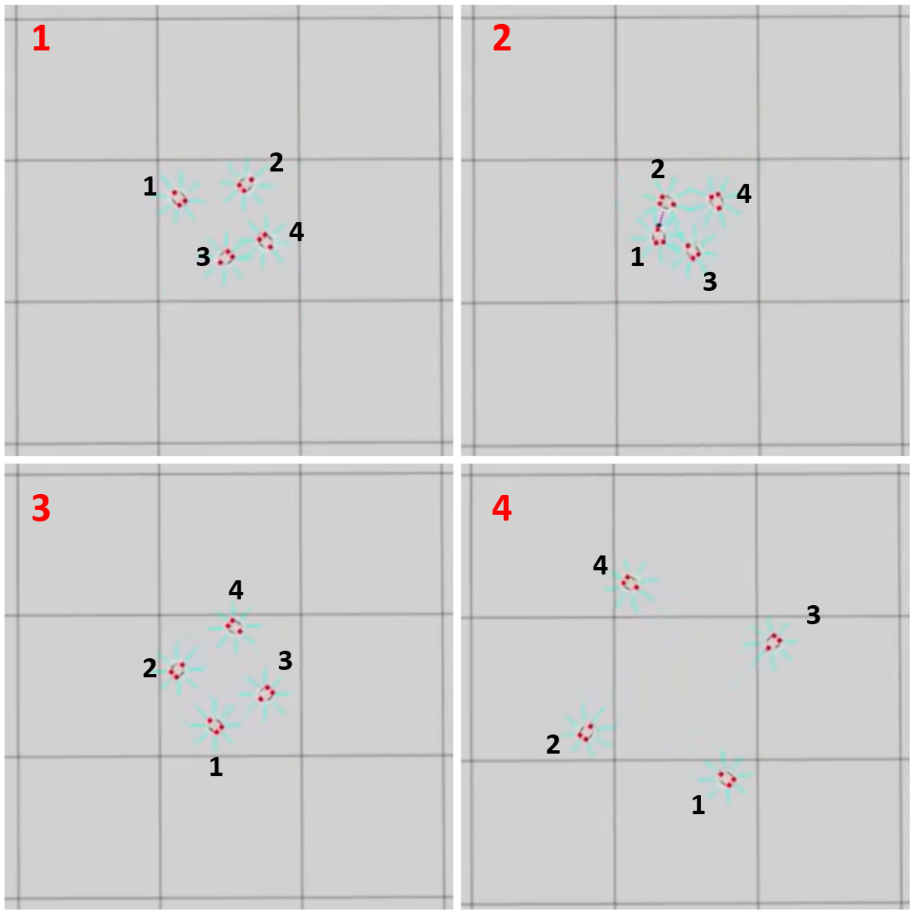


Figure 4.15: Collision avoidance demonstration in simulation

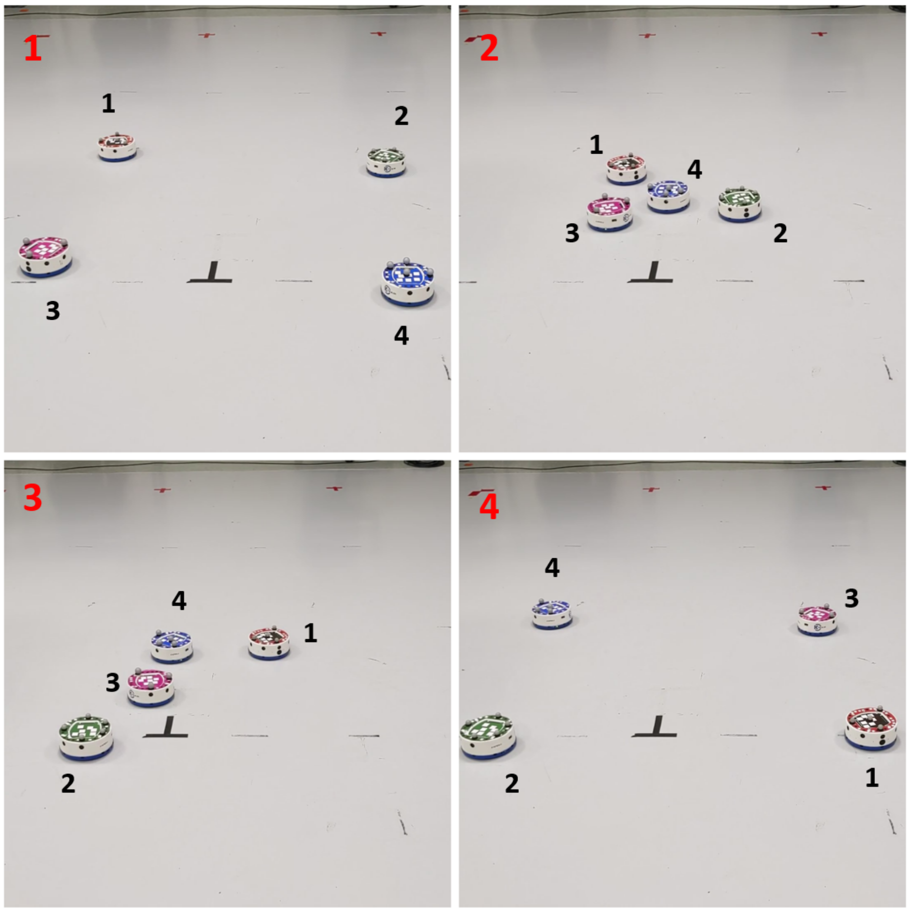


Figure 4.16: Collision avoidance demonstration on real robots

However, the potential-field based approach does not always guarantee collision avoidance. Additionally, as the density of the robots increases, they are prone to falling into a deadlock state. A deadlock state occurs when robots are perpetually stuck in one position. This could happen due to a local minima being created by the potential field approach because the force of attraction equals force of repulsion: this leads to a 0 net force acting on the robot, thus preventing it from moving. We discuss a solution to mitigate this issue in Chapter 5.

Collective Transport

Formation-based collective transport has applications in warehouse settings [32] where AMRs (autonomous mobile robots) can be used to transport large/oversized objects safely. It is also a topic of interest in swarm robotics research, especially in the context of formation control and navigation [14] [27]. While robust navigation and control techniques are required to guarantee success, it is also vital to improve user-interaction for effectively programming robot formations. In their work on human-swarm interaction [23] [22], Patel *et al.* designed an augmented reality based interface to use a swarm of robots to perform collective transport. Patel *et al.* use an environment-oriented modality interface to automatically assign a robot swarm to the object to be transported, and a robot-oriented modality interface to assign individual robots to mitigate any failures.

We take inspiration from this work to show how our interface can be used to perform collective transport. We adopt a robot-oriented modality assignment to allow users to create robot-formations for transporting objects. In the following experiment, we transport a box using a formation of four robots while while avoiding a static object (a middle wall). The following script describes this experiment:

```
% declare robotIDs used in this experiment
robotIDs = [1 2 3 4];

% load map of environment
map = buildMap(true);

% initialize robots
initializeRobots(robot_IDs);

% get current positions of agents as a cloud
```

```
initial_distribution_cloud = getCurrentPositions(robot_IDs);
% get start cloud graphically
gpi = graphicalPositionInput(map, length(robot_IDs), ...
    initial_distribution_cloud);
start_cloud = gpi.start();

% assign robots to start formation
current_formation = Formation(initial_distribution_cloud, ...
    Orientation(0, 0, 0), 0.0, robot_IDs);
start_formation = assign(map, current_formation, start_cloud);

% construct and follow path to move robots to target formation
path = Path([current_formation start_formation]);
follow(path);

% select goal graphically
goal_pose = gpi.goal(start_formation);

% calculate path to goal
path = planPath(start_formation, goal_pose, map, @prmPlanner);

% follow generated path
follow(path);
```

In this script, the user first creates a formation from the robot distribution at the start of the experiment. The user then selects a new formation to be placed around the object (box) to be transported and moves the robots to that formation. The user then chooses a ‘goal’ position for the object (where the object needs to be transported). The `planPath` function returns a valid path for the formation to transport the box to this goal. Finally, this path is executed using the `follow` function.

Frame 1 in Figure 4.17 shows a user selecting a formation to be placed around the object to be transported. Frame 2 shows an inflated map generated based on this formation's diameter. This map is used to select a 'safe' goal for the formation. A safe goal is any point in the white portion of the inflated map, and determines where the formation's centroid needs to go. Frame 3 shows the planner returning a safe path for the formation (in red), and frame 4 shows the formation of robots following that path in simulation.

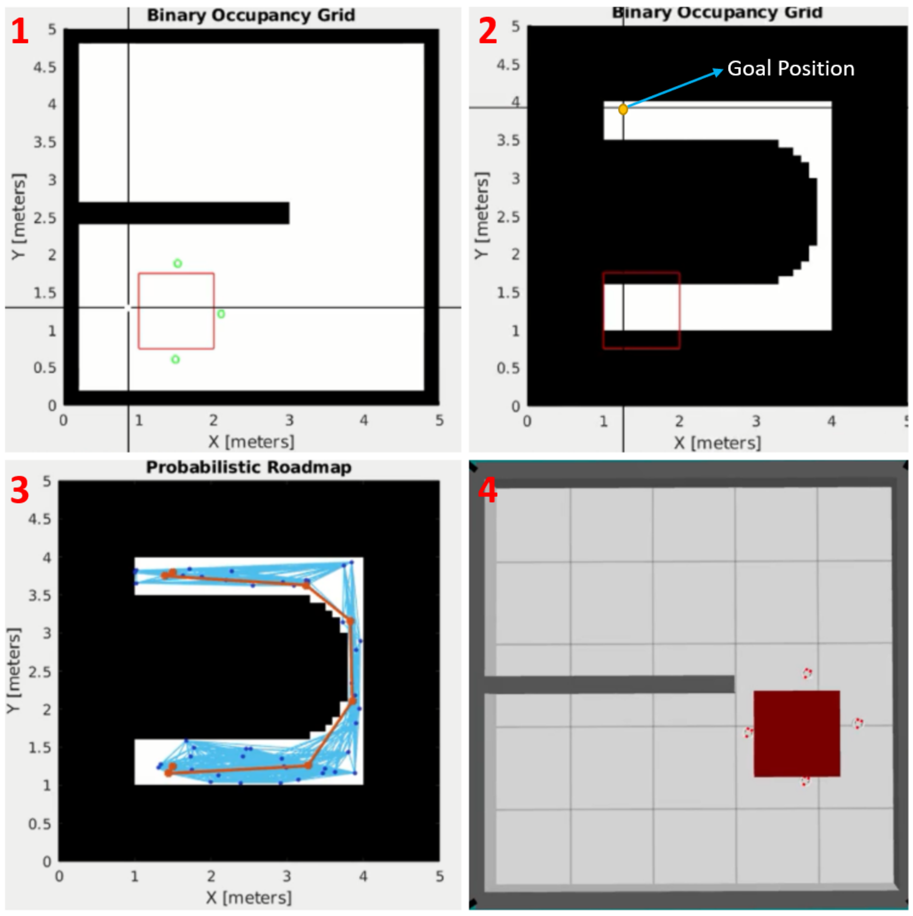


Figure 4.17: Planning process for collective transport application

Figures 4.18 and 4.19 show the collective transport experiment in simulation and on real robots respectively.

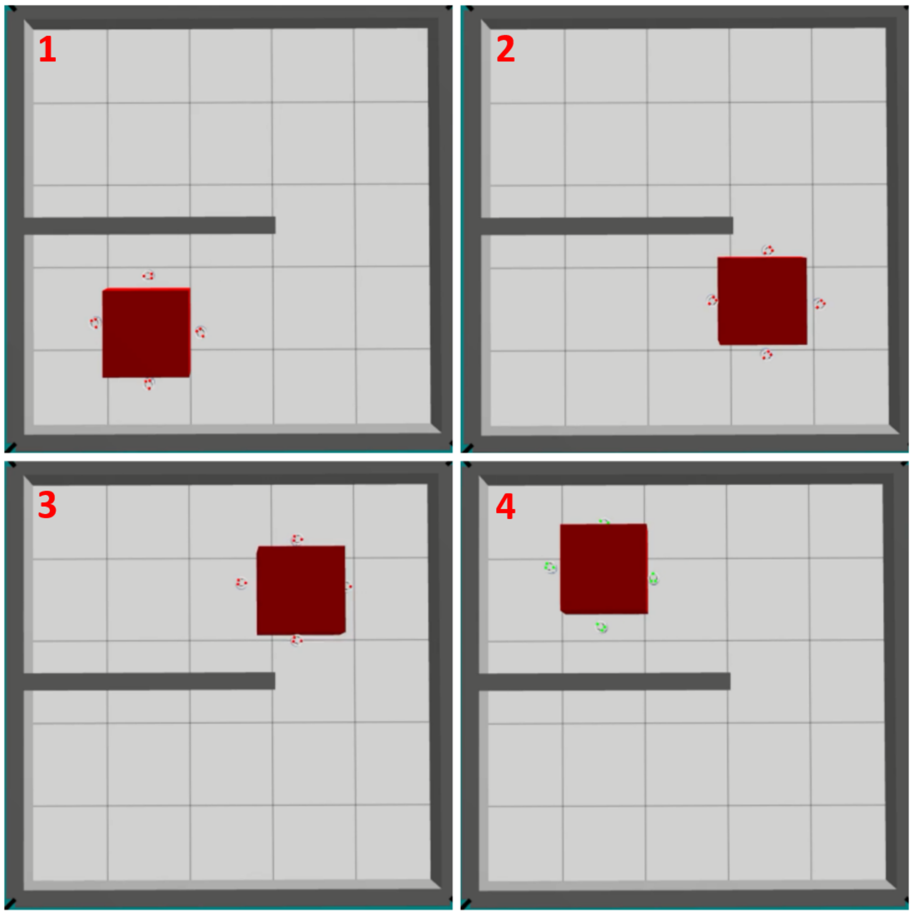


Figure 4.18: Simulated demonstration of collective transport

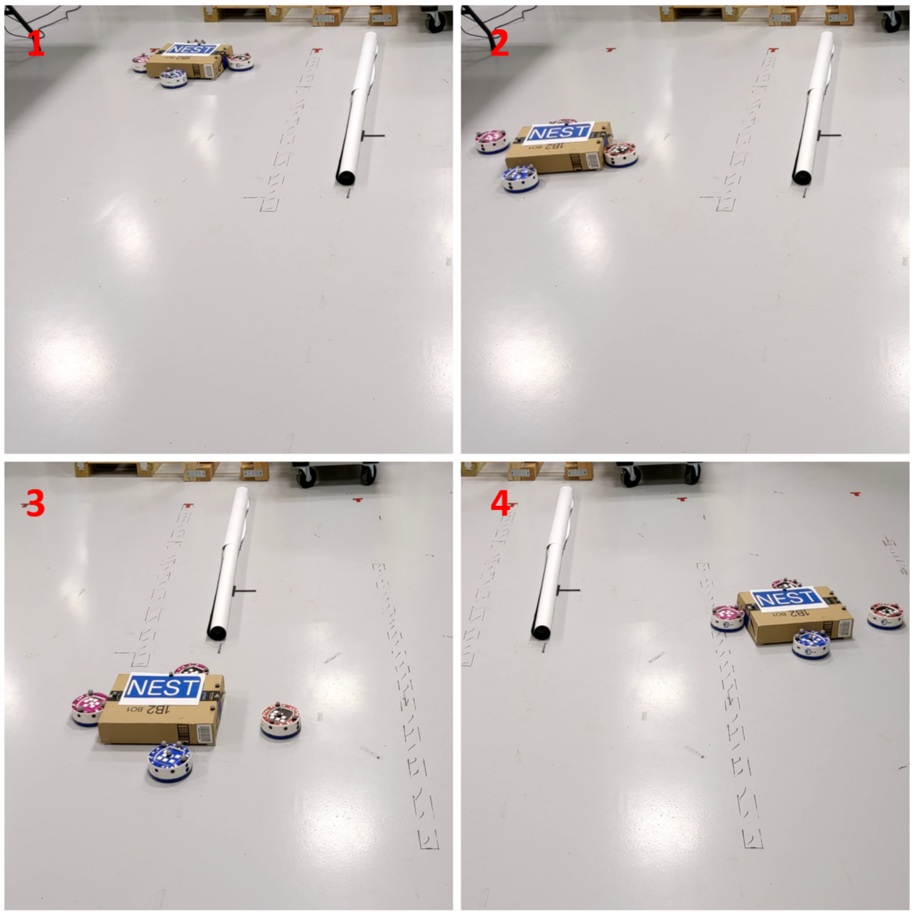


Figure 4.19: Real-robot demonstration of collective transport

4.2 Goal Assignment Performance

4.2.1 Setup

It is implicit that the goal assignment process will take longer as the swarm size increases. Therefore, we measure the performance of our problem formulation to judge its utility for applications of varying swarm sizes. In order to characterize our solution's performance, we observed the time taken by MATLAB's `intlinprog` function to find a feasible solution for randomly spawned agents in a 500x500 size grid. We start with a swarm size of 1 agent and increment the swarm size by 1 at each iteration till the size grows to 500 agents. We measured the time-elapsed for `intlinprog` function call to calculate goal assignment at each iteration. The experiment was repeated 25 times. This experiment was conducted on an Ubuntu 20.04 machine with i5-6300U dual core CPU and 20 GB 2133 MHz RAM. The code snippet below shows this process. The `milp` function in this code calls `intlinprog` and returns the time taken by `intlinprog` to find a solution.

```
start_size = 1;
end_size = 500;
map_order = 500; % max x and y points each in the map
runs = 25;
current_time_vector = zeros(end_size, 1);
data_matrix = [start_size:end_size]';
for j = 1:runs
    for i = start_size:end_size
        % randomly generate start and end coordinates for
           agents in this map
        start_coordinates =  [randi(map_order, i, 2)];
        end_coordinates = [randi(map_order, i, 2)];
        current_time_vector(i) = milp(start_coordinates,
           end_coordinates);
    end
    data_matrix = cat(2, data_matrix, current_time_vector);
end
```

4.2.2 Result

Figure 4.20 shows a plot of the time costs of goal assignment associated with all swarm sizes averaged over 25 runs. It can be seen that the goal assignment process starts becoming noticeable beyond a swarm size of 100 agents.

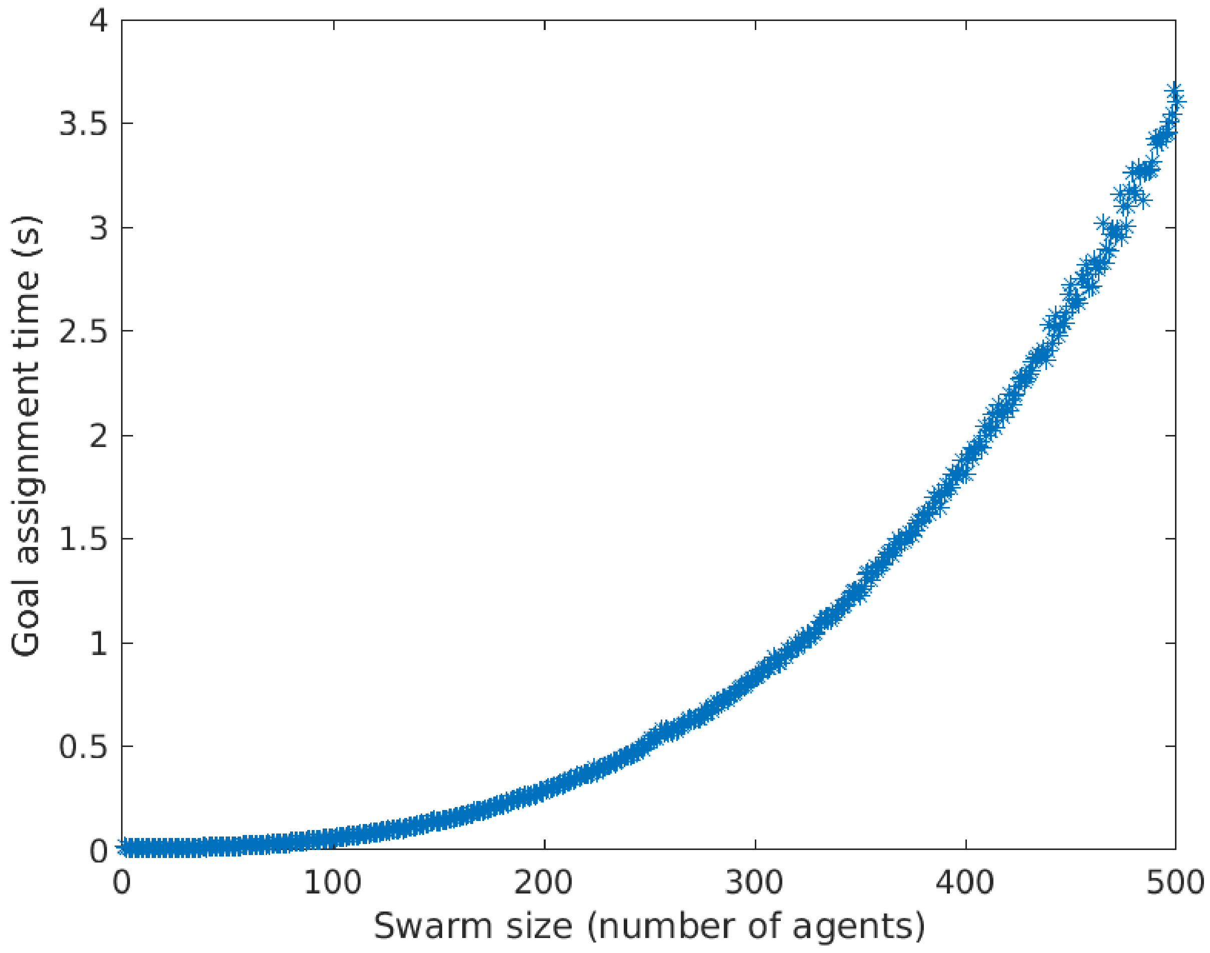


Figure 4.20: Performance of MILP-based goal assignment as the number of agents in the experiment increases.

It should be noted that the time taken by the goal assignment process might not be relevant if the assignment is done in the planning phase of an experiment. For example, in the Formation Assignment experiment 4.1.2, the formation assignment calculation is completed before the robots start to move. However, the execution time for formation assignment might become a bottleneck if an experiment involves sequential formation assignment calculations and motion operations (as shown in the collective transport experiment 4.1.4). This would especially impact applications where the swarm size is large. One example is drone shows where the formation is required to frequently change shapes in a smooth motion. Drone shows often involve large formations, some having occasionally crossed 1000 drones [19].

Chapter 5

Conclusion

Efforts in software development and research for swarm robotics have focused on enabling emergent behavior through decentralized communication among robots. Some tasks that are dependent on robot swarms require the swarm to achieve a specific shape, called a formation. Examples of such tasks include drone navigation in varying wind conditions [10], collective transport [27], and drone light shows [19]. Additionally, drone light shows and drone navigation require the swarm to frequently switch between a set of predetermined formations. While software packages and research efforts have introduced methods for programmatically creating robot formations, these methods do not facilitate reusability or ease of use from the perspective of programming formations [17] [31]. This creates a need for a software interface that enables users to program robot swarms in a formation-centric manner.

In this thesis, we addressed this need by introducing a set of types and functions that constitute a formation-centric software interface. The types were developed to let a user define a robot formation in a minimalistic manner, whereas functions were created to describe how a user could interact with a formation to enable elementary motion. We introduced formation assignment and manipulation as two main features of this software interface and provided our formulation for each feature. For formation assignment, we introduced a MILP (Mixed-Integer Linear Programming) based algorithm which minimizes the total distance travelled by the swarm while changing formations. For formation manipulation, we introduced a set of four motion primitives as well as a path planning function which returns a viable path for a rigid formation using a user-provided planner. Additionally, we described a potential-field based navigation technique used to execute the motion generated through the functions in this interface.

To show the effectiveness of this interface, we demonstrated similarity in code execution using simulated and real-robot experiments for formation assignment, motion primitives, collision avoidance and collective transport. Lastly we shared the performance of our MILP-based formation assignment algorithm and discussed its limitations.

5.1 Future work

There are multiple avenues of improvement for this work which can be categorized as follows:

1. **Guaranteed decentralized collision avoidance:** The current potential-field based navigation approach in this work relies on a motion capture system to enable collision avoidance between robots. This prevents robots from avoiding objects that are not tracked. A decentralized collision avoidance technique can be developed by calculating the repulsive force in Equation 3.18 using sensor data from ultrasound or proximity sensors on robots. However, this would still not guarantee collision avoidance. This problem has been addressed by Wang *et al.* who have demonstrated collision-free navigation for multirobot

systems [34]. Wang *et al.* use barrier control certificates to satisfy safety constraints using a potential field navigation approach in a decentralized manner. Their work also addresses deadlock avoidance which is an important feature in guaranteeing that the robots reach their goal. We can modify our navigation approach by incorporating their solution.

2. **Implementation on other software platforms:** The MATLAB-Buzz software architecture offers extended functionality through MATLAB toolboxes and Buzz features. However, for wider adoption, we would like to replicate the implementation of this interface on different platforms. Introducing this interface as a ROS package will make it accessible to the general robotics community.

3. **Optimal navigation:** In this work, each robot follows a line-of-sight path to its assigned navigation goal while avoiding obstacles using an artificial potential field approach. This approach is computationally inexpensive due to the use of a euclidean distance based heuristic to calculate the path cost. However, the reactive nature of this navigation strategy might lead to less optimal results as the robots take longer to avoid static obstacles. Graph-based navigation techniques such as A* and RRT calculate more informed paths by accounting for static obstacles, but they require a higher cost of computation. Therefore, it might be worthwhile to study the trade-off between artificial potential fields and graph-based navigation strategies to determine which approach leads to more optimal navigation. Although such comparison has been conducted for single robot systems and robot swarms [24], it is important to study the difference in performance of these navigation strategies especially in the context of formation assignment, where the computational cost of a planner becomes

significant for larger swarm sizes.

5.2 Lessons learned

While working with real robot hardware, I learned that it is helpful to look at problems from a different perspective to troubleshoot an issue. When I was replicating simulated experiments with real-robots, I assumed the max force values (see Equations 3.18 and 3.19) to be the same for simulated and real robots. I observed that using the same max force values for real robots led to an 'oscillation' behavior: the robots would turn in-place, alternating direction for every turn. After trying different configuration settings on the real robots, I decided to lower the max force values in my programming interface. By doing so I realized that a higher max-force value was causing the robots to overshoot their turn while trying to align with the given heading. This was preventing them from navigating to their goal.

I also learned that it is faster to get a different perspective from another team member. I invested quite some time to learn how to use the Vicon motion capture system. While trying to configure all nodes in the communication loop for the real-robot setup (see Figure 4.1), I branched off from the default configuration settings for the Vicon server. This prevented ARGoS from communicating with the Vicon system. My team member retried configuring Vicon with default settings for the server, which solved the issue. This issue would have been resolved sooner if I reached out for help at an earlier stage.

This project was my first attempt at conducting research, and I picked up a few lessons along the way. The most important lesson I learned was how to present research work in a proper format. I realized that establishing context for a work using current literature helps the reader understand the true contribution of the work. I also realized that using the appropriate level of abstraction helps the reader to keep track of most important ideas presented in the work.

Bibliography

[1] Argos extensions. https://www.argos-sim.info/extensions.php.

[2] Argos khepera extension. https://www.argos-sim.info/extensions.php.

[3] Argos khepera extension: Setwheelspeedsfromvector. https://github.com/ilpincy/argos3-kheperaiv/blob/7807c3b71e13256c915dcc909707fee50a179273/src/plugins/robots/kheperaiv/control_interface/buzz_controller_kheperaiv.cpp#L362.

[4] C++ mex functions. https://www.mathworks.com/help/matlab/matlab_external/c-mex-functions.html.

[5] Matlab - double-precision arrays. https://www.mathworks.com/help/matlab/ref/double.html.

[6] Matlab binaryoccupancymap. https://www.mathworks.com/help/nav/ref/binaryoccupancymap.html.

[7] Matlab navigation toolbox. https://www.mathworks.com/products/navigation.html.

[8] Matlab prm example. https://www.mathworks.com/help/robotics/ug/probabilistic-roadmaps-prm.html.

[9] Matlab robotics toolbox. https://www.mathworks.com/products/robotics.html.

[10] Alkouz, B., and Bouguettaya, A. Formation-based selection of drone swarm services. In *MobiQuitous 2020-17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services* (2020), pp. 386–394.

[11] Beltrame, G., Merlo, E., Panerati, J., and Pinciroli, C. Engineering safety in swarm robotics. In *Proceedings of the 1st International Workshop on Robotics Software Engineering* (2018), pp. 36–39.

[12] Chang, X., Cai, Z., Wang, Y., and Yi, X. micros swarm framework. https://wiki.ros.org/micros_swarm_framework.

[13] Du, X., Luis, C. E., Vukosavljev, M., and Schoellig, A. P. Fast and in sync: Periodic swarm patterns for quadrotors. In *2019 International Conference on Robotics and Automation (ICRA)* (2019), IEEE, pp. 9143–9149.

[14] Farivarnejad, H., and Berman, S. Multirobot control strategies for collective transport. *Annual Review of Control, Robotics, and Autonomous Systems 5* (2022), 205–219.

[15] Farivarnejad, H., Wilson, S., and Berman, S. Decentralized sliding mode control for autonomous collective transport by multi-robot systems. In *2016 IEEE 55th conference on decision and control (CDC)* (2016), IEEE, pp. 1826–1833.

[16] Habibi, G., Xie, W., Jellins, M., and McLurkin, J. Distributed path planning for collective transport using homogeneous multi-robot systems. In *Distributed Autonomous Robotic Systems* (Tokyo, 2016), N.-Y. Chong and Y.-J. Cho, Eds., Springer Japan, pp. 151–164.

[17] Kaiser, T. K., Begemann, M. J., Plattenteich, T., Schilling, L., Schildbach, G., and Hamann, H. Ros2swarm-a ros 2 package for swarm robot behaviors.

[18] Khaldi, B., and Cherif, F. An overview of swarm robotics: Swarm intelligence applied to multi-robotics. *International Journal of Computer Applications 126*, 2 (2015).

[19] Kung, C.-m., Yang, W.-S., Wei, T.-Y., and Chao, S.-T. The fast flight trajectory verification algorithm for drone dance system. In *2020 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)* (2020), pp. 97–101.

[20] Levoy, M., and Whitted, T. The use of points as a display primitive.

[21] Nilles, A. Q., Beckman, M., Gladish, C., and LaViers, A. Improv: Live coding for robot motion design. In *Proceedings of the 5th International Conference on Movement and Computing* (2018), pp. 1–6.

[22] Patel, J., and Pinciroli, C. Improving human performance using mixed granularity of control in multi-human multi-robot interaction, 2019.

[23] Patel, J., Xu, Y., and Pinciroli, C. Mixed-granularity human-swarm interaction. In *2019 International Conference on Robotics and Automation (ICRA)* (2019), pp. 1059–1065.

[24] Patle, B., Babu L, G., Pandey, A., Parhi, D., and Jagadeesh, A. A review: On path planning strategies for navigation of mobile robot. *Defence Technology 15*, 4 (2019), 582–606.

[25] Pinciroli, C., and Beltrame, G. Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2016), pp. 3794–3800.

[26] Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Stirling, T., Gutiérrez, A., Gambardella, L. M., and Dorigo, M. 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.

[27] Rubenstein, M., Cabrera, A., Werfel, J., Habibi, G., McLurkin, J., and Nagpal, R. Collective transport of complex objects by simple robots: theory and experiments. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems* (2013), pp. 47–54.

[28] Saidonr, M. S., Desa, H., and Rudzuan, M. N. A differential steering control with proportional controller for an autonomous mobile robot. In *2011 IEEE 7th International Colloquium on Signal Processing and its Applications* (2011), pp. 90–94.

[29] Sauter, J. A., Mathews, R. S., Yinger, A., Robinson, J. S., Moody, J., and Riddle, S. Distributed pheromone-based swarming control of unmanned air and ground vehicles for rsta. In *Unmanned Systems Technology X* (2008), vol. 6962, SPIE, pp. 109–120.

[30] Stoy, K. Using situated communication in distributed autonomous mobile robotics. pp. 44–52.

[31] Testa, A., Camisa, A., and Notarstefano, G. Choirbot: A ros 2 toolbox for cooperative robotics. *IEEE Robotics and Automation Letters 6*, 2 (2021), 2714–2720.

[32] Tse, S. K., Ben Wong, Y., Tang, J., Duan, P., Leung, S. W. W., and Shi, L. Relative state formation-based warehouse multi-robot collaborative parcel moving. In *2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)* (2021), pp. 375–380.

[33] Turpin, M., Mohta, K., Michael, N., and Kumar, V. R. Goal assignment and trajectory planning for large teams of aerial robots. In *Robotics: Science and Systems* (2013).

[34] Wang, L., Ames, A. D., and Egerstedt, M. Safety barrier certificates for collisions-free multirobot systems. *IEEE Transactions on Robotics 33*, 3 (2017), 661–674.

Appendix A

Data Type API

Data member / function	**Description**
`Position`	x, y and z components to determine agent position
`Orientation`	roll, pitch and yaw components to determine agent orientation
`Pose`	Position and Orientation components to determine agent pose
`Cloud`	Array of Pose: represents an unordered distribution of agents
`Formation`	Array of Pose: represents an ordered distribution of agents
`Path`	Array of Formation: represents the path a swarm will take by transitioning through these formation
`graphicalPositionInput`	Data type used to graphically create a formation from a cloud, or choose goal pose for the created formation

Table A.1: Overview of data types

Data member / function	Description
`x:double`	x component
`y:double`	y component
`z:double`	z component
`Position(double, double, double):Position`	Constructor: creates `Position` from given x, y, and z values
`plus(Position, Position):Position`	operator overloading for addition
`minus(Position, Position):Position`	operator overloading for subtraction
`mtimes(Position, double):Position`	operator overloading for multiplication
`xyz():3x1 double`	method for returning x, y and z components as 3x1 column vector

Table A.2: Overview of Position class

Data member / function	Description
`roll:double`	roll rotation component
`pitch:double`	pitch rotation component
`yaw:double`	yaw rotation component
`Orientation:Orientation(double, double, double)`	Constructor: creates Orientation from given roll, pitch and yaw values
`plus(Orientation, Orientation):Orientation`	operator overloading for addition
`minus(Orientation, Orientation):Orientation`	operator overloading for subtraction
`mtimes(Orientation, double):Orientation`	operator overloading for multiplication
`rpy():3x1 double`	method for returning roll, pitch and yaw components as 3x1 column vector

Table A.3: Overview of Orientation class

Data member / function	Description
`position:Position`	Position (x, y, z) information about the agent
`orientation:Orientation`	Orientation (roll, pitch, yaw) information about the agent
`Pose(Position, Orientation)`	Constructor: creates Pose from given Position and Orientation
`plus(Pose, Pose)`	operator overloading for addition
`minus(Pose, Pose)`	operator overloading fro subtraction

Table A.4: Overview of Pose class

Data member / function	**Description**
`poses:Pose`	Pose array holding Pose for each agent
`Cloud(Pose[])`	Constructor for creating Cloud from given Poses
`cloud2mat():3xn double`	method for getting matrix of positions of agents in the cloud. Rows 1, 2 and 3 correspond to x, y z for each agent. There are n columns, where n=swarm size.

Table A.5: Overview of Cloud class

Data member / function	**Description**
`agent_ids:double[]`	array of agents ids for agents in this formation
`diameter:double`	diameter of this formation
`heading:Orientation`	heading of the formation. Always points in the local x-direction by default
`centroid:double`	center of mass of the formation
`cloud:Cloud`	underlying cloud for this formation
`agent_diameter:double`	diameter of each agent of the formation (homogeneous formation)
`Formation(Cloud, Orientation, double, double[]):Formation`	Constructor for creating a Formation from the given cloud. The second argument sets the orientation of the formation, the third argument is used to set agent diameter, and the fourth argument is used to set the agent ids.
`translate(Position, binaryOccupancyMap):Formation`	translation primitive: requires valid translation vector of type Position.
`rotate(double, binaryOccupancyMap):Formation`	rotation primitive: requires valid rotation angle of type double.
`scale(double, binaryOccupancyMap):Formation`	scaling primitive: requires valid scale factor of type double.
`shear(double, binaryOccupancyMap):Formation`	rotation primitive: requires valid shear factor of type double.

Table A.6: Overview of Formation class

Data member / function	**Description**
`formations:Formation`	array of `Formation`
`Path(Formation[])`	Constructor for creating Path object from given array of `Formations`

Table A.7: Overview of Path class

Data member / function	**Description**
`map:binaryOccupancyMap`	map of the environment
`number_of_points:int`	Number of points to select graphically for the start cloud
`current_cloud`	cloud of agents at the start
`graphicalPositionInput( binaryOccupancyMap, int, Cloud)`	constructor for setting map, number_of_points and current_cloud
`start():Cloud`	method for graphically selecting a cloud
`goal(Formation):Pose`	method for graphically selecting a goal pose for a formation

Table A.8: Description of graphicalPoseInput class

Appendix B

Follow function: Source Code

The code snippet below shows the source code for the `follow` function.

```
function result = follow(path)
    result = 1;
    global rid positions;
    robotIDs = cell2mat(path.formations(1).agent_ids);

    % move agents to next formation
    for i = 1:length(path.formations)
        formation = path.formations(i);
        cloud_matrix = formation.cloud.cloud2mat();
        positions = (cloud_matrix(1:2, :))';
        robotIDs = cell2mat(formation.agent_ids);
        moveRobots(robotIDs);
    end

    % check all agents are within allowable distance of their
       resepective
    % goal positions
    for i = length(robotIDs)
        rid = robotIDs(i);
        result = result & reachedGoal(rid);
    end
end
```

The first `for` loop is intended for navigation. The next target position for each robot in the current formation is stored in the global variable `positions`. On each iteration of the first `for` loop, the position and robot id information is extracted for the next `Formation` in `Path`. A `Formation` object ensures these positions and agent id order match. The `moveRobots` function is responsible for moving each robot to its position in the next formation.

The second `for` loop is intended for ensuring that the robots reached their goal. The `reachedGoal` function implements this logic by simply checking that each robot is within a certain allowable proximity of its goal.

`moveRobots` and `reachedGoal` are implemented as follows:

```
function moveRobots(robotIDs)
    global robots ridMap rid;
    % logical array keeping track of whether robots have reached
       their goal
    robotStatus = ones(1, length(robotIDs));

    % Set all robot LEDs to red before start of navigation
    for i = 1:length(robotIDs)
            rid=robotIDs(i);
            set_leds(255, 0, 0);
    end

    % Use goal proximity to determine end-of-experiment
    while(sum(robotStatus) > 0)
        ping();
        for i=1:length(robotIDs)
            rid = robotIDs(i);
            % robot didn't reach goal when last checked
            if robotStatus(i) == 1
                % check if robot has reached goal
                if reachedGoal(rid)
                    robotStatus(i) = 0;
                     % set robot LED to green
                    set_leds(0, 0, 255);
                    stop();
                else
                    step(); % keep moving robot if it hasn't
                       reached goal
                end
            end
        end
    end
end
```

```
function status = reachedGoal(rid)
    global robots positions ridMap;

    current_pose = pose();
    goal = positions(ridMap(rid), :);

    if pdist([current_pose(1, 1:2); goal], 'euclidean') < 0.1 %
       arbitrary
        status = 1; % true
    else
        status = 0; % false
    end
end
```

The `step` function in `moveRobots` is responsible for calculating the net force acting on a robot, and consequently calling the `goto` function for sending the control commands to the robots based on this net force. This process is described in detail in the **Navigation and control strategies** section.

Appendix C

Net Force Calculation: Source Code

```
function total_force = centrallizedForceVector(max_force)
    % declare global variables
    % positions: holds list of goals for each robot
    % ridMap: an ordered map containing which robot holds what id (
       robots are ordered from 1-n)
    % rid: indicates the rid of the robot being navigated currently
    global positions ridMap robots rid;

    %% Repulsive force

    % account for robot diameter while calculating distances wrt
       other
    % robots
    r_robot = 7; % radius of kheperaIV robots (cm)

    % account for 'safe distance' between two robots to avoid
       collision
    dist_safe = 10; % tunable parameter (cm)

    % Function handle for repuslive force: force should decrease
       with increase in distance
    F_r = @(dist_r):min(max_force, max_force*(1/cappedDistance(
       dist_r, r_robot, dist_safe)^2));

    % get information about neighbors of this robot
    bvm_state = robots(ridMap(rid)).bvm_state;

  % Calculate repulsive forces based on neighbor distance
    if ~ isempty(bvm_state.neighbors)
        % get angles of all neighbors with respect to this robot
        thetas = vertcat(bvm_state.neighbors.azimuth);
        % get line-of-sight distances of this robot from its
           neighbors
        neighborDistances = vertcat(bvm_state.neighbors.distance);
        % calculate position vectors from this robot to neighbors
           in polar form
        rhos = arrayfun(F_r, neighborDistances);

        % convert position vectors to neighbors to Cartesian form
        [r_x, r_y] = pol2cart(thetas, rhos);
        d_x = sum(r_x);
        d_y = sum(r_y);
    end

    %% Attractive force

    goal = positions(ridMap(rid), :);

    % convert goal to robot frame
    goal = world2RobotFrame(goal(1), goal(2)).';

    % find goal angle with respect to robot x frame
    goalAngle = findAngle([1, 0], goal);

    % obtain line-of-sight distance between robot and goal
    goalDist = pdist([0 0; goal], 'euclidean');

    % calculate force of attraction based on line-of-sight distance
        between
    % robot and goal
    F_a = min(max_force, max_force*dist_a);

    % convert force vector to Cartesian form
    [g_x, g_y] = pol2cart(goalAngle, F_a);

    % Vector addition of attractive and repulsive forces
    f_x = d_x + g_x;
    f_y = d_y + g_y;

    total_force = [f_x f_y];
end

function final_dist = cappedDistance(dist_r, r_robot, dist_safe)
    final_dist = dist_r - 2*r_robot - dist_safe;
    if final_dist < min_dist
        final_dist = min_dist;
    end
    final_dist = single(final_dist);
end
```