Autonomous Re-Configuration of Multi-Agent Formation in the Presence of a New Agent

Bachelor Thesis

Faculty of Science and Engineering

Industrial Engineering and Management

June 2018

Author:
Mariano Perez Chaher
S2872196

Supervisors:
prof. dr. Bayu Jayawardhana
dr. Mauricio Muñoz Arias

1 Abstract

Formation control has developed over the last years to accommodate robotic agents to complete increasingly complex tasks. A studied approach for flexible robot formations is that of distributed control, consisting of decentralized data processing of agents, allowing all agents to remain in formation without a global frame of reference and no communication between agents. This thesis sets to create an algorithm that allows an existing formation to add and extract agents during operation to make the system distributedly dynamic. A revolving LIDAR scanner is taken as the only sensing tool for the agents, providing an all-around point cloud of their environment. Finally, after covering the working of the algorithm, tests are performed through simulations of seven robots and real-life applications on four robots.

Table of Contents

1 ABSTRACT 1

2 INTRODUCTION 5

2.1 THESIS STRUCTURE 5

3 PROBLEM ANALYSIS 6

3.1 AUTONOMOUS FORMATION CONTROL 6
3.2 DISTRIBUTED CONTROL 7
3.3 STAKEHOLDER ANALYSIS 7
3.3.1 Problem Owner Analysis 8
3.3.2 Research Staff at the DTPA Lab 8
3.3.3 DTPA Lab researchers 8
3.3.4 Mauricio Muñoz Arias 8
3.3.5 Region of Smart Factories 8
3.4 SYSTEM ANALYSIS 9
3.5 RESEARCH QUESTION 10
3.5.1 Sub-Questions 10

4 LITERATURE REVIEW 10

4.1 NOTATION 10
4.2 GRAPH ANALYSIS OF FORMATIONS (DE MARINA PEINADO, CAO, & JAYAWARDHANA, 2016) 11
4.3 FORMATION RIGIDITY 11
4.3.1 Henneberg Insertion 12
4.4 HECTOR DE MARINA'S CONTROLLER 12
4.5 CONTROLLER WITH ESTIMATOR 13
4.6 ILLUSTRATIVE EXAMPLE 14

5 EXPANSION OF EXISTING FORMATION 15

5.1 DETECTION OF AGENTS 15
5.2 FORMATION CONTROL WITH ADDITION OF AGENT 16
5.2.1 Data Processing Node for N 16
5.2.2 Controller for N 19
5.2.3 Controller with Estimator 20

6 SIMULATION AND RESULTS 22

6.1.1 Software 23
6.2 COMPUTATION PROCESS 23
6.3 EXPERIMENTS 24
6.3.1 Basic Formation Control 24
6.3.2 Extended formation Control 26
6.3.3 Addition and Extraction of Agents 29
6.3.4 Estimator 33

7 EXPERIMENTAL SETUP 35

7.1 DTPA LAB EQUIPMENT 35
7.1.1 Nexus Robots 35
7.1.2 Available Sensors 35
7.2 RESULTS 36
7.2.1 Basic Formation of Four Agents 37
7.2.2 Addition and Extraction of an Agent 38

8 CONTRIBUTIONS 40

8.1 PHYSICS BASED SIMULATION 40
8.2 DYNAMIC DISTIBUTED FORMATION CONTROL 40
8.3 INSIGHT INTO THE APPLICATION OF ESTIMATORS 40

9 DISCUSSION 40

9.1 FUTURE RESEARCH 42

10 CONCLUSION 43

11 REFERENCES 44

12 APPENDIX 45

12.1 DATA PROCESSING NODE FOR N AGENTS 45
12.2 CONTROLLER NODE FOR N AGENTS 47
12.3 CONTROLLER NODE WITH ESTIMATOR FOR N AGENTS 51

List of Variables

B – Incidence matrix of formation

D_x – Diagonal matrix of x

z – Stacked vector of sensed relative positions by agents

$\hat{z}$ – Stacked vector of inter-agent distances

d – Vector of desired goal distances between agents

p – Matrix of agent's locations

u – Input to robot formation

e – Vector of inter-agent distance errors

S_1 – Matrix used for control law with estimator

μ – Mismatch between agents

$\hat{\mu}$ – Estimating variable

z_{values} – Matrix with information about neighbouring agents

n – Number of agents

O – Output matrix of LIDAR sensor

2 Introduction

A robot formation consists of a group of robots moving as a collective where each robot maintains their own individually determined position and orientation. The research field of multi-agent systems is involved in the development of control methods for such formations to expand on their efficiency and their possible uses for complex automated tasks. The uses of formations can already be observed in existing industries. Figure 1 shows various instances among which the intel's Shooting Star drone display at the winter Olympics of 2018 and formation flight of military aircrafts. The difference between the two mentioned uses of formation control is that the former is an example of autonomous control, meaning that the drones are not being directly controlled by a person, instead they rely on other input systems.

Autonomous control can then be further divided by the input the robots use to decide its motion path. The example of Intel's Shooting Star drones is a display of pre-programmed motion patterns. However, in more complex dynamic systems that require the formation to react to undetermined and new environments, a strictly pre-planned path would not suffice. Hence, a different approach is needed, based on different variables from the robot's location within the formation. For this purpose, robots are equipped with sensors that provide information about its environment and algorithms are made to process data from the sensors and control the robots motion.

The analysis and implementation of dynamic formations, that is, a formation capable of re-configuring its number of existing agents automatically, seems to be lacking in the current literature of distributed formation control. This thesis will explore an algorithmic approach to achieving a dynamic formation with LIDAR sensors using previously designed gradient-based control laws.

Within the University of Groningen, the discrete technology and production automation (DTPA) laboratory is contributing to the body of knowledge of multi-agent systems through research into applied control of multiple robots. For this thesis, their facilities and resources were used in order to develop and apply an algorithm suitable for the controlled addition of an external agent to an existing formation.

2.1 Thesis Structure

The goal of this thesis is to design, test and implement an algorithm suitable for the formation control of a multi-agent system in the presence of new agents. To explain how this is achieved this report will use the following outline: First, the problem context is outlined to determine the existing challenges of the research goal, including a systemic view of the problem and a stakeholder analysis. Secondly, a literature review explaining the different mathematical tools that was be used in the control of the robots is done. Followed by the method used to develop the desired algorithm and an analysis of its performance in a simulated environment and experimental setup. And, to conclude, a discussion on the results compared to the initial research question, suggestions for future research and the conclusion.

Figure 1: Uses of Formation. The top-left image displays a human controlled formation of military aircrafts. The top-right image is the natural phenomenon of bird flocking. The bottom-left image is the display of Intel's Shooting Star drones at the winter Olympics of 2018. The bottom-right image is a conceptualization of the European Space Agency LISA mission to detect gravitational waves

3 Problem Analysis

This section will expand on the topic of autonomous formation control and list the challenges that are found in this field.

3.1 Autonomous Formation Control

The control of a network of agents has already been studied in multiple instances and many methods for ensuring the agents' proper behaviour have been developed. Among these are: leader-follower approach, virtual structure approach, behavioural approach, position-based approach, bearing-based approach, distance-based approach and displacement-based approach. Each one of these methods has its own benefits and drawbacks, as covered in Siemonsma's paper (Siemonsma, 2017), making the choice of methodology based on a case-by-case basis. This paper will focus on the distance-based approach. This method utilizes the distance between agents in order to obtain the desired shape of the formation. Therefore, only the distance between neighbouring agents relative to each agent's local frame is required to be known. However, although less advanced sensing equipment is needed, it requires the formation to be rigid by definition, a concept that will be further explained in Section 4.

Apart from the choice in formation keeping approaches, another detail to be determined is if the data should be processed through a central system, which knows all the agent's data, or

a distributed system, where each agent is responsible for its own local data and nothing is shared with its neighbours. The details of distributed control will be elaborated on in Section 3.2, however, in the case of this thesis it was a requirement of the problem owner to utilize distributed control as it is the currently implemented system at the University of Groningen.

Creating a formation of robots is simpler when the initial configuration of the overall formation, such as number of agents and initial locations of said agents is known. In the case of a dynamic formation the problem lays in the uncertainty of when and where a new agent could appear to adhere to the formation, as well as, how the formation should react to it. Different measures need to be taken in order to avoid agents detecting unwanted neighbours, such as limiting the range of sensors. These requirements also limit the shapes that a formation may take as global knowledge of the formation is not known. And finally, as a new agent joins the formation it is important that it does not disrupt the existing formation to the extent that it breaks it. To solve this final problem, it could be possible to limit the reaction of the agents within the formation to allow the newly detected robot to reach its own equilibrium position before the formation reacts to its motion, however the addition of this process adds complexity to the solution.

A problem that may surge in the application of possible formation controllers is the presence of noise in the sensor data. Therefore, to solve this problem, initially a simulation of the agents within the formation was set to show the operation of the proposed approach, allowing for idealized conditions where the focus lays on the controller itself rather than adjusting for unpredictable variables.

3.2 Distributed Control

Data coming from different agents within a formation can be analysed in a central unit, assuming that all agents can communicate or that the information of all agents is available, or within each of the agents themselves. The later methodology is known as distributed control and can make the formation more flexible, easier to implement and reduce computational loads as agents only process the information of its neighbours.

The implementation of distributed control requires specific rules as the information available to each agent is limited. Usual control methods developed involve graph rigidity and are usually based on gradients of potential functions, among which de Marina's thesis (de Marina Peinado, Cao, & Jayawardhana, 2016) who proposes the control law explained in Section 4.4 that will be used within this thesis.

However, distributed control has shown to bring practical problems at the time of implementation, such as, the possible distorted formation shape and constant drift of the group caused in cases where agents disagree on their readings (Belabbas, Mou, Morse, & Anderson, 2012) (Sun, Anderson, Mou, & Morse, 2013).

3.3 Stakeholder Analysis

The way the problem will be approached depends on the entities that will be affected by the outcome of the research, especially the ones that are considered to be more influential. Apart from the problem owner other stakeholders include: research staff and researchers of the DTPA lab at the University of Groningen, Mauricio Muñoz Arias and the Region of Smart

Factories. Due to the organizational nature of the project the stakeholders are put into groups, however, in the following subsections more details are given about who is encapsulated in each division, finishing with Figure 2 to visually represent the influence of the stakeholders.

3.3.1 Problem Owner Analysis

In this case Bayu Jayawardhana will be considered as the problem owner and main stakeholder interested in positive results. This decision is based on the fact that he has created the request for the project, he will follow it closely as it progresses, and the outcome could allow him to create future research projects and possibly expand the functionality of the Nexus robots. His goal is the creation of a new algorithm to govern the robot formation, however, he may constrain the way the problem is approached by limiting the software under which the programs may be built as he may require all equipment to work under one standard to avoid problems in the future.

3.3.2 Research Staff at the DTPA Lab

The research staff of the DTPA lab control the laboratory where the Nexus robots will be studied and, therefore, are involved in the project through as a supplier and sponsor, under this category fall Jacquelien Scherpen and Ming Cao. Although their involvement in the project will be low, considering that they there are not concerned with its development, they have potential power to limit or provide access to the DTPA facilities and resources. Therefore, it is important within the project to ensure appropriate use of the provided resources and not damage any equipment.

3.3.3 DTPA Lab researchers

The DTPA lab researches, as a group, will provide help in the learning process and support with operation of the equipment. Student researchers tend to be at the laboratory a considerable amount of time and, when possible, given their expertise, they can provide information about the general procedures in the control of the robots. As stakeholders they are considered to be relatively interested given some of them are also working on projects related to the Nexus robots, however, even if they may provide help they cannot directly influence the final goal of the project itself.

3.3.4 Mauricio Muñoz Arias

As second supervisor, doctor Muñoz will provide help related to the design methodology and will advise professor Jayawardhana on the final evaluation of the project. His position sets him in the lower right corner in Figure 2, as his influence on the project will be low since he is not concerned with the exact goals of the problem, although he is still interested in a positive outcome, meaning that consideration of his advice and demands should be shown.

3.3.5 Region of Smart Factories

It should also be considered that any potential Investors that may see a use in further research into the technology could be a potential stakeholder. These entities fall into the category of “Smart Factories” which includes companies such as Phillips Drachten and, other companies that form the “Region of Smart Factories” and similar entities could be highly interested in

investing in the project, however due to the maturity of the current project, this is not expected.

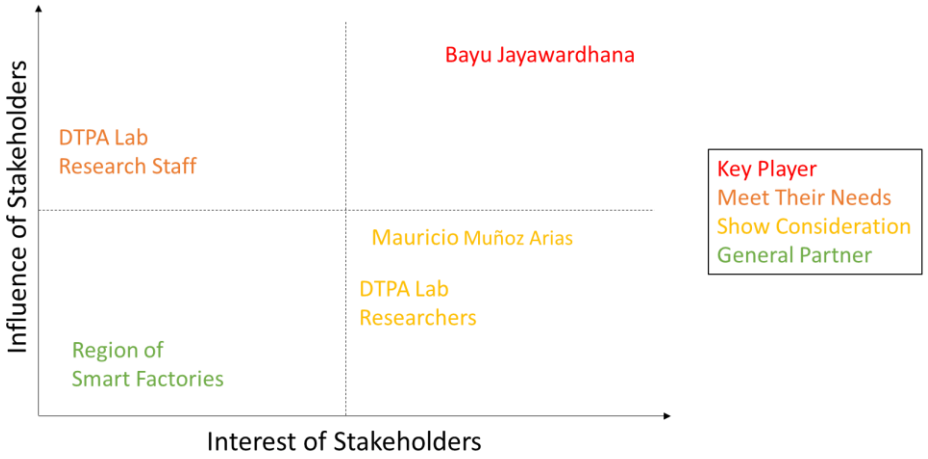


Figure 2: Stakeholder Map

3.4 System Analysis

Given that the formation will be composed of multiple robots, a schematic view of the system will include multiple elements representing each agent within the formation. Each agent is comprised of three parts, its sensors, that will provide information about the other robot's locations, its controller, that will process the data from the sensors and output values to make the robot stay within the formation, and actuators, to move the robot.

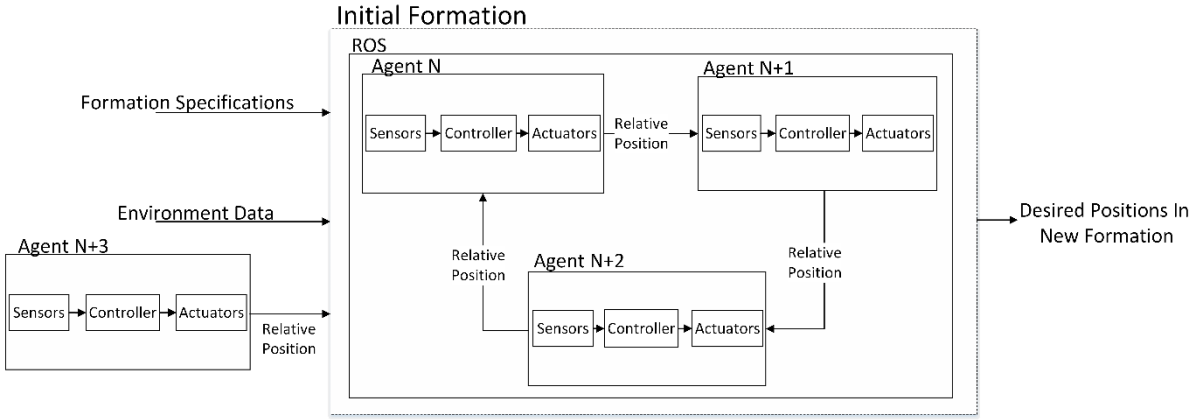


Figure 3: Formation System View

The agent elements will be at the same time part of the ROS system, the software that will control their operation, both in simulations and real-life tests. The inputs into this system will be: the formation specifications, such as desired distance between agents, the environment's data, which includes obstacles and other robots, and finally, the extra agent that wants to adhere to the formation.

The final output of the system will be the desired locations of the agents as their number in the formation remains static, then, as a new agent appears as input the outcome will still result in obtaining the desired positions of all agents within the formation, however this time with one more agent within it.

3.5 Research Question

The current research question investigated in this paper is: “How can a formation of multiple agents be distributedly controlled and reconfigured in order to react to an additional agent?”.

3.5.1 Sub-Questions

In the pursuit of answering the main research question previously set, the following sub-questions will be covered:

- What actions should the robots take to ensure the formation is maintained?
- What characteristics indicate that a formation has been achieved?
- How can the formation performance be evaluated?
- How should agents distinguish neighbouring agents?
- How should new agents adhere to the existing formation?
- How should the formation react to the addition of a new agent?
- How scalable is the reconfiguration method in dealing with large numbers of new agents?
- How does it perform in an experimental setup?

4 Literature Review

Given that this paper will build on the research of previous knowledge generated within, and outside of, the University of Groningen, this section will cover essential information gathered from relevant papers and will introduce some of the topics related to formation control which will be used to create the desired solution later on in the thesis.

Within the University of Groningen some papers have already been published on the distributed control of a Nexus robot formation. Among these papers is the PhD thesis of Hector de Marina: “Distributed Rotational and Translational Maneuvering of Rigid Formations and Their Applications”. In his paper he designed a controller for the nexus robots to create a formation, and in doing so he also covered some of the basics required to maintain a formation in terms of graph theory. This knowledge will be applied within this paper in order to ensure the formation is kept as new agents join and his controller will be used to focus this research mainly on the addition of a new agent rather than the formation itself.

4.1 Notation

Throughout this thesis specific notation will be used, this section will expand on its meaning. The number of dimensions in which motion will take place will be limited to two dimensions, hence $\mathbb{R}^2$. I_2 representing a 2-dimensional identity matrix. For a given matrix $A \in \mathbb{R}^{n\times p}$, define $\bar{A} \triangleq A \otimes \mathrm{I}_2 \in \mathbb{R}^{n2\times p2}$, where the symbol $\otimes$ denotes the Kronecker product. We denote $\|x\|$ the Euclidean norm of the vector x. The $col(\ )$ operator denotes the column vector collecting all the arguments as the vector’s components. For a stacked vector $x \triangleq [x_1^T \quad x_2^T \quad x_3^T \quad \cdots \quad x_k^T]^T$ with $x_i \in \mathbb{R}^{n\times 1}$, $i \in \{1,\dots,k\}$, the diagonal matrix is defined as $D_x = diag\{x_i\} \in \mathbb{R}^{kn\times k}$. The symbol $|\mathcal{E}|$ represents the cardinality of edges. The elements of a matrix R are represented by the r_{ik} where the subscripts i and k indicate the i^{th} row and k^{th} column.

4.2 Graph Analysis of Formations (de Marina Peinado, Cao, & Jayawardhana, 2016)

Given the nature of formations they can be initially described as basic graphs, where each agent is represented as a vertex and the distance between the agents form the edges of the graph. Considering a formation of n robots, with location $p_i \in \mathbb{R}^2$ and $i \in \{1, \ldots, n\}$, where each robot is able to determine the location of its neighbours. A graph $G = (\mathcal{V}, E)$ with vertex set $\mathcal{V} = \{1, \ldots, \mathrm{n}\}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the undirected graph representing the neighbour relationships of the agents. N_i is the set of neighbours of robot i defined by $\mathrm{N} \triangleq \{\mathrm{j} \in \mathcal{V}: (\mathrm{i}, \mathrm{j}) \in \mathcal{E}\}$. Then the elements of the incidence matrix B for the graph $\mathbb{G}$ can be built with the following conditions:

$$b_{ik} \triangleq \begin{cases} +1 & if \quad i = \mathcal{E}_k^{tail} \\ -1 & if \quad i = \mathcal{E}_k^{head} \\ 0 & otherwise \end{cases}$$

where $\mathcal{E}_k^{tail}$ and $\mathcal{E}_k^{head}$ represent the tail and head nodes of edge $\mathcal{E}_k$. Once the graph has been set, the framework $(\mathbb{G}, p)$ can be defined, where p is the stacked vector of the agents' locations p_i. The sensed relative positions by the agents can be represented by another stacked vector

$$z = \bar{B}^T p$$

and the inter-agent distances are

$$\tilde{z} = \underset{\forall k \in \{1,\ldots,|\mathcal{E}|\}}{\mathrm{col}} \sqrt{z_i^2 + z_j^2}$$

where the values of $\tilde{z}_k = \sqrt{z_i^2 + z_j^2}$ correspond to the relative position associated with the k^{th} edge $\mathcal{E}_k = (i, j)$.

A matrix d is defined which contains the desired distances to be obtained by each edge. The values of d_k are dependent on the desired distance and can differ for every edge, however for this thesis all values of d_k will be considered to have the same size. This vector is defined as follows:

$$d = \underset{\forall k \in \{1,\ldots,|\mathcal{E}|\}}{\mathrm{col}} d_k$$

Finally, a formation is achieved when the values of $\tilde{z}$ are of the same magnitude as the corresponding values of d, indicating that the current inter-agent distances are equal to the desired values. The process on how to achieve so with control laws will be introduced in Section 4.4.

4.3 Formation Rigidity

Ensuring the formation keeps its desired shape can be done through the application of what is known in graph theory as "graph rigidity". The explanation for this concept originates from (de Marina, Jayawardhana, & Cao, 2016).

A rigid formation is simply a formation whose corresponding framework $(\mathbb{G}, p)$ is distance rigid or just rigid. Roughly speaking the purpose of using a rigid formation is that it is

technically not possible to smoothly move one node of the framework without altering the position of the rest while maintaining the inter-agent distances. For a more extended proof on the concept, the topic is further explained in de Marina's thesis (de Marina Peinado, Cao, & Jayawardhana, 2016).

4.3.1 Henneberg Insertion

In $\mathbb{R}^2$ an infinitesimally and minimally rigid framework can always be constructed through the Henneberg insertion (Lebrecht, 1911) through the following operations:

- Add a new vertex to the graph with edges connecting to two previously existing vertices
- Subdivide an edge of the graph, and add an edge connecting the newly formed vertex to a third previously existing vertex

Roughly speaking, through this algorithm, when a new vertex needs to be added to a existing graph, two new connections should be made. For a more extended proof on the concept, the topic is further explained in de Marina's thesis (de Marina Peinado, Cao, & Jayawardhana, 2016).

4.4 Hector de Marina's Controller

Now that the basics have been set, it is possible to describe the gradient based distributed control laws within de Marina's paper. To avoid complexity in the analysis and design of the system the dynamics of the formation control are defined by the following first order system:

$$\dot{p} = u$$

where u represents the stacked vector of control inputs $u_i \in \mathbb{R}^2$ for $i = \{1, \dots, n\}$ and $\dot{p}$ is the differential of the agents positions and by extension the velocity of the agents.

Each edge $\mathcal{E}_k$ is assigned a potential function V_k with a minimum at the desired distance $\|z_k^*\|$. Therefore, the potential function can be defined as:

$$V(z) = \sum_{k+1}^{|\varepsilon|} V_k(z_k)$$

by defining $\nabla_{\mathrm{x}} \triangleq \left[\frac{\partial}{\partial x_1} \quad \dots \quad \frac{\partial}{\partial x_m}\right]^T$ it is possible to obtain the gradient descent control:

$$u_i = -\nabla_{\mathrm{p_i}} \sum_{k+1}^{|\varepsilon|} V_k(z_k)$$

by combining this last expression with the definition of z the following closed-loop dynamics can be obtained:

$$\dot{p} = -\bar{B} D_z D_{\tilde{z}} e = -R(z)^T D_{\tilde{z}} e$$

$$\dot{z} = \bar{B}^T \dot{p} = -\bar{B}^T R(z)^T D_{\tilde{z}} e$$

$$\dot{e} = l D_{\tilde{z}} D_z^T \dot{z} = -l D_{\tilde{z}} R(z) R(z)^T D_{\tilde{z}} e$$

where $e, \tilde{z} \in \mathbb{R}^{|\mathcal{E}|}$ are the stacked vectors of e_k and $\|z_k\|^{-1}$ for all $k = \{1, \dots, |\mathcal{E}|\}$ and $R(z) = \bar{B}D_z$ represents the rigidity matrix of the formation. For a more detailed explanation of this controller and extensive proofs on how to obtain the previous equations refer to de Marina's paper.

Therefore, the control law used within this thesis is the following:

$$u = -c_1 \bar{B} D_z D_{\tilde{z}} e$$

where c_1 is a gain.

4.5 Controller with Estimator

Since the dynamics of a physical formation will most likely be affected by errors in measurements or calibration of the sensing tools of the agents, this could cause mismatches in the goals of neighbouring robots, leading to a constant drift as two agents attempt to achieve an equilibrium distance they disagree on. To explain this concept, we start by showing how the input into the agent will be altered due to a mismatch μ :

$$u = -c_1 \bar{B} D_z D_{\tilde{z}} e - \bar{S}_1 D_z \mu$$

where μ_k is a mismatch in the readings between two robots, $\mu \in \mathbb{R}^{|\mathcal{E}|}$ is the stacked column vector of μ_k for all $k \in \{1, \dots, |\mathcal{E}|\}$, and the elements of $S_1 \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{E}|}$ are derived from $\mathbb{G}$ as follows:

$$s_{1_{ik}} \triangleq \begin{cases} 1 & if \quad i = \mathcal{E}_k^{tail} \\ 0 & if \quad i = \mathcal{E}_k^{head} \\ 0 & otherwise \end{cases}$$

The agents located at $\mathcal{E}_k^{tail}$ will be the local estimators, meaning that within the matrix S_1 the estimating agents are defined, and these agents can easily be changed by changing the direction of the corresponding arrow in $\mathbb{G}$. The task of the estimating agents will be to estimate and compensate the mismatch μ_k with the agent located at its respective position $\mathcal{E}_k^{head}$. To do so an estimator $\hat{\mu}_k$ is introduced, therefore to counterbalance any mismatches the following control input function is introduced:

$$u = -c_1 \bar{B} D_z D_{\tilde{z}} e - c_2 \bar{S}_1 D_z (\mu - \hat{\mu})$$

Where c_2 is a gain, and the estimating agent will calculate a value for $\hat{\mu}_k = \mu$ and counterbalance the disagreements in readings. In simple terms the calculation of $\hat{\mu}_k$ can be reduced to

$$\dot{\hat{\mu}}_k = e_k - \mu_k$$

$$\hat{\mu}_{k_{t+1}} = \hat{\mu}_{k_t} + \dot{\hat{\mu}}_k$$

where t and $t + 1$ indicate the initial and following iteration. However, for a more extensive proof, de Marina expands on it in his paper.

4.6 Illustrative Example

Considering a formation of three agents as shown in Figure 4, this subsection will make use of the previously introduced notation and theory to illustrate its functionality and clarify its meaning.

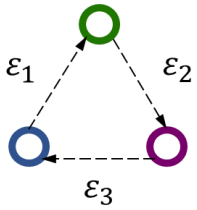


Figure 4: Formation of three agents

The formation under consideration has an incidence matrix defined by:

$$B = \begin{bmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$$

With matrix p as follows:

$$p = [x_1 \quad y_1 \quad x_2 \quad y_2 \quad x_3 \quad y_3]^T$$

where the values inside the matrix p indicate the x and y coordinate of the respective agents.

With matrix z and $\tilde{z}$ as follows:

$$z = \bar{B}^T p = [x_1 - x_2 \quad y_1 - y_2 \quad x_2 - x_3 \quad y_2 - y_3 \quad x_3 - x_1 \quad y_3 - y_1]^T$$

$$D_z = \begin{bmatrix} z_1 & 0 & 0 \\ z_2 & 0 & 0 \\ 0 & z_3 & 0 \\ 0 & z_4 & 0 \\ 0 & 0 & z_5 \\ 0 & 0 & z_6 \end{bmatrix}$$

$$\tilde{z} = \left[\frac{1}{\sqrt{(z_1 - z_2)^2}} \quad \frac{1}{\sqrt{(z_3 - z_4)^2}} \quad \frac{1}{\sqrt{(z_5 - z_6)^2}}\right]^T$$

therefore, making $\tilde{z}$ a matrix of the reciprocal of the inter-agent distances and e:

$$e = [\tilde{z}_1 - d_1 \quad \tilde{z}_2 - d_2 \quad \tilde{z}_3 - d_3]^T$$

Given these matrices is now possible to calculate the input u. However, in the case of application of estimators the following matrices are needed:

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mu = [\mu_1 \quad \mu_2 \quad \mu_3]^T \qquad \hat{\mu} = [\hat{\mu}_1 \quad \hat{\mu}_2 \quad \hat{\mu}_3]^T$$

where S represents the estimation pattern of a cyclic formation as indicated in Figure 4 by the direction of the edges.

5 Expansion of Existing Formation

In this section the methodology implemented to make a formation dynamic, and therefore able to incorporate new agents, will be described.

5.1 Detection of Agents

The implementation of an algorithm to include additional agents to an existing formation can be approached in multiple ways, however, in formations not using distributed control it can become exceptionally taxing for the central processing unit to adjust the formation as the number of agents increases. Given that distributed control will be applied in this research it is possible to include a new agent while only requiring extra processing from its neighbours and not the whole formation.

Given that distributed control will be used, the only information available to each agent is its number of neighbours, the angle at which its neighbours are located relative to the agent in consideration and their distance. To start with, the smallest initial formation possible is a triangular shape with 3 agents and 3 edges. The addition process of one robot to said initial formation would require the addition of two new edges to maintain the formation's rigidity, as explained through the Henneberg insertion in Section 4.3.1. This process of adding two new edges can continue for n agents, meaning that the formation should always have an amount of edges equal to "$n \times 2 - 3$". For this thesis it was decided not to create square formations or other shapes as it complicates the process of neighbour selection.

As the previously described process continues a triangular lattice would be obtained as shown in Figure 5, meaning that the maximum number of neighbours for one agent will be 6.

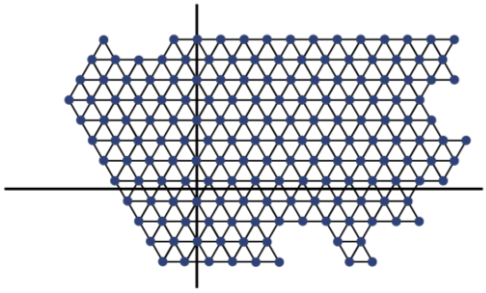


Figure 5: Triangular Lattice (representation of formation of n agents using the introduced algorithms in the following sections)

To control such a formation, it is important that the algorithms are able to detect n number of robots and are able to adapt its connections if a new agent appears. Different limitations will occur depending on the used detection methods. For this research a LIDAR system will be used due to its ability to scan the agent's surrounding environment, however, the limitation of such a system is its inability to easily differentiate between obstacles and new agents, therefore for testing purposes agents will be subject to empty environments where the only objects are other potential agents.

Before analysing the data obtained from the LIDAR scanner the process it goes through to generate data will be outlined as it also affects how and if new robots will be included into the formation. As shown in Figure 6 the laser starts at reference point 0 and revolves around

its axes anti-clockwise until it completes a full revolution, at which point it will have gathered G data points depending on its resolution. Any agents in its path will be identified and their distances will be recorded resulting in a 1 by G matrix. If any new agents appear in the range of the laser they will be detected and added onto the matrix. For this thesis, the value of G is assumed to be 360 to simplify the calculations. In the case of different resolutions minor changes would need to be applied to the algorithm to ensure the proper calculations are being performed.

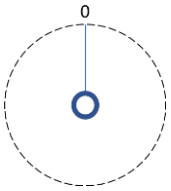


Figure 6: Representation of LIDAR process around agent with its starting reference point at 0

The relevance of the initial reference point of the LIDAR is due to its influence on the addition of robots. As a new robot appears in the laser's range it will be detected but not be directly added as the last entry in the z_{values} matrix, instead, if it is the second robot to be detected, it will take the place of the previous robot in its position and the previous agent occupying that position will be switched to third place, therefore altering the row order of z_{values}.

5.2 Formation Control with Addition of Agent

This subsection will elaborate on the algorithms to include an extra agent into an existing formation. These algorithms will be divided into two sections, a "Data Processing Node" where the matrix received from the LIDAR scanner is analysed to extract the desired information about the neighbouring agents and a "Control Node" where the data created in the previous node is assembled into the matrices needed to generate the input u. The distinctive task of the algorithms to be introduced is their ability to adapt the needed matrices to the appearance of new agents.

The process to make the agents detect each other requires specific notation. n is the number of neighbours of an agent. $R \in \mathbb{R}^{1\times n}$ is a cell of arrays containing vectors of the neighbours' angle locations. The algorithm considers the first index count of a matrix to start at 0. The operator $R[x]$ indicates the index x of matrix R, while $R[x][y]$ for a cell R indicates the index y of the vector located at the index x of R. The operation $R[x:y]$ indicates the interval of values within R from index x to y. The operator $length()$ outputs the length of a matrix. The operator $[p,r]$ indicates two variables being appended as in the following example:

$$[p,r] = \begin{bmatrix} p_1 & | & r_1 \\ p_2 & | & r_2 \end{bmatrix}$$

5.2.1 Data Processing Node for N

As previously mentioned, the only initial input for the agent to determine how to act is the matrix $O \in \mathbb{R}^{1\times G}$ generated by the LIDAR. To begin with, the information of the matrix must be adjusted to meet the requirements of the maximum and minimum ranges in order to avoid undesired readings. Algorithm 1 is designed to enforce those requirements and adjust the agent's vision. It must be stated that the maximum range has the requirement of being larger

than the desired distance d yet smaller than $d + \frac{d\sqrt{3}}{2}$ which will be the distance of its neighbour's neighbours.

Algorithm 1: Application of minimum and maximum ranges

Input: O
Output: O
for $i = 0$ **to** G **do**
    if $O[i] < min\_range$ **then** $O[i] = 0$
    if $O[i] > max\_range$ **then** $O[i] = 0$

Then, once the matrix O has been adapted, a matrix of the indices in which the data is located in O is assembled to determine the angles at which the neighbours are located. Algorithm 2 describes how this can be achieved.

Algorithm 2: Determine angles at which neighbours are located

Input: O
Output: z_a
for $i = 0$ **to** G **do**
    if $O[i] > min\_range$ **then**
        $z_a[i] = i$
    if $O[i] < max\_range$ **then**
        $z_a[i] = i$

Now that the angles of all neighbours are known, they must be differentiated to determine which angles correspond to which agents, as well as calculating how many neighbours there are.

Algorithm 3: Determine number of agents and their locations

Input: z_a
Output: R, n

```
p=0
n=0
for i = p to length(z_a) do
    if z_a[i] − z_a[i + 1] >= −10 and i ≠ (length(z_a) − 2) then
        r = [r, z_a[i]]
    else if −10 <= z_a[i] − z_a[i − 1] <= 10 then
        r = [r, z_a[i]]
        R = [R, r]
        n = n + 1
        p = 1 + i
    else if z_a[i] − z_a[i + 1] >= −10 and i = (length(z_a) − 2) then
        r = [r, z_a[i]]
        r = [r, z_a[i + 1]]
        R = [R, r]
        n = n + 1
```

Within Algorithm 3 the purpose of the for-loop is to calculate the number of agents n and the cell of arrays R. Within the for-loop the purpose of each If condition is as following:

- First "If-loop": observes if the difference between $z_a[i]$ and $z_a[i+1]$ is larger than 10, indicating that the reading belongs to the same agent, and therefore appending it to the r matrix.
- Second "If-loop": in the case that the following reading is larger than 10 the second loop is activated where the last value is appended to r, the finalized r matrix for agent n appended to the cell R, the count of neighbours increases by 1 and the placeholder p is increased to the value $i+1$ in order for the for loop to continue with the values of the next agent.
- Third "If-loop": ensures that when the loop has reached the ending of matrix z_a the last agent is accounted for.

Now that the number of agents and their locations are known it is possible to determine the x and y distances. Algorithm 4 first creates a matrix of average angles z_{ax} and a matrix of the closest distances z_{nx}. To obtain z_{ax} the mean value of each neighbour's angle locations is taken and rounded to obtain one average value. Then, in order to determine the distances to the neighbours, the minimum value of the matrix O at that specific agent's location is extracted to represent the inter-agent distance.

Algorithm 4: Analyze agent data

Input: R, O
Output: z_{ax}, z_{nx}
for $i = 0$ **to** n **do**
    $z_{ax}[i] = int(round(mean(R[i])))$
    $z_{nx}[i] = min(O[R[i][0 : length(R[i])]])$

Algorithm 5 is used to calculate the x and y distances to each agent. This step involves the simple calculation using the shown formulas to obtain the $z_x \in \mathbb{R}^{1\times n}$, a matrix with each respective neighbour's x distance, and $z_y \in \mathbb{R}^{1\times n}$ matrix with the y distances

Algorithm 5: Calculate x and y distances to neighbours

Input: z_{ax}, z_{nx}
Output: z_x, z_y
for $i = 0$ **to** n **do**
    $z_x[i] = cos((z_{ax}[i] - 180) * 2 * \pi/360) * z_{nx}[i]$
    $z_y[i] = sin((z_{ax}[i] - 180) * 2 * \pi/360) * z_{nx}[i]$

Algorithm 6 is used to assemble a matrix of the distances to each agent, the x and y distances in order to feed the controller with the final matrix $z_{values} \in \mathbb{R}^{n\times 3}$ and organize the obtained information

Algorithm 6: Assemble matrix of distances for controller

Input: z_{nx}, z_x, z_y
Output: z_{values}
$z_{values} = [z_{nx}, z_x, z_y]$

5.2.2 Controller for N

Consider the following control law as introduced in Section 4.4:

$$u = c_1 \bar{B} D_z D_{\tilde{z}} e$$

Provided the analysed LIDAR data in the form of z_{values}, it is now the task of the controller to assemble the required matrices. However, given that the z_{values} matrix is dynamically changing and its size could change at any time an agent is added to the formation, the following Algorithm 7 is designed to accommodate to any possible addition by assembling each matrix through a for-loop.

Algorithm 7: Assemble controller matrices

Input: z_{values}, d
Output: $\bar{B}D_z$, $D_{\tilde{z}}$, E

for $i = 0$ **to** n **do**
  $B_x = [B_x, z_{values}[1 + 3 \times i]]$
  $B_y = [B_y, z_{values}[2 + 3 \times i]]$
  $D = [D, (z_{values}[3 \times i])^{-1}]$
  $E = [E, z_{values}[3 \times i] - d]$
$\bar{B}D_z = [B_x^T, B_y^T]$
$D_{\tilde{z}} = diag(D)$

The control input can now be computed as all its required variables have been calculated. Through this algorithm it is possible to maintain a formation of n robots, and in order to test its performance a simulation was performed, and its results can be seen in Section 6.3.1.

Given the theory introduced in the literature review, the following mathematical definition for the choice of each agent's $\bar{B}D_z$ was made, to clarify the selection process for the distributed matrices derived from de Marina's notation. First of all, is the expansion of rigidity matrix $R(z)$ for an overall 3 agent formation:

$$R(z)^T = \bar{B}D_z = \begin{bmatrix} x_1 & y_1 & -x_1 & -y_1 & 0 & 0 \\ 0 & 0 & x_2 & -y_2 & -x_2 & y_2 \\ x_3 & y_3 & 0 & 0 & -x_3 & -y_3 \end{bmatrix}$$

Yet, the matrix for agent n will be limited only to its existing knowledge. Therefore, an agent's individual rigidity matrix can be noted as follows:

$$\bar{B}D_{z_n} = \underset{\forall k \in E(n,i)}{\text{col}} [r_{k(2n-1)} \quad r_{k(2n)}]$$

where i represents all neighbouring agents of n, making $E(n, i)$ the set of edges connecting n with its neighbours.

5.2.3 Controller with Estimator

As discussed in the literature review, with distributed control there is the possibility of reading mismatches causing inconsistent distance goals for each agent. In order to dampen the effect of mismatches, estimators can be implemented.

Applying the following control law requires a few matrices to be determined within the controller node:

$$u = -c_1 \bar{B}D_z D_{\tilde{z}} e - c_2 \bar{S}_1 D_z (\mu - \hat{\mu})$$

First of all, a pattern to choose the estimating agents must be created to avoid agents estimating each other. Since the shape of the formation can change at any given point as a new agent can be added anywhere around the formation, it was decided to reduce the analysis to four basic shapes.

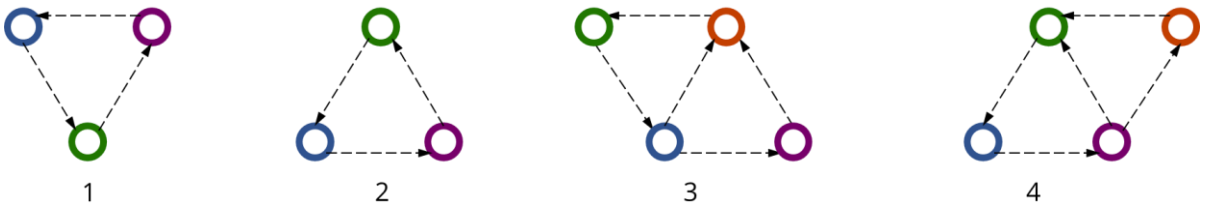


Figure 7: Basic shapes for selecting estimating agents

Initially the approach was to make each agent estimate their first found neighbour with respect to the reference point, assuming all agents have the same orientation and therefore same relative reference point. However, this approach encounters a problem in shapes 1 and 4. To solve this problem each agent should be able to determine more than just which neighbour was scanned first but also their relative locations.

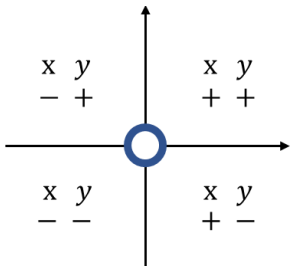


Figure 8: Coordinate framework of an agent

In Figure 8 an agent's coordinate framework is shown. Given these coordinates, some conditions can be created within the controller to differentiate where each neighbour is located, as their x and y directions are stored in the z_{values} matrix.

Firstly, the number of agents that will be estimated by a given agent must be specified. For this design it was decided to either estimate 1 or 2 agents, the first in the case of two available neighbours and the second in case of 3 or more neighbours. The following expression can then be derived:

$$\begin{cases} \hat{\mu} = 0 \;\; if \;\; n = 2 \\ \hat{\mu} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \;\; if \;\; n \geq 3 \end{cases}$$

where the rows of $\hat{\mu}$ indicate how many agents it will estimate. Algorithm 8 provides the methodology used to estimate the correct agents to avoid discrepancies.

The algorithm consists of various If conditions to account for the different locations of neighbouring agents and adapt to changes if new agents appear. The first if condition is a loop to select an agent to estimate in cases of only one neighbour and the second if condition is used in the case two agents must be estimated. For simplicity, the basic function of each loop is explained with a comment within Algorithm 8.

Algorithm 8: Choice of estimating agents

Input: z_{values}
Output: $\hat{\mu}$, $\bar{S}D_z$

$now = time()$
$\Delta T = now - old$
if $\hat{\mu} \in \mathbb{R}$ **then**
    /* Estimate one agent */
    if $z_{values}[1] < 0$ ***and*** $z_{values}[2] < 0$ ***and*** $z_{values}[-2] < 0$ ***and*** $z_{values}[-1] > 0$ **then**
        /* Estimate last found agent */
        $\dot{\hat{\mu}} = 2 \times (E[-1] - \hat{\mu})$
        $\hat{\mu} = \hat{\mu} + \dot{\hat{\mu}} \times \Delta T$
        $\bar{S}D_z = [z_{values}[-2]/z_{values}[-3],\ z_{values}[-1]/z_{values}[-3]]$
    else
        /* Estimate first found agent */
        $\dot{\hat{\mu}} = 2 \times (E[0] - \hat{\mu})$
        $\hat{\mu} = \hat{\mu} + \dot{\hat{\mu}} \times \Delta T$
        $\bar{S}D_z = [z_{values}[1]/z_{values}[0],\ z_{values}[2]/z_{values}[0]]$
if $\hat{\mu} \in \mathbb{R}^{1\times 2}$ **then**
    /* Estimate two agents */
    if $z_{values}[1] < 0$ ***and*** $z_{values}[2] < 0$ ***and*** $z_{values}[-2] < 0$ ***and*** $z_{values}[-1] > 0$ ***and*** $z_{values}[-5] > 0$ **then**
        /* Estimate last and penultimate found agents */
        $\dot{\hat{\mu}} = 2 \times [E[-1] - \hat{\mu}[0],\ E[-2] - \hat{\mu}[1]]$
        $\hat{\mu} = \hat{\mu} + \dot{\hat{\mu}} \times \Delta T$
        $\bar{S}D_z = [z_{values}[-2]/z_{values}[-3],\ z_{values}[-1]/z_{values}[-3];$
        $z_{values}[-5]/z_{values}[-6],\ z_{values}[-4]/z_{values}[-6]]$
    else
        /* Estimate first found agent */
        $\dot{\hat{\mu}} = 2 \times [E[0] - \hat{\mu}[0],\ \hat{\mu}[1]]$
        $\hat{\mu} = \hat{\mu} + \dot{\hat{\mu}} \times \Delta T$
        $\bar{S}D_z = [z_{values}[1]/z_{values}[0],\ z_{values}[2]/z_{values}[0];$
        $0,\ 0]$
$old = now$

After application with more shapes, it was found that this algorithm is not able to correctly select all estimating agents for all shapes as more agents add unpredicted complexity to the shape. Nonetheless, the Algorithm 8 does operate for the suggested simple shapes and some of their expansions.

6 Simulation and Results

This section will cover the application of controllers and algorithms on a computer simulation of a Nexus robot formation to analyse their performance before proceeding to a real-life application.

A simulation of agents using LIDAR scanners was made to obtain the required data. For this purpose, new launch files and settings were created to start applying the controllers. Once multiple nexus robots were being simulated LIDAR scanners of 360 points per rotation were introduced into Gazebo to detect the distance to neighbouring robots. These initial conditions were set as the simulations are based on the available equipment used for the real-life tests as outlined in Section 7.

6.1.1 Software

To test any possible algorithms and controllers, the “Gazebo” physics engine tool was used. This software package allows to simulate the functioning of the agents with realistic three-dimensional world physics. Creating a simulation to test possible solutions is simpler and less time consuming that trying to apply it directly to real robots. Furthermore, within the simulation ideal conditions can be assumed, while testing with robots may not result in initial correct functioning due to high amounts of noise and other unexpected variables.

The Nexus robots require specific commands in order to operate, at the same time these require a specific platform to allow flow of information and that is the “Robot Operating System” (ROS). This open source software provides tools to create robot applications (www.ROS.org, 2017). The version of ROS that will be used, named ROS Kinetic, requires to be run on a distribution of Linux, in this case Ubuntu 16.04.

6.2 Computation Process

To understand how data is being manipulated, this subsection covers the computational graph obtained from the GUI of the “RQT graph” package offered by ROS.

In Figure 9 the various ROS nodes (Orange) and ROS topics (Blue) for one of the robots within the formation are shown.

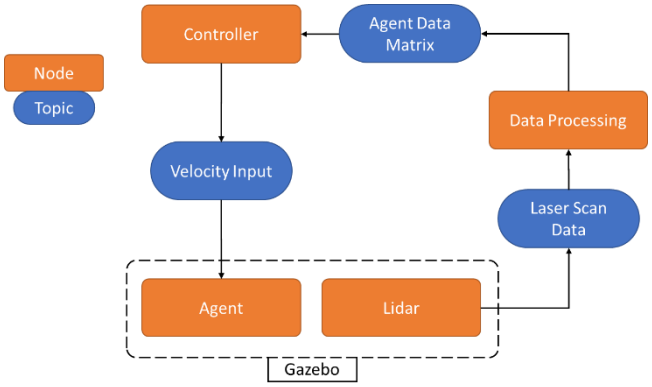


Figure 9: Computational map within one agent

Each node is shortly described to understand their purpose

- The **LIDAR** node is the sensor within Gazebo that scans the environment for obstacles, in this case an obstacle represents an agent in the formation, and outputs a matrix as previously described in Section 5.1.
- The **Data Processing** node receives the data published by the LIDAR node and processes it to determine the angles and distances to other agents, which then publishes as the “agent data matrix” topic in the form of the z_{values} matrix.
- The **Controller** node applies the rules derived from de Marina’s paper and the algorithm described in Section 5.2.2 and output the velocities at which the respective nexus robot should move.
- Finally, the **Agent** node in Gazebo represents the entity of the robot and therefore operates according to the values calculated within the controller.

In a real life experiment the Gazebo environment can simply be replaced by the respective LIDAR scanner and Robotic agent used. For clarity purposes the code used for the data processing node and controllers can be found in the Appendix.

6.3 Experiments

Within this section the performance of the formation controllers is analysed to ensure the operation of the agents functions correctly when applying the suggested algorithms.

6.3.1 Basic Formation Control

Figure 10 shows the visualization of the simulation within Gazebo. Each robot is represented by its similarly shaped yellow forms, while the dark shape over it represents the LIDAR scanner. Each scanner detects the shape of the other scanner in order to determine the position of the other robots. After running the data processing nodes and controllers the robots proceed to move to their desired locations within the formation.

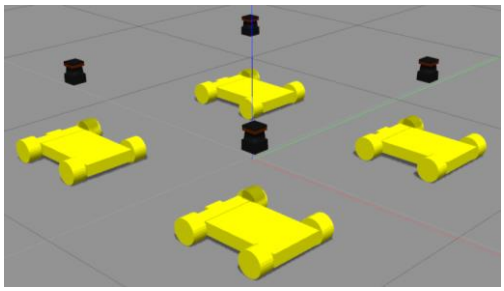


Figure 10: Representation of a four-agent formation within Gazebo

Figure 11 and Figure 12 show the velocities of robots and the inter-agent distances respectively as they proceed to take their place within the formation. From these graphs it can be seen that the controllers do converge, although a small error is present preventing the inter-agent distance from achieving an error of 0. This offset could be due to small variances within the simulation or the GPU of the computer not operating correctly. Nonetheless, the values achieved can be considered to be acceptable as all agents achieve a visible state of equilibrium both in inter-agent distances, which indicates the agents are in proper formation, and in input velocity, which shows the overall formation stays stationary and does not drift. This effect is present in the following sections as well and just as in this case, it will be disregarded as agents achieved an apparent still state within the simulations.

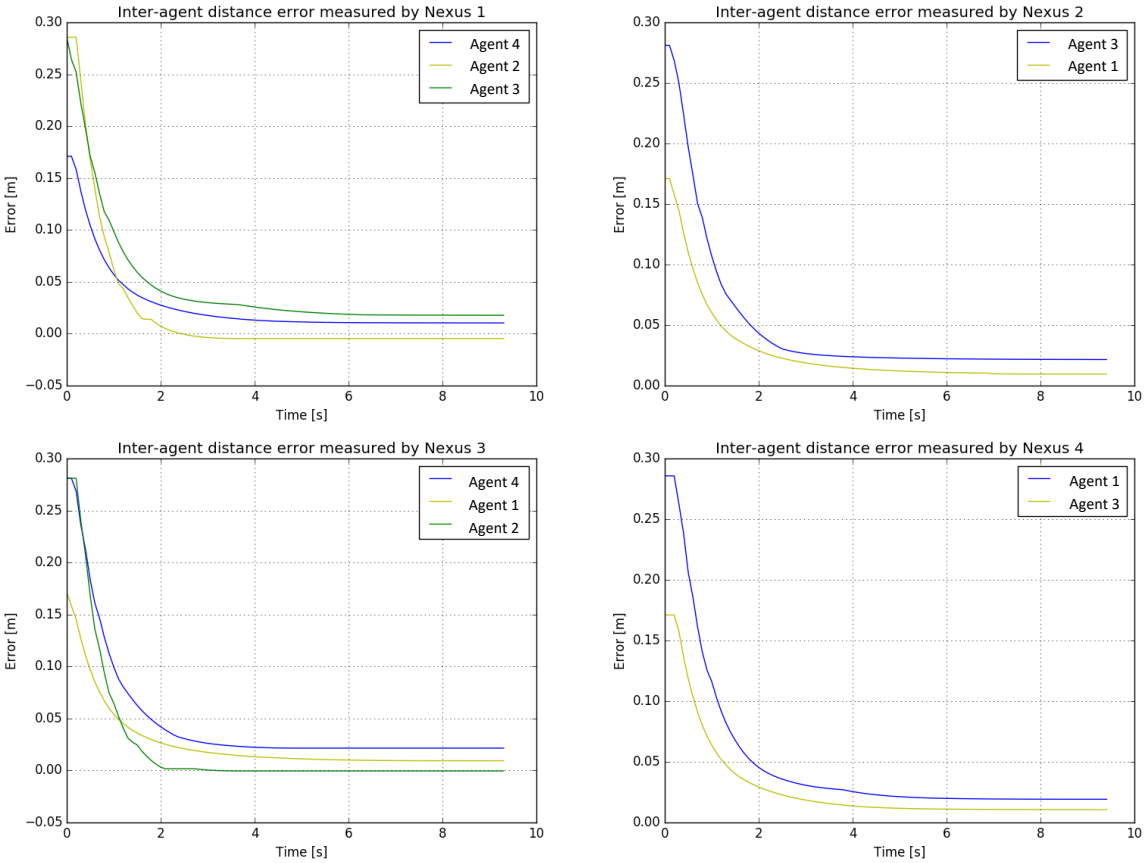


Figure 12: Inter-agent distance errors for each agent in a four-agent formation

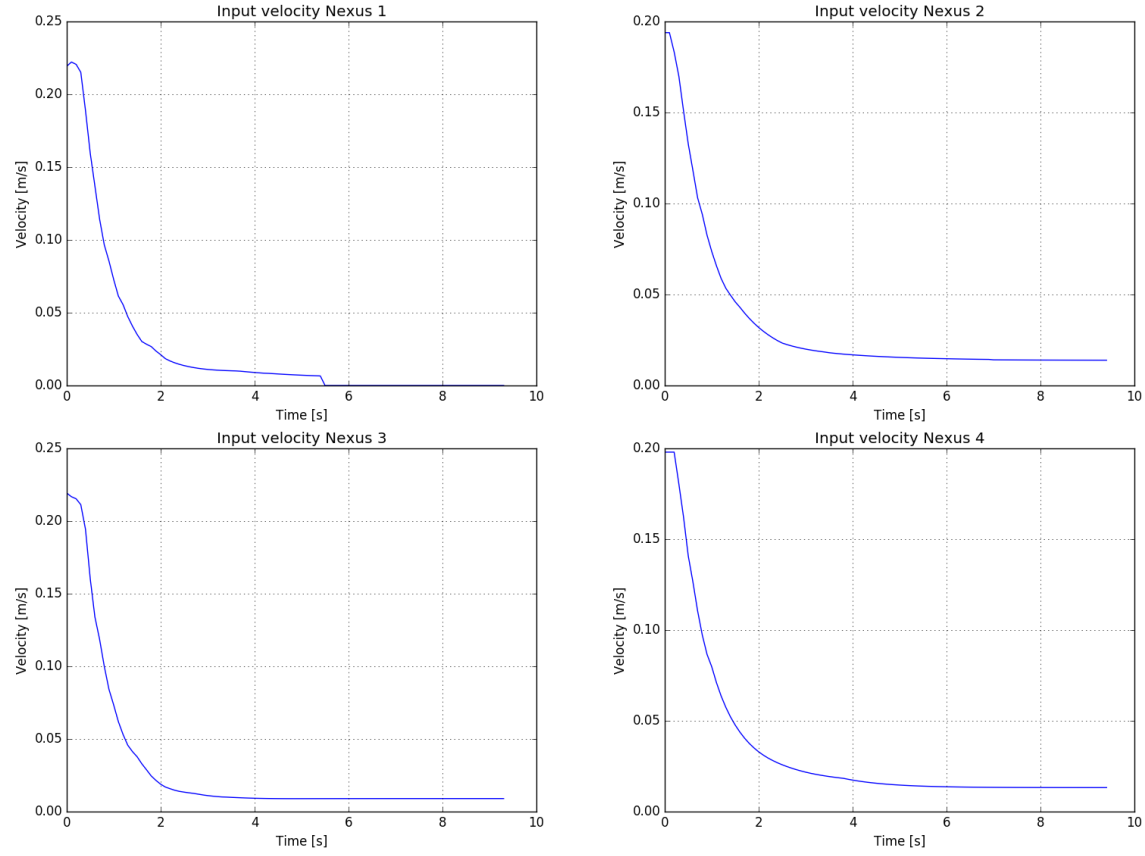


Figure 11: Velocities for 4-agent formation

With the correct operation of this basic formation of four agents the next step is to check the operation of an extended formation with more agents using the same exact controller to show the flexibility of the algorithm.

6.3.2 Extended formation Control

This section shows that the same data processing and controller nodes with the introduced algorithms used in Section 6.3.1 work for an extended number of agents. However, due to computing power limitations the agents simulated were 7 as shown in Figure 13, although it would be expected that any number of agents would still achieve the same results under similar conditions.

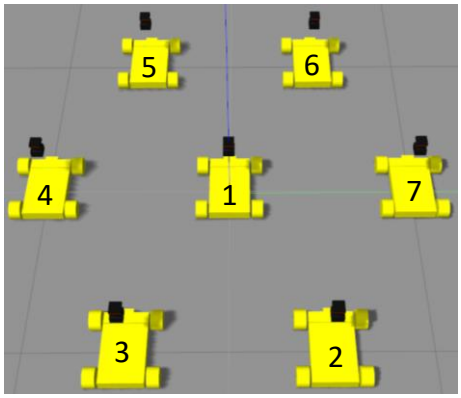


Figure 13: Simulated formation of 7 agents and their identifiers

Each agent's initial position was predefined as shown in Figure 13 and this test will focus on the operation of formation control rather than addition and extraction as covered in Section 6.3.3. The resulting inter-agent distances and velocities of each robot can be seen in Figure 14 and Figure 15 respectively:

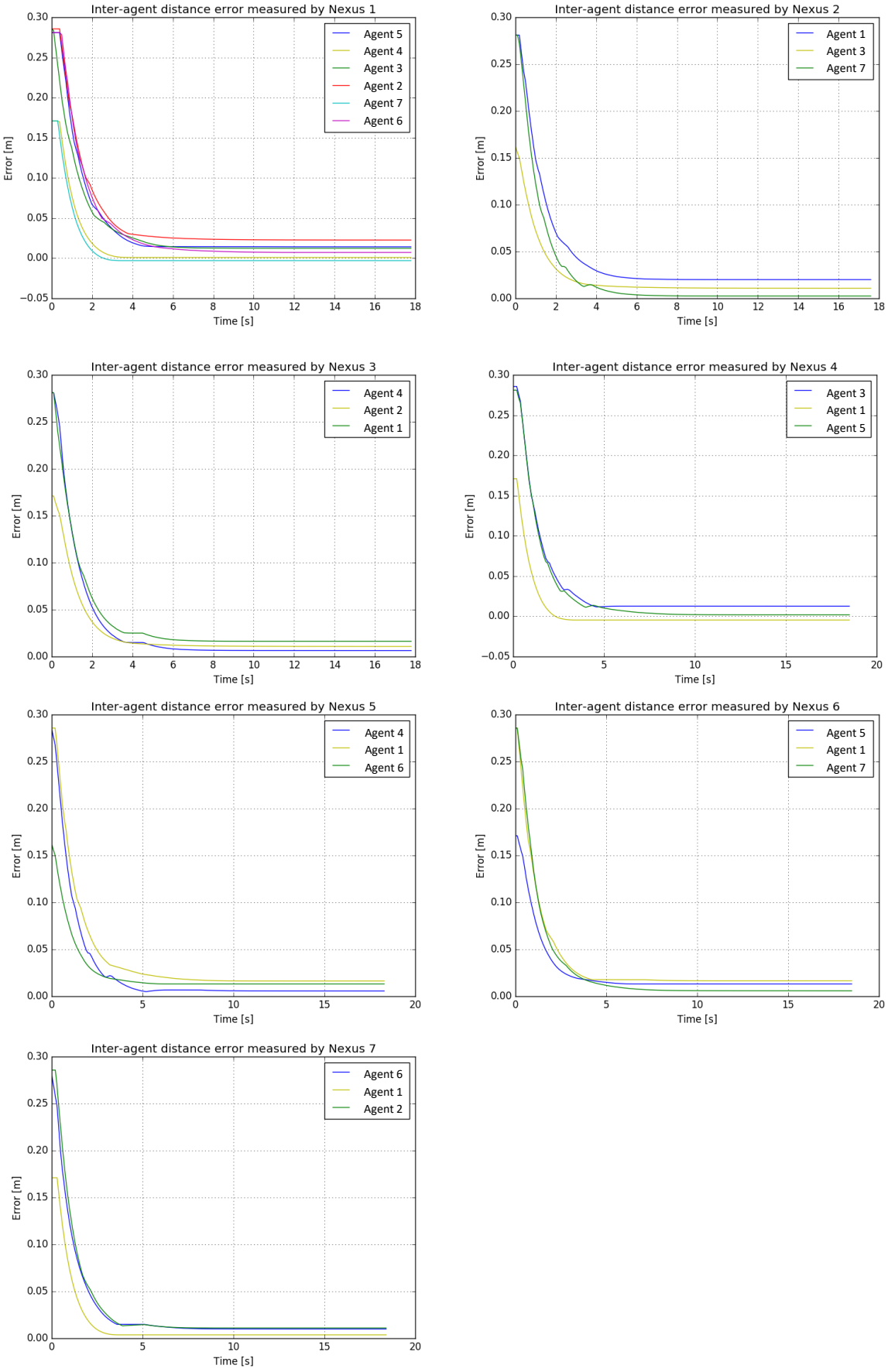


Figure 14:Inter-agent distances for a formation of 7 agents. e1, e2, e3 and so on, respectively represent the agent in their order of detection starting from the initial reference point.

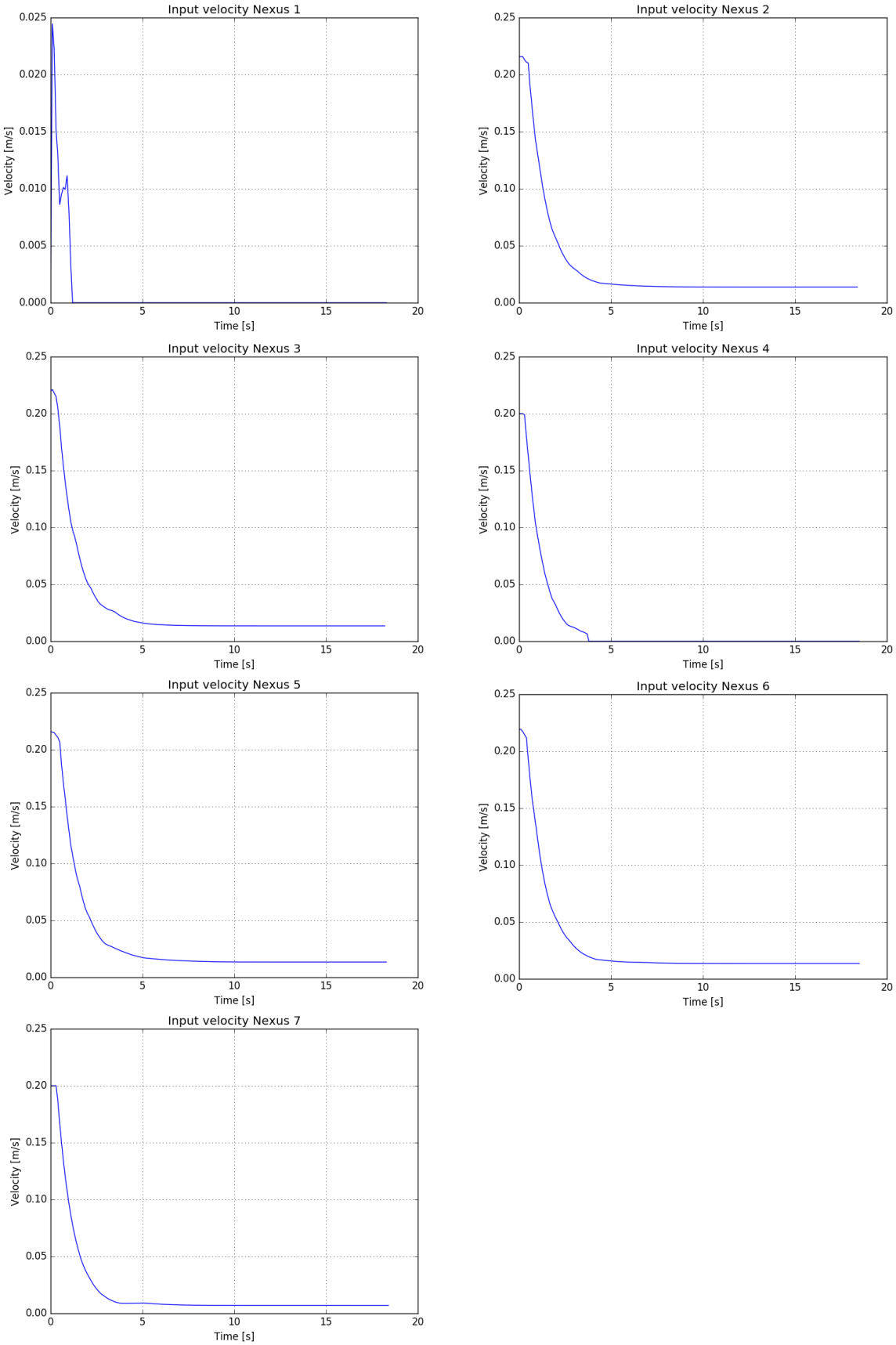


Figure 15: Agent's velocities over time for a formation of 7 agents.

From these results it can be determined that the same controller works for different amounts of robots, showing that with the implementation of the suggested algorithm the formation is flexible and capable of working with an increasing number of agents under ideal conditions.

6.3.3 Addition and Extraction of Agents

Within this section addition of new agents into the formation will be analyzed. Figure 16 shows in two pictures the process the formation will be subjected to to test the introduced algorithms in Section 5.2.1 by adding and extracting an agent to an already defined formation.

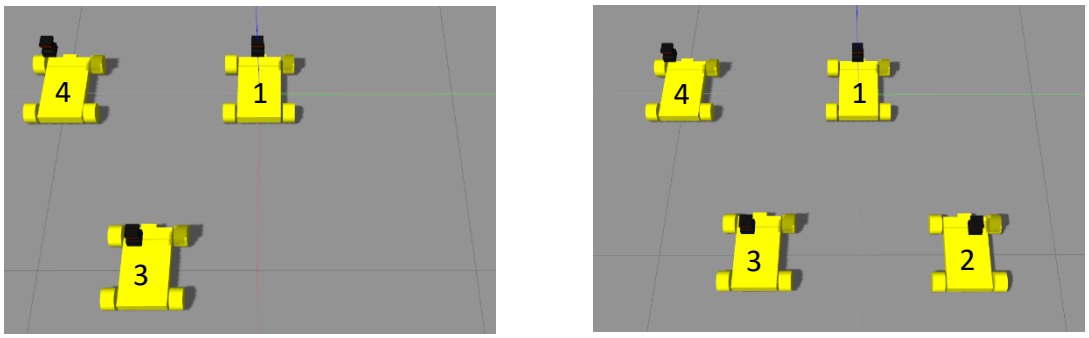


Figure 16: Process of addition of robot. The image on the left shows the initial formation, while the image on the right shows the final formation with the addition of agent 2

The process for adding a robotic agent consisted on the following steps:

- Spawn agents 1, 3 and 4 in Gazebo.
- Start Data Processing Node and Controllers for agents 1, 3 and 4 to obtain the initial triangular formation as shown in Figure 16.
- Give time to formation to achieve equilibrium.
- Spawn agent 2 with Data Processing and Controller Nodes active in the range of agents 1 and 3 to be detected.
- Allow formation to reach the new equilibrium.

The effect of this process can be observed in Figure 18, where the inter-agent distances are plotted over time. It should be noted that the initial value of 0 for the connection with Agent 2 is just just a placeholder while the agent is not detected. Initially, the equilibrium for the first formation is obtained, then, Agent 2 is detected by the formation, which causing the green spike in the plots of agents 1 and 3. Finally, the inter-agent distances achieve equilibrium again, which shows that the addition of a new agent was succesful.

The plots for Agent 2 in Figure 18 and Figure 17 do not have the same x-axis as the rest of the plots due to agent not working while it is not part of the formation. In future revisions of the used code this detail could be adapted, however, in its current state it only displays information of its neighbours once it's already detected them.

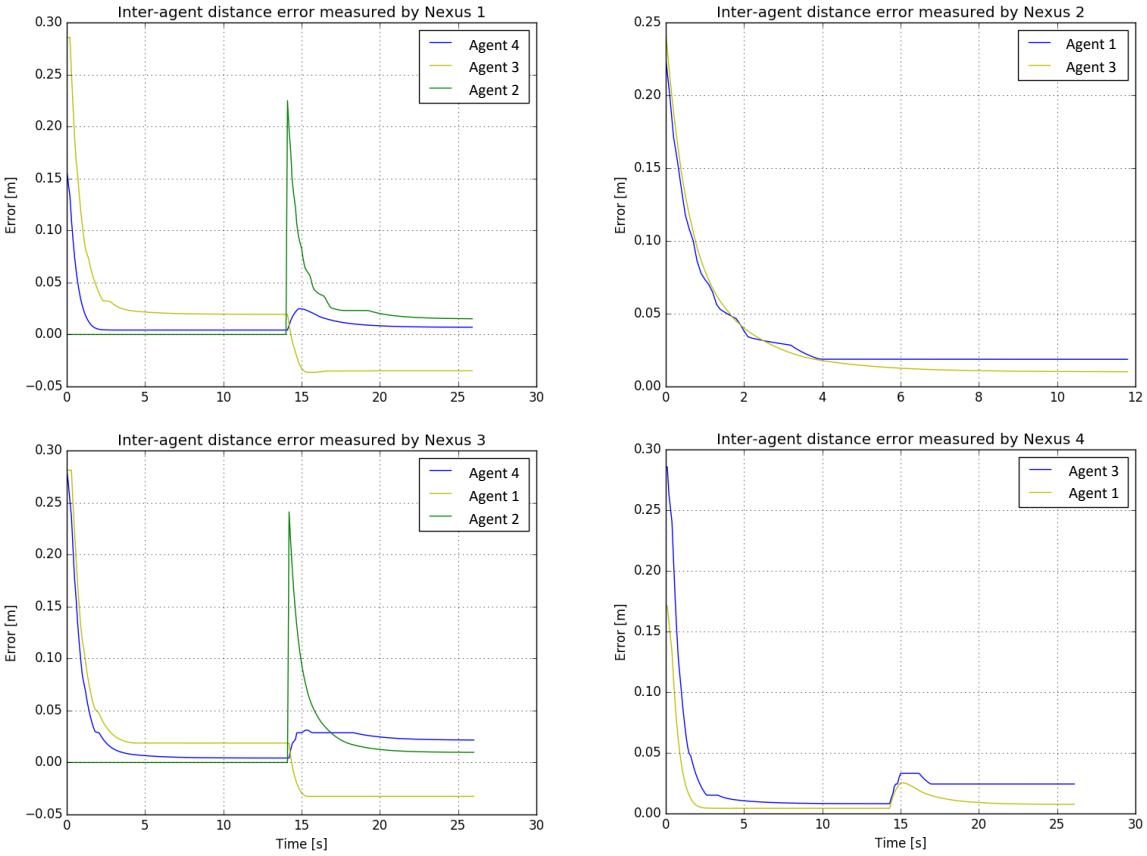


Figure 18: Addition of Agent 2 (Inter-agent Distances)

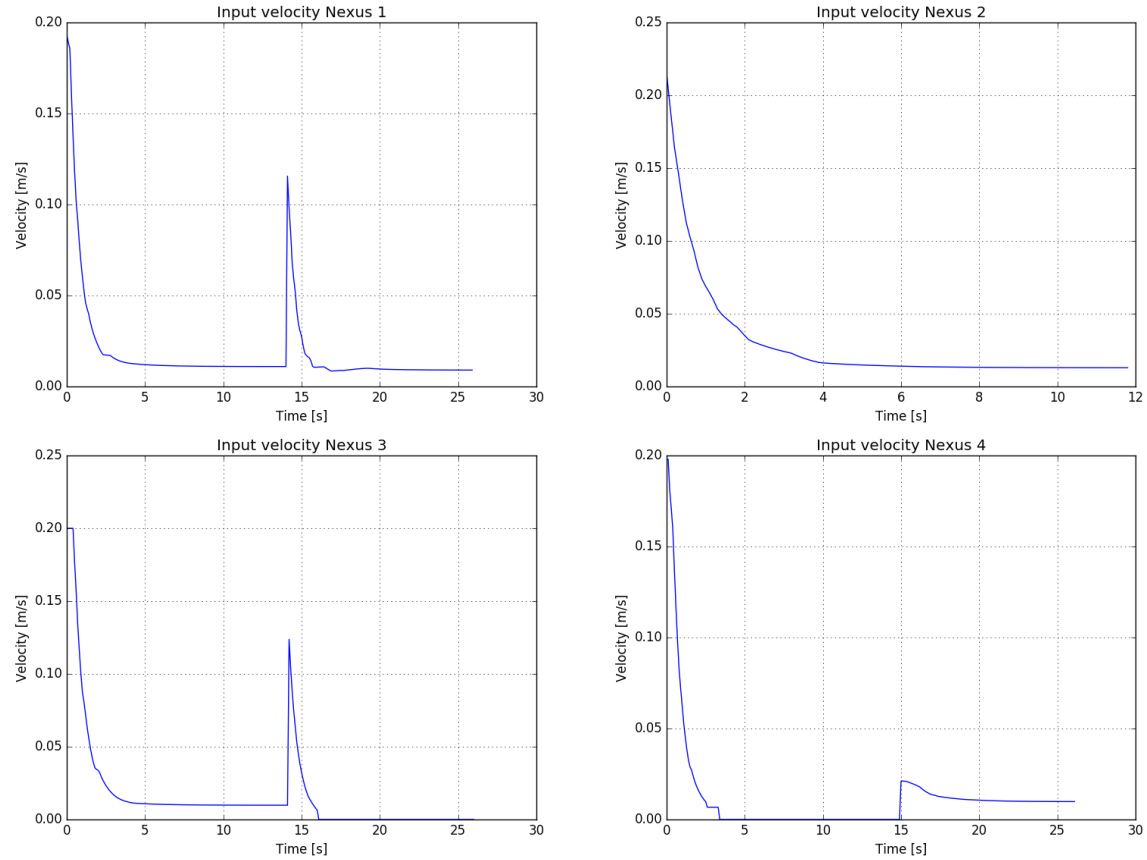


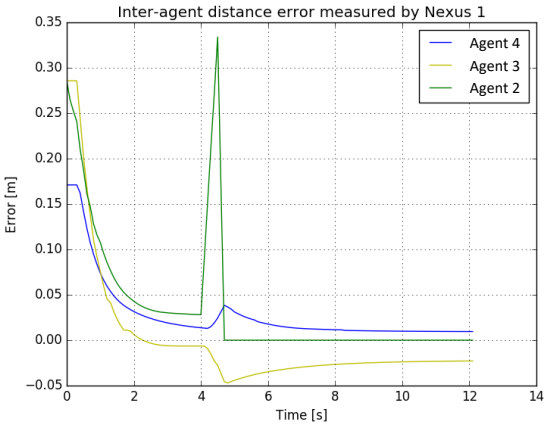
Figure 17: Insertion of Agent 2 (Velocities)

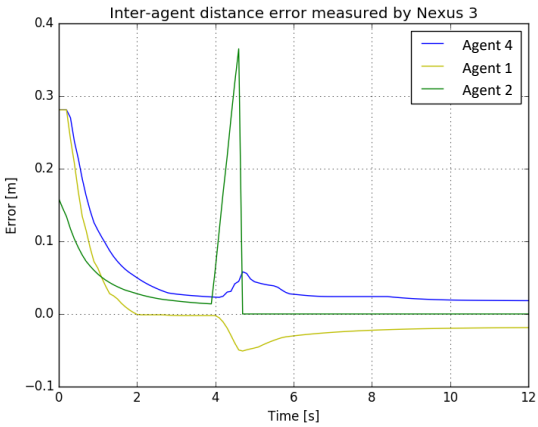
The following step is to check the performance of the formation with the extraction of Agent 2 when it has already achieved an equilibrium. The following steps outline the process utilized to test how the extraction of Agent 2 affected the formation:

- Spawn all agents.
- Start Data Processing Node and Controllers for all agents to achieve a four-agent formation.
- Once equilibrium is achieved an input is given to Agent 2 to go in direction opposite to the location of the formation's center, in order to exit it faster than agent's 1 and 3 could follow it.
- Let the new 3-agent formation achieve equilibrium.

The effect of this process can be observed in Figure 19, where the inter-agent distances are plotted over time. Starting with the plot of Agent 1, it can be seen how the three inter-agent distances of its neigbours first tend towards equilibrium close to 0. Then, 4 seconds in, a spike occurs for the inter-agent distance with Agent 2, which occurs due to Agent 2 trying to exit the formation by accelerating away from it. Once Agent 2 is out of range, and therefore not a part of the formation anymore, the inter-agent distance with Agent 2 becomes symbolically 0. As Agent 1 tried to initially follow Agent 2, it caused a small disturbance in the inter-agent distances with the other neighbours, however, it can be seen that afterwards the new triangular formation corrects itself. The plot for Agent 3 follows the same structure as the one for Agent 1. Finally, the plot for Agent 4 only displays a small disturbance in the inter-agent distances due to the displacements of Agents 1 and 3, which is quickly corrected. This behaviour shows that, apart of the new agent's neighbours, no other agents are majorly disturbed with the distributed addition of a new agent using the proposed algorithm. Furthermore, some improvements could be introduced to further remove the effects on the existing formation as suggested in Section 9.1.

Similarly to the process of adding an agent, the plot of Agent 2 does not compile properly when run under the specified circumstaces, therefore it was ommited for Figure 19 and Figure 20.





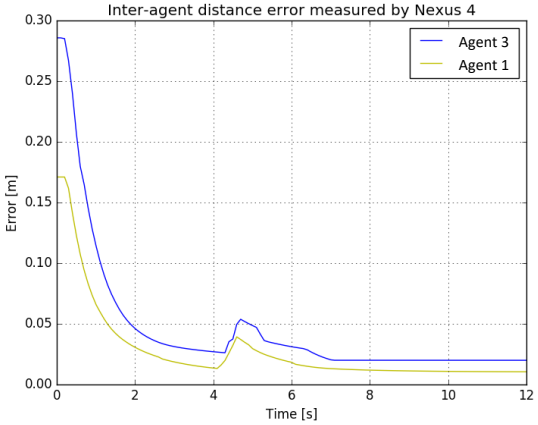
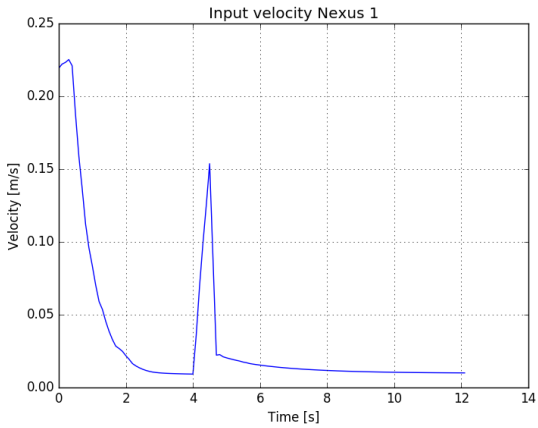
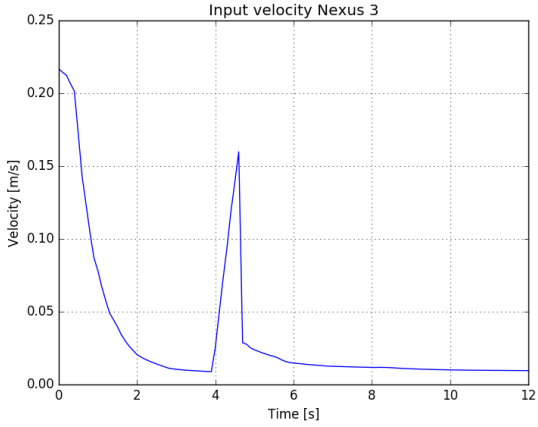


Figure 19: Extraction of Agent 2 (Inter-agent Distances)





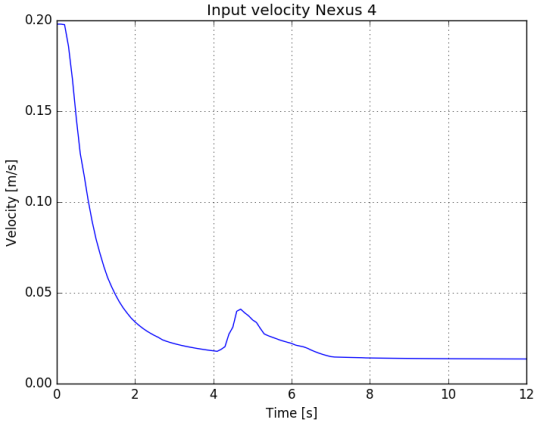


Figure 20: Extraction of Agent 2 (Velocities)

6.3.4 Estimator

This section will analyse the performance of the estimator control law and the algorithm developed in Section 5.2.3. For this simulation, Agent 4 was given a goal distance of 1 meter, compared to the rest of the formation wanting to achieve distances of 0.9 meters with their neighbours. This difference would therefore create a mismatch between Agent 4 and its neighbours 1 and 3 in the same type of four agent formation shown in Figure 16 with estimating pattern as shown in Figure 7 and calculated with Algorithm 8. With the application of the estimator it was expected that the formation would achieve stability without drifting, although with a distorted final shape.

Initial applications of the control law did not show the expected formation equilibrium as the formation tended to drift due to the mismatches. After some tests it was noticed that the reason why this took place was due to the gain c_2 not being of adequate magnitude. Through trial-and-error the value of 2.9 was found to result in a still formation. However, as soon as the mismatch of Agent 4 was changed a new gain was required, therefore, this estimator application suffers from a high dependency on the gain which is undesirable as in non-simulated environments readings in error would be unknown. To resolve this problem in future applications it would be recommended to apply de Marina's second suggested estimator approach as it is gain independent (de Marina Peinado, Cao, & Jayawardhana, 2016, pp. 38-42).

The performance of the previously described simulation can be seen in Figure 21 and Figure 22. From the latter it can be seen that the velocities tend towards 0 for all agents, indicating that the formation is still, not drifting, and that the estimator has managed to eliminate the effect of Agent 4's mismatch. However, the velocity plot of Agent 4 displays an anomaly every approximately 5 seconds. These spikes do not have an apparent reason to happen and can be ignored as it did not have an effect on the overall formation. Further research should be done to determine what is causing such jumps in the velocity of the agent with the mismatch. Lastly, Figure 21 shows that the inter-agent distances between agents are achieved properly, apart from a larger, although negligible, error brought by the mismatch of Agent 4 in the plots of agents 1 and 3.

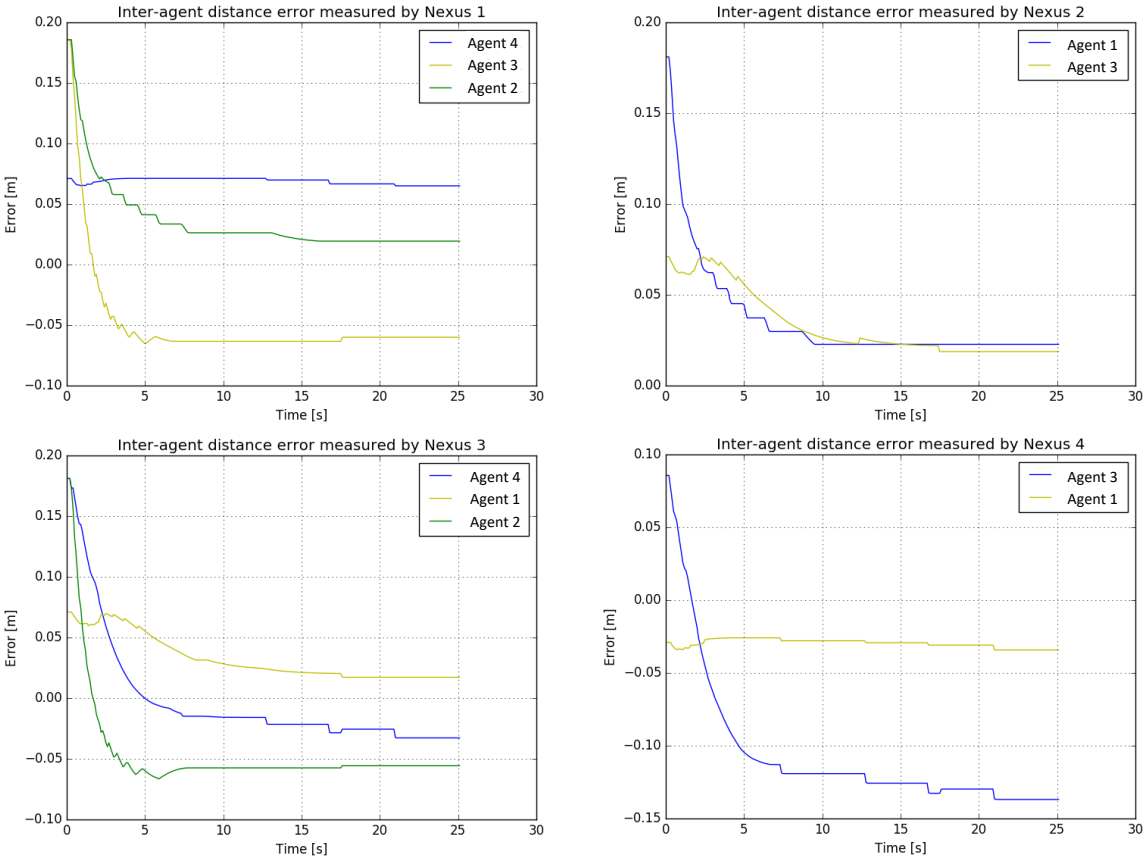


Figure 21: Inter-agent distance for a four-agent formation where agent 4 has a distance goal of 1 m and the rest of 0.9 m

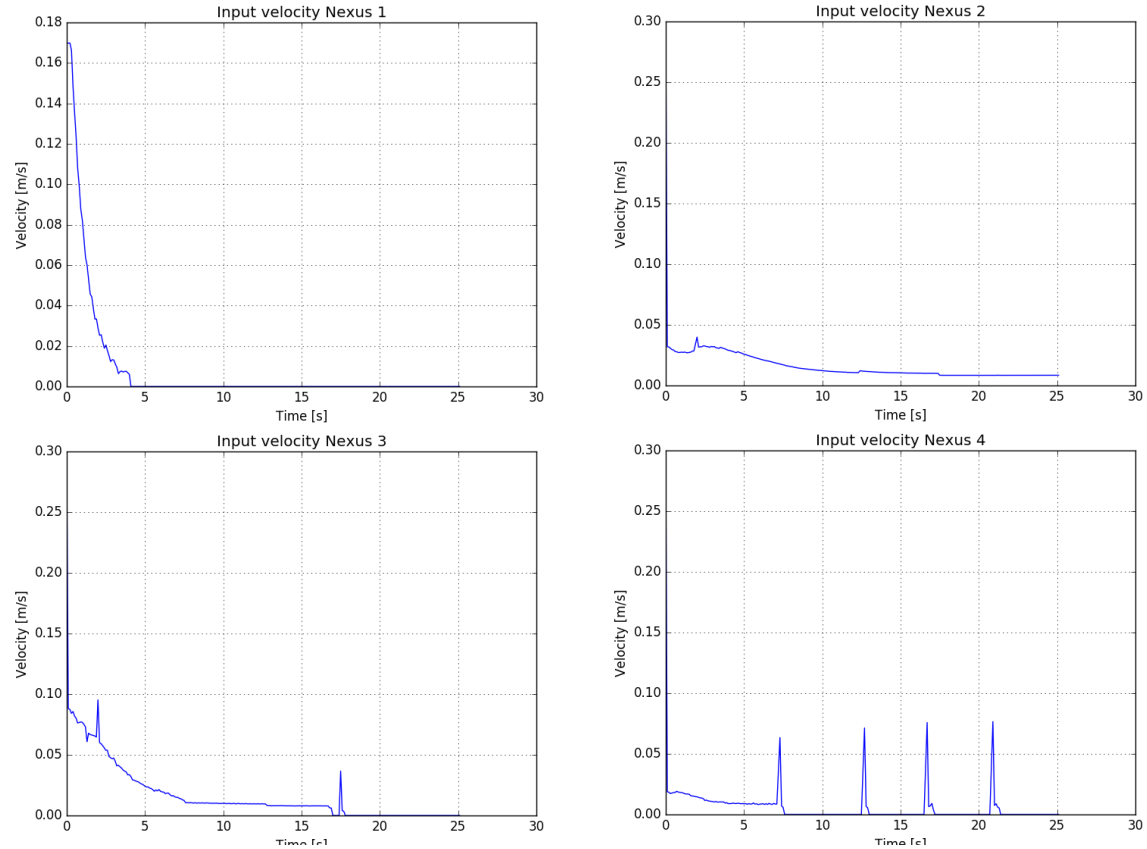


Figure 22 :Velocities for a four-agent formation where agent 4 has a distance goal of 1 m and the rest of 0.9 m

7 Experimental Setup

At the time of writing of this thesis, the DTPA laboratory's robots are only able to go into formation and move with it. However, adding the ability to add a new agent without disrupting the current formation while it is in operation would expand on the system's flexibility and open new possibilities for improvement, such as the addition of a human agent to the formation.

7.1 DTPA lab equipment

This section will introduce the available equipment for this research. These resources will be used in the application process.

7.1.1 Nexus Robots

Although the field of formation control also includes flying unmanned vehicles, this project will focus on the coordination of a set of land vehicles known as "Nexus 4WD Mecanum Wheel Mobile Arduino Robotics Car 10011". These robots have been assembled at the DTPA lab and include "100mm Aluminium Mecanum wheels, Faulhaber 12V motors with optical encoders, Arduino 328 Controller, Arduino IO Expansion, ultrasonic and IR sensors" (de Marina, Jayawardhana, & Cao, 2016). Following in Figure 23 is a picture of the fully equipped robot, including a robotic arm. It should be noted that the white cylinder on top is a 3D printed expansion for the RPLIDAR in order to improve the chances of the robots detecting each other at large distances.

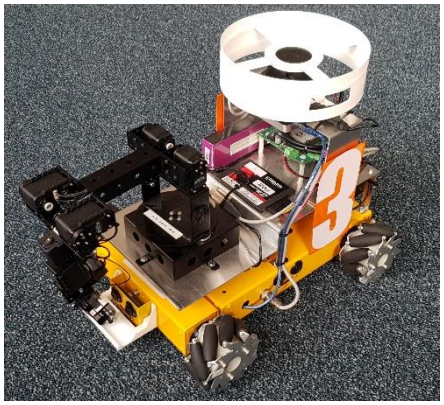


Figure 23: Nexus robot with LIDAR (Inside white cylinder) and a robotic arm mounted on top

One of the distinctive features of these robots, apart from all their sensors, are the omnidirectional wheels that allow it to translate on any direction, making it holonomic. Such a capability is useful when manoeuvring in tight spaces that do not allow extended movements (Doroftei & Spinu, 2007).

7.1.2 Available Sensors

The available sensors at the laboratory to scan the surrounding environment were the RP-lidar scanner, Kinect camera and IR sensors. These sensors could allow to distinguish between obstacles and possible human leaders, as well as, enable possible communication between the leader and the formation. This section will briefly expand on the functionality and operation of each sensor to give a clear understanding of what the current robots are capable of.

The RPlidar scanner installed on top of the robot is a 360-degree sensor that measures distance by emitting and receiving back a pulse laser light. These types of sensors are helpful in the detection of obstacles and mapping of the current environment. The main concern with the use of this sensor is the fact that there is no distinction between types of obstacles and would therefore handicap the detection of a nearby agent. In Figure 24 a schematic of the LIDAR's operation is shown. Basically, as it rotates around its axis it stores the distance where it finds a surface in which the LIDAR's light bounces. Given that the angular resolution is of one degree, meaning that a revolution would provide 360 data points, the recorded distances will be stored in a 1 by 360 matrix, where the assigned value will be the distance to the detected object, or 0 if nothing was found. Furthermore, the used RPlidar A1 used has a distance resolution of 0.2 meters. (Slamtec, 2018)

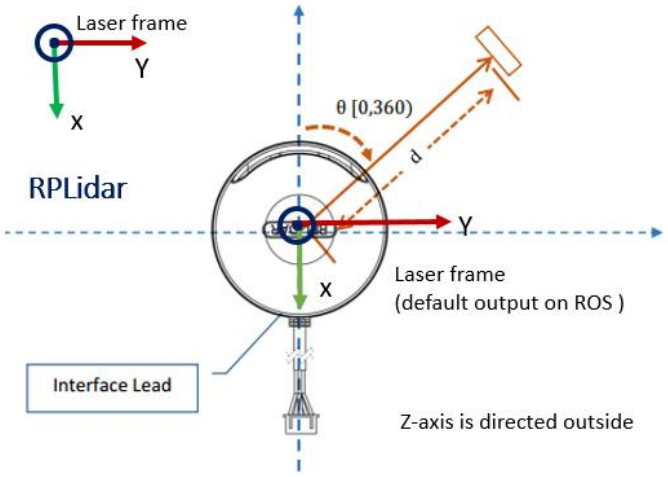


Figure 24: Schematic of the RPlidar's functioning (Θ - degree location of scan; d - distance to surface) (Robopeak, 2016)

The Kinect 3D camera is a unidirectional visual input capable of detecting depth due to its multiple integrated sensors. This camera can have multiple uses, from complex interaction with the user to object detection. The main limitation of using this camera is directional limitation as it is only facing in front of the robot and is not able to detect anything to its sides or back.

The IR sensors on the sides of the robot are able to detect heat signatures, which can become useful in low visibility situations or smoke-filled rooms where the previously mentioned sensors would fail to operate. A use of this sensor could be, but not limited to, distinction between human presence and any obstacle in proximity.

As previously covered, these experiments will use the available RPlidar A1 to detect the robot's neighbours within the formation as it allows to perceive other agents without the directional limitation of cameras. However, in further research the rest of the available sensors could be used, for example, to detect the presence of a human leader.

7.2 Results

This section will display the results of the application of the algorithm on the Nexus robots, to test its operation under real conditions.

7.2.1 Basic Formation of Four Agents

With the formation type as shown in Figure 25, this section shows the performance of the algorithms on the Nexus robots under laboratory conditions with no obstacles.

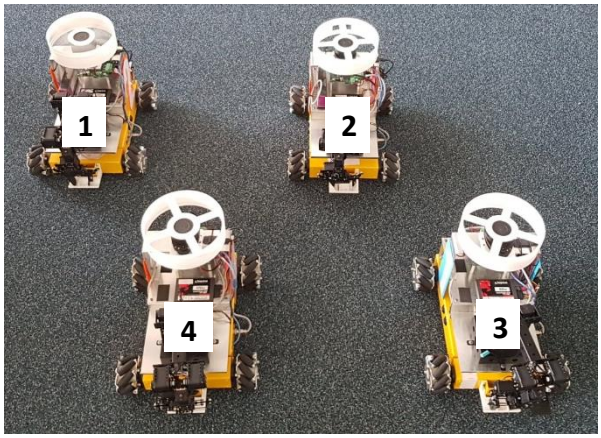


Figure 25: Experimental Setup order of Nexus Robots

From Figure 26 it is possible to see that the system does converge in terms of inter-agent distance, although not as smooth as the results from simulations. The main outliers can be observed in the plots of Agents 2 and 4. These occurrences can be considered to be random noise in the operation of the robots or a reading error from the LIDAR as they are not reflected on the errors of the agents' neighbours, therefore further measures could be taken in future research to avoid similar errors.

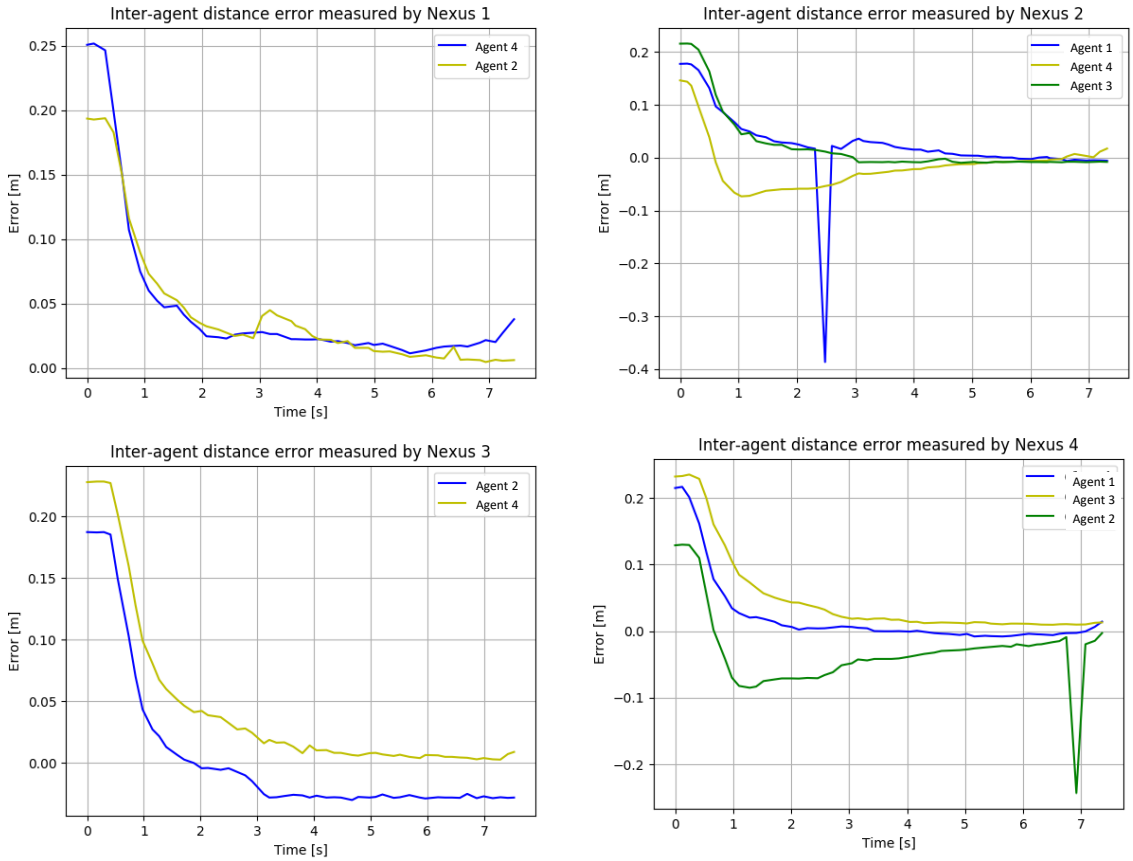


Figure 26: Inter-agent distances for each robot in a four-agent formation

7.2.2 Addition and Extraction of an Agent

Now that the basic operation of the algorithm is shown in the previous section the following step will be to show the operation of extraction and addition of agents into an already converged formation of the same shapes as in Section 6.3.3, with the same extraction and addition procedures.

Just as in Section 6.3.3, the plot for Agent 2 is omitted as the code would not function properly when disconnecting from the formation, due to the plotting tools being located in the controller which needs to detect neighbours in order to operate.

Figure 28 and Figure 27 exhibit a similar pattern as observed in the simulations, thus they can be interpreted in the same way. However, just as in the previous sub-section, the inter-agent distance errors display sudden jumps probably due to the resolution of the lasers not being able to detect minimal changes in distances, since the distance resolution is of 0.2 meters.

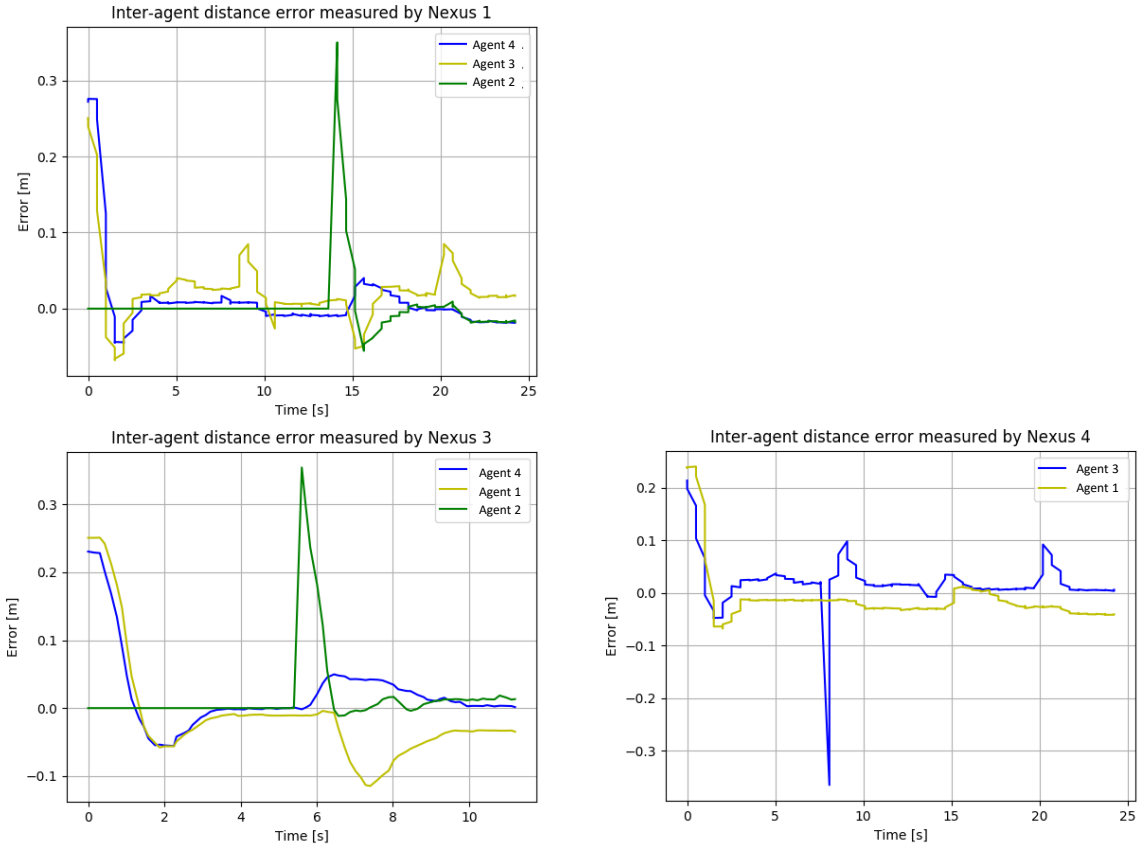


Figure 28: Inter-agent distances over time for Nexus 1, 3 and 4 with the addition of Nexus 2

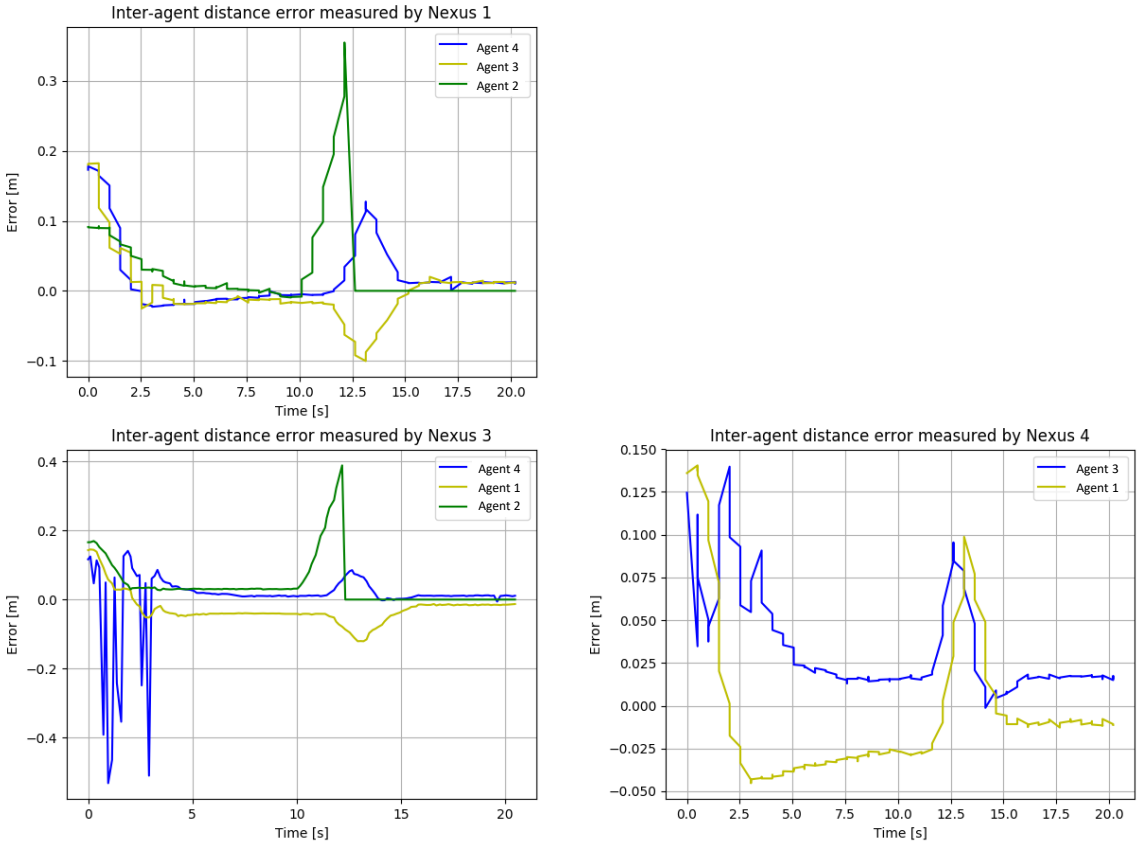


Figure 27: Inter-agent distances over time for Nexus 1, 3 and 4 with the extraction of Nexus 2

8 Contributions

8.1 Physics Based Simulation

The LIDAR based multi-agent formation Gazebo simulation was created to simplify the testing process of code implementation into the Nexus Agents in the DTPA Lab. In this thesis the simulation was used to test extraction and addition of robots, however, any future applications of the RPlidar laser and the Nexus robots could be used for it. Therefore, this platform also allows for the validation of code before its implementation, which can save time and resources, and make it easier to identify flaws in the system.

To simplify the simulation process, launch files were also created so that, through ROS, only one file needs to be run to obtain all agents and corresponding LIDAR sensors within Gazebo. Furthermore, the simulation is highly customizable, including the LIDAR's specifications, number of agents, and agent's initial positions.

8.2 Dynamic Distibuted Formation Control

Exploring existing literature of multi-agent systems there seems to be lack of analysis of distributedly controlled and dynamically changing formation structures, as most focus seems to be on a fixed structure and time-invariant formations. The concept of formation shape changes for obstacle avoidance has been explored, but in such cases, formations tend to be predefined. If the formation is able to dynamically change then formation assembly in unknown environments is made possible, dealing with dynamically changing environments would become easier, and exchanging or combining agents between different formations is made possible.

This thesis suggests an algorithmic approach to creating a dynamically changing formation and then the approach is experimentally validated to show its operation. An important detail of this approach is that no information is shared between agents.

8.3 Insight into The Application of Estimators

Through simulations, the dependency of the current implementation of estimators from de Marina's thesis was found to be highly dependent on the gain c_2. Therefore, for a better performance of estimators it is suggested to implement the second order estimating system suggested by de Marina.

9 Discussion

The main research question set to be answered for this thesis was "How can a formation of multiple agents be distributedly controlled and reconfigured in order to react to an additional agent?". The question was approached through the suggested algorithms in Section 5.2. The final answer was highly influenced by the control laws used and the final goal of application on the Nexus robots, hence the holonomic nature of the agents implemented in simulations. As shown in Sections 6 and 7, the suggested approach results in the desired dynamic behaviour of the formation. However, some improvements could be done to the used code as will be suggested in Section 9.1.

To answer the main research question different sub questions were set. Starting with “What actions should the robots take to ensure the formation is maintained?”, the approached methodology is that of a triangular lattice formation to simplify the neighbouring agent selection. Then, all agents were started with the same orientation to avoid problems introduced by having agents with different initial scanning reference points, and finally, all agents should be at a close enough distance, defined by a maximum scanning range, in order to enter the formation. If these conditions are followed, all agents are equipped with a LIDAR sensor and use de Marina’s gradient based control laws, then all agents would be able to create and maintain a formation used the suggested approach.

The following sub question is “How can the formation performance be evaluated?”. The used method to evaluate the performance of the formation was to ensure the error of the inter-agent distances between neighbours tended towards zero. Then, to evaluate the operation of the suggested algorithms on the formation, a physics-based simulation was created where ideal conditions can be applied to quickly test their performance, and finally, tests were done in real robots to determine possible flaws in real life systems.

The third sub question was “How to distinguish neighbouring agents?”, which would depend on the sensor chosen to obtain information from the environment. For this purpose, a LIDAR was chosen as it provides basic and easy to process information about the agent’s surroundings. Once the data was obtained, the task of distinguishing neighbours is performed in the Data Processing Node, where agents are separated based on the angle at which the LIDAR collided with an object. If two data points happen to not be consecutive, it indicates that the new data point belongs to a new entity, and therefore, a new agent distinguished. Problems with this approach are that LIDAR sensors tend to be expensive compared to the cost of cameras and distinguishing between obstacles and agents is a complex task given only the point cloud provided by the LIDAR. However, through LIDAR the process of obtaining 3D data of the environment is possible, while basic cameras only provide a 2D image, making the process of determining distances to objects more complicated.

The fourth sub question was “How should new agents adhere to the existing formation?”. Since distributed control was being used and the goal of adding new robots into the formation distributedly is not to increase exponentially the levels of computational power required, the suggested approach is that of the Henneberg insertion algorithm. Through this method, the minimum amount of required edges is created, and the formation retains the required condition of being minimally rigid. In the applied process of agents joining the formation, the currently implemented code also forces agents within the formation to approach the new incoming agent. This effect may not be desirable as the addition and extraction of new agents would cause unpredictable motion on the initial formation. However, extra features could be added to the code, as suggested in Section 9.1, to reduce or eliminate this effect.

The fifth sub-question was “How should the formation react to the addition of a new agent?”. As covered in the previous sub question, in the addition or extraction process the goal of distributed control is to avoid having a significant impact on the overall formation and reducing computational load. Currently, the implemented methodology causes the formation to approach the incoming agent, which could be eliminated in further research.

The sixth sub question was “How scalable is the reconfiguration method in dealing with large numbers of new agents?”. As shown in simulations using the suggested algorithm, the formation is able to run from 3 to 7 agents, and showing that the addition of agents is possible, then through induction it can be determined that a formation using the designed algorithm, and following its current basic conditions, can have an unlimited number of agents. Due to the distributed control system, the number of agents does not affect whether or not a new agent can join the formation, making the reconfiguration method highly scalable. Furthermore, although not covered in this thesis, it could be possible for multiple agents to join the formation at separate locations simultaneously, just like in a flock of birds where every agent is completely independent.

The last sub question was “How does it perform in an experimental setup?”. The results to this question can be seen in Section 7, and it can be said that the results are satisfactory as a formation is achieved and the addition and extraction of agents is possible. Although fine tuning of the experimental results was not possible due to a time restriction, it could be done in future research by focusing on the application of the algorithm, as this thesis focused on its development.

9.1 Future Research

For all the experiments and simulations shown in this thesis, the problem introduced by the initial reference point is circumvented by avoiding locating agents in that critical point. A more sustainable approach, that would allow robots to have different orientations, would be to introduce an algorithm capable of distinguishing if a robot located at the reference point is only one entity and not two.

To improve the agent addition process different stages could be added to the data processing node. This approach would allow the formation to enter a state of “re-configuration” only if new agents need to be added. These stages could reduce the chances of detecting an obstacle as an agent and would improve the robustness of the formation. This problem could also be solved by adding extra sensors, such as a 360-degree camera. This extra sensory input would allow the distinction of obstacles and agents, and by extension be able to distinguish when an agent needs to be added if there is one in range. However, adding sensors would increase the costs of agents in experimental setups, as well as increasing computational load on each agent.

In this thesis the application of estimators is not shown as experimental setup results were inconclusive and not able to maintain a proper formation. The suggestion for this effect is explained in the gain dependency shown in simulations. Therefore, to eliminate this effect, and possibly improve the experimental application, it would be recommended to implement de Marina’s second suggested estimator as it is gain independent.

To further analyse the performance of the introduced algorithm it could be possible to determine the settling time of the formation after addition and extraction of agents or analyse the effect on the centre of mass of the formation when new agents are added. However, before that is done, it would be recommended to revise the used code and possibly add some

other features, as the code used for this thesis only focused in basic operation under multiple conditions rather than tuning its operation to improve its efficiency.

Finally, to improve on the used code, various more features could be added. The used code does not provide any possibility for the motion of the formation and agents are forced to remain stationary within the formation, therefore, a distributed motion methodology would need to be developed. The introduction of a "Confidence array" as suggested in Ballard's paper (Ballard, 2008) could be done for each agent to have knowledge on the formation's overall shape and expanding the formations capabilities, although it would introduce communication between agents. Finally, now that addition of agents is possible, a human could be introduced as a leader agent, although extra sensors, such as a Kinect camera, would be required to achieve proper execution.

10 Conclusion

The purpose of this thesis was to design, test and experimentally validate a dynamically changing formation structure capable of automatically adding and extracting agents during operation. To facilitate the testing of the proposed solution a Gazebo physics-based simulation of multiple agents was created. This simulation allowed to test the proposed algorithms under ideal conditions to check their operation and avoid any complications introduced by testing them in the real world. Finally, the algorithms were tested on Nexus robots in a laboratory setting to check their performance in the real world.

The proposed algorithm, using de Marina's control laws, allowed the formation to experimentally add and extract agents automatically during operations and maintain equilibrium after re-configuration. The experimental results obtained from the application on the Nexus robots returned acceptable data, although tuning of the used code should be done to reduce noise and dampen any sudden movements. The application of estimators to reduce the effects of mismatches was inconclusive experimentally, although a high dependency on the gain was found in simulations, which leads to the recommendation of application of another estimating methodology.

In conclusion, the proposed algorithmic approach allows for automatic distributed re-configuration of agents in a triangular lattice formation for an unlimited number of agents using LIDAR sensing technology. In further applications of the designed algorithm more features could be added, such as the ability to distinguish between obstacles and potential agents or stages to the addition process, to improve its performance under experimental conditions.

11 References

Ballard, L. (2008). Experiments in Distributed Multi-Robot Coordination. *All Graduate Theses and Dissertations*, 169.

Belabbas, A., Mou, S., Morse, A., & Anderson, B. (2012). Robustness issues with undirected formations. *Proc. of the 51st IEEE Conference on Decision And Control* (pp. 1445–1450). Maui, HI, USA: IEEE.

de Marina Peinado, H., Cao, M., & Jayawardhana, B. (2016). *Distributed formation control for autonomous robots.* Groningen: University of Groningen.

de Marina, H. G., Jayawardhana, B., & Cao, M. (2016). Distributed Rotational and Translational Maneuvering of Rigid Formations and Their Applications. *IEEE Transactions on Robotics, 32*(3), 684-697. doi:10.1109/TRO.2016.2559511

Doroftei, V. G., & Spinu, V. (2007). Omnidirectional Mobile Robot-Design and Implementation. In M. K. Habib, *Bioinspiration and Robotics: Walking and Climbing Robots* (pp. 511-528). Wien: I-Tech Education and Publishing.

Lebrecht, H. (1911). *Die graphische Statik der starren Systeme.* B. G. Teubner.

Navarro, I., Pugh, J., Martinoli, A., & Matia, F. (2009). A Distributed Scalable Approach to Formation Control in Multi-Robot Systems. In H. Asama, H. Kurokawa, J. Ota, & K. Sekiyama, *Distributed Autonomous Robotic Systems 8* (pp. 203-214). Berlin, Heidelberg: Springer.

Robopeak. (2016, August 8). *How to use RPlidar*. Retrieved from Github: https://github.com/robopeak/rplidar_ros/wiki/How-to-use-rplidar

Siemonsma, J. (2017). *Distributed translational and rotational control of rigid formations and its applications in industries.* Groningen: University of Groningen.

Slamtec. (2018). *RPlidar A1*. Retrieved from www.slamtec.com: http://www.slamtec.com/en/lidar/a1

Sun, Z., Anderson, B., Mou, S., & Morse, A. (2013). Non-robustness of gradient control for 3-D undirected formations with distance mismatch. *Proc. of the 2013 IEEE Australian Control Conference* (pp. 369–374). Fremantle, WA, Australia : IEEE.

www.ROS.org. (2017, 03 31). Retrieved from About ROS: http://www.ros.org/about-ros/

12 Appendix

12.1 Data Processing Node for N agents

```
#!/usr/bin/env python

from __future__ import division

import sys
from math import *
import rospy
import numpy as np
from rospy_tutorials.msg import Floats
from std_msgs.msg import Int32
from rospy.numpy_msg import numpy_msg
from sensor_msgs.msg import LaserScan


class determine_z_values:
    ''''' Determines the inter-agent distance values and publishes it to z_values topic '''

    def __init__(self):
        ''''' Initiate self and subscribe to /scan topic '''

        # which nexus?
        self.name = 'n_'+str(int(sys.argv[1]))
        # Desired distance - used for sending if no z is found or if the dataprocessingnode is shutdown:
        # robot not influenced if one z not found
        self.d = np.float32(0.8)

        # set min and max values for filtering ranges in meter during initiation
        self.min_range = 0.25
        self.max_range = 1.2

        # prepare shutdown
        self.running = True
        rospy.on_shutdown(self.shutdown)

        # prepare publisher
        self.pub = rospy.Publisher(self.name +'/z_values', numpy_msg(Floats), queue_size=1) #Publish the distances and angles to agents in range
        self.PUB = rospy.Publisher(self.name +'/agents', Int32 , queue_size=1) #Publish the amount of surrounding agents
        # subscribe to /scan topic with calculate_z as callback
        rospy.Subscriber('/'+self.name+'hokuyo_points', LaserScan, self.calculate_z)

        np.set_printoptions(precision=2)

    def calculate_z(self, msg):
        ''''' Calculate the z_values from the scan data '''
        # Check if not shutdown

        self.ranges= np.asarray(msg.ranges)

        if self.running:
            # Save the angles (hits) of the robots in separate arrays
            z_a = np.where((self.ranges >= self.min_range) & (self.ranges <= self.max_range))[0] # The zero at the end is to access the first value of the tuple created by "np.where", so z_a is just an array
            n = 1

            for i in range(len(z_a)-1):  # Calculates the number of robots
                if (z_a[i] - z_a[i + 1] >= -10):

                    continue
```

```
58.
59.                 elif (-10 <= z_a[i] - z_a[i-1] <= 10):
60.
61.                     n = n + 1
62.
63.             R=[]
64.             r=np.array([])
65.             Zval=np.array([])
66.             P = 0
67.
68.             # Compares difference between angles in z_a to decide if it's a new robot, and if it i
        s it creates and array r, which is then added to the list R
69.             for i in range(P,len(z_a)-1):
70.
71.                 if (z_a[i] - z_a[i + 1] >= -10) and i!=(len((z_a))-2):
72.
73.                     r=np.append(r,z_a[i])
74.                 elif (-10 <= z_a[i] - z_a[i - 1] <= 10):
75.                     r=np.append(r,z_a[i])
76.                     R.append(r)
77.                     P = 1+i
78.                     r=np.array([])
79.
80.                 elif (z_a[i] - z_a[i + 1] >= -10) and i==(len((z_a))-2):
81.                     r=np.append(r,z_a[i])
82.                     R.append(r)
83.
84.             for i in range(n):  #transform list R to array of integers
85.                 R[i]=R[i].astype(int)
86.
87.
88.             self.PUB.publish(n)
89.
90.             self.z_aX=np.zeros([len(R)])
91.             self.zn_X=np.zeros([len(R)])
92.             self.z_a_min=np.zeros([len(R)])
93.             self.z_a_max=np.zeros([len(R)])
94.             self.zn_min=np.zeros([len(R)])
95.             self.zn_max=np.zeros([len(R)])
96.
97.             for j in range(0,n):
98.
99.                 if R[j] != np.array([]):
100.
101.                     self.z_aX[j] = int(np.round((R[j]).mean()))
102.
103.                     self.zn_X[j] = np.float32(np.min(self.ranges[(R[j][0:(len(R[j]))])]))
104.
105.                     self.z_a_min[j] = np.min(np.int_(R[j][0:len(R[j])]))
106.
107.                     self.z_a_max[j] = np.max(np.int_(R[j][0:len(R[j])]))
108.
109.                     self.zn_min[j] = np.min(np.float32(self.ranges[np.int_(R[j][0:len(R[j])])]))
110.
111.                     self.zn_max[j] = np.max(np.float32(self.ranges[np.int_(R[j][0:len(R[j])])]))
112.
113.                     self.z_aX=self.z_aX.astype(int)  #transform z_aX to an array of integers
114.
115.             Zval=[]
116.             self.zx=np.zeros([len(R)])
117.             self.zy=np.zeros([len(R)])
118.             self.z_values=[]
119.
120.             for i in range(0,n): #FOR SOME REASON i thought this should go to n-1)
121.
```

```
                    self.zx[i] = np.float32(np.cos((self.z_aX[i]-
            np.int_(180))*2*np.pi/360)*self.zn_X[i])

                    self.zy[i] = np.float32(np.sin((self.z_aX[i]-
            np.int_(180))*2*np.pi/360)*self.zn_X[i])

                    Zval.append([self.zn_X[i], self.zx[i], self.zy[i]])

                for i in range(0,n):  #Loop to print the x and y distances of detected agents

                    print 'zx_[',i,'] = ', self.zx[i]
                    print 'zy_[',i,'] = ', self.zy[i]
                    print '--'
                print '--------------------'

                for i in range(0,n): #This loop puts all the values into the z_values matrix

                    self.z_values= np.concatenate(Zval[:][:])

                self.z_values=np.asarray(self.z_values,dtype=np.float32)  #this line ensures that all
            numbers are type np_float (without it the publisher doesn't publish the proper values)

                #Publish z_values
                self.pub.publish(self.z_values)


        def shutdown(self):
            # Setting z = d in order to stop the robots when shutting down the dataprocessingnode_2 no
            de
            rospy.loginfo("Stopping dataprocessingnode_"+str(int(sys.argv[1]))+"...")
            self.running = False
            self.pub.publish(self.z_values)
            print ('Shutting Down')
        rospy.sleep(1)

if __name__ == '__main__':
    rospy.init_node('dataprocessingnode_'+str(int(sys.argv[1])), anonymous=False)
    determine_z_values()
    rospy.spin()
```

12.2 Controller Node for N Agents

```
#!/usr/bin/env python
import sys
import rospy
import matplotlib.pyplot as pl
import numpy as np
from rospy_tutorials.msg import Floats
from std_msgs.msg import Int32
from rospy.numpy_msg import numpy_msg
from geometry_msgs.msg import Twist

class controller:
    ''''' The controller uses the interagent distances to determine the desired velocity of the Ne
    xus '''

    '''''  NOTE: this script requires the dataprocessing node to be running first, as well as an i
    nput
        argument, an example of how to properly run this code in the terminal is as following:

        rosrun lasmulticontrol3 dataprocessingnode_N.py "1"

        where "1" is the value assigned to the robot
        '''
```

```
21.
22.      def __init__(self):
23.          ''''' Initiate self and subscribe to /z_values topic '''
24.          self.name='n_'+str(int(sys.argv[1]))
25.          # controller variables
26.          self.running = np.float32(1)
27.          self.d = np.float32(0.8)
28.          self.dd =  np.float32(0.8)
29.          self.c = np.float32(0.5)
30.          self.U_old = np.array([0, 0])
31.          self.U_oldd = np.array([0, 0])
32.
33.
34.          # prepare Log arrays
35.          self.E1_log = np.array([])
36.          self.E2_log = np.array([])
37.          self.E3_log = np.array([])
38.          self.E4_log = np.array([])
39.          self.E5_log = np.array([])
40.          self.E6_log = np.array([])
41.          self.Un = np.float32([])
42.          self.U_log = np.array([])
43.          self.time = np.float64([])
44.          self.time_log = np.array([])
45.          self.now = np.float64([rospy.get_time()])
46.          self.begin = np.float64([rospy.get_time()])
47.          self.k = 0
48.
49.          # prepare shutdown
50.          rospy.on_shutdown(self.shutdown)
51.
52.          # prepare publisher
53.          self.pub = rospy.Publisher(self.name+'/cmd_vel', Twist, queue_size=1)
54.          self.velocity = Twist()
55.
56.          # subscribe to z_values topic
57.          rospy.Subscriber(self.name+'/z_values', numpy_msg(Floats), self.controller)
58.          rospy.Subscriber(self.name+'/agents', Int32, self.n)
59.          # subscribe to controller_variables
60.          rospy.Subscriber('/controller_variables', numpy_msg(Floats), self.update_controller_variab
     les)
61.
62.      def n(self, data):   #This is used to extract the value of n (i.e. the number of robots the ag
     ent detected, published from the from the dataprocessor node)
63.          if self.running < 10:
64.              self.n=data.data
65.          elif 10 < self.running < 1000:
66.              self.shutdown()
67.
68.      def update_controller_variables(self, data):
69.          ''''' Update controller variables '''
70.          if self.running < 10:
71.              # Assign data
72.              self.controller_variables = data.data
73.
74.              # Safe variables
75.              self.running = np.float32(self.controller_variables[0])
76.              self.d = np.float32(self.controller_variables[1])
77.              self.dd = np.float32(self.controller_variables[2])
78.              self.c = np.float32(self.controller_variables[3])
79.
80.
81.      def controller(self, data):
82.          ''''' Calculate U based on z_values and save error velocity in log arrays '''
83.          if self.running < 10:
84.              # Input for controller
```

```
            z_values= data.data

            Bx=np.array([])
            By=np.array([])
            D = np.array([])
            E = np.array([])
            Ed=[]

            for i in range(self.n):
                Bx=np.append(Bx, z_values[1+3*i])
                By=np.append(By, z_values[2+3*i])
                D=np.append(D, z_values[3*i]**(-1))
                E=np.append(E,z_values[3*i]-self.d)
                Ed.append([E[i]])

            # Formation shape control
            BbDz=np.append([Bx],[By], axis=0)
            Dzt=np.diag(D)
            Ed=np.asarray(Ed)

            # Control law
            U = self.c*BbDz.dot(Dzt).dot(Ed)
            print "U = ", -U

            # Saturation to reduce agents shaking
            v_max = 0.2
            v_min = 0.02
            for i in range(len(U)):
                if U[i] > v_max:
                    U[i] = v_max
                elif U[i] < -v_max:
                    U[i] = -v_max
                elif -v_min < U[i]+self.U_old[i]+self.U_oldd[i] < v_min : # preventing shaking
                    U[i] = 0

            # Set old U values in order to prevent shaking
            self.U_oldd = self.U_old
            self.U_old = U
            print self.n

            # Append 0 to error if no robot is detected to be able to plot later
            if self.n > 3:
            self.E3_log = np.append(self.E3_log, 0)
            if self.n > 4:
            self.E4_log = np.append(self.E4_log, 0)
            if self.n > 5:
            self.E5_log = np.append(self.E5_log, 0)
            if self.n > 6:
            self.E6_log = np.append(self.E6_log, 0)


            # Append error and velocity in Log array
            if self.n < 3 or self.n > 0:
                self.E1_log = np.append(self.E1_log, Ed[0])
                self.E2_log = np.append(self.E2_log, Ed[1])
            if self.n > 2:
                self.E3_log = np.append(self.E3_log, Ed[2])
            if self.n > 3:
                self.E4_log = np.append(self.E4_log, Ed[3])
            if self.n > 4:
                self.E5_log = np.append(self.E5_log, Ed[4])
            if self.n > 5:
                self.E6_log = np.append(self.E6_log, Ed[5])

            self.Un = np.float32([np.sqrt(np.square(U[0])+np.square(U[1]))])
            self.U_log = np.append(self.U_log, self.Un)
```

```
151.
152.             # Save current time in time log array
153.             if self.k < 1:
154.                 self.begin = np.float64([rospy.get_time()])
155.                 self.k = 10
156.             self.now = np.float64([rospy.get_time()])
157.             self.time = np.float64([self.now-self.begin])
158.             self.time_log = np.append(self.time_log, self.time)
159.
160.             # publish
161.             self.publish_control_inputs(U[0], U[1])
162.             A=self.n
163.             elif 10 < self.running < 1000:
164.                 self.shutdown()
165.
166.     def publish_control_inputs(self,x,y):
167.         ''''' Publish the control inputs to command velocities'''
168.
169.         self.velocity.linear.x = x
170.         self.velocity.linear.y = y
171.
172.         self.pub.publish(self.velocity)
173.
174.     def shutdown(self):
175.         ''''' Stop the robot when shutting down the controller node '''
176.         rospy.loginfo("Stopping Nexus_"+str(int(sys.argv[1]))+"...")
177.         self.running = np.float32(10000)
178.         self.velocity = Twist()
179.         self.pub.publish(self.velocity)
180.
181.         rospy.sleep(1)
182.         pl.close("all")
183.         pl.figure(0)
184.         pl.title("Inter-agent distance error measured by Nexus "+str(int(sys.argv[1])))
185.         pl.plot(self.time_log, self.E1_log, label="e1_nx"+str(int(sys.argv[1])), color='b')
186.         pl.plot(self.time_log, self.E2_log, label="e2_nx"+str(int(sys.argv[1])), color='y')
187.         if self.n > 2:
188.             pl.plot(self.time_log, self.E3_log, label="e3_nx"+str(int(sys.argv[1])), color='g')
189.         if self.n > 3:
190.             pl.plot(self.time_log, self.E4_log, label="e4_nx"+str(int(sys.argv[1])), color='r')
191.         if self.n > 4:
192.             pl.plot(self.time_log, self.E5_log, label="e5_nx"+str(int(sys.argv[1])), color='c')
193.         if self.n > 5:
194.             pl.plot(self.time_log, self.E6_log, label="e6_nx"+str(int(sys.argv[1])), color='m')
195.         pl.xlabel("Time [s]")
196.         pl.ylabel("Error [m]")
197.         pl.grid()
198.         pl.legend()
199.         pl.savefig("/home/mariano/Desktop/Plots/Nexus_Distance_"+str(int(sys.argv[1]))+".png")
200.
201.         pl.figure(1)
202.         pl.title("Input velocity Nexus "+str(int(sys.argv[1])))
203.         pl.plot(self.time_log, self.U_log, label="pdot_nx"+str(int(sys.argv[1])), color='b')
204.         pl.xlabel("Time [s]")
205.         pl.ylabel("Velocity [m/s]")
206.         pl.grid()
207.         pl.savefig("/home/mariano/Desktop/Plots/Nexus_Velocity_"+str(int(sys.argv[1]))+".png")
208.         pl.pause(0)
209.
210. if __name__ == '__main__':
211.     try:
212.         rospy.init_node('controller_'+str(int(sys.argv[1])), anonymous=False)
213.         controller()
214.         rospy.spin()
215.     except:
216.         rospy.loginfo("Controller node_"+str(int(sys.argv[1]))+" terminated.")
```

12.3 Controller Node with Estimator for N Agents

```
#!/usr/bin/env python
import sys #This import is used to allow for inputs into the function (hence the use of "sys.argv[
]")
import rospy
import matplotlib.pyplot as pl
import numpy as np
from rospy_tutorials.msg import Floats
from std_msgs.msg import Int32
from rospy.numpy_msg import numpy_msg
from geometry_msgs.msg import Twist

class controller:
    ''''' The controller uses the interagent distances to determine the desired velocity of the Ne
xus '''

    def __init__(self):
        ''''' Initiate self and subscribe to /z_values topic '''
        print "hello"
        self.name='n_'+str(int(sys.argv[1]))

        # controller variables
        self.running = np.float32(1)
        self.d = np.float32(0.9)
        if int(sys.argv[1])==4 : #Introduce mismatch to Agent 4
            self.d = np.float32(1)

        self.dd = np.float32(np.sqrt(np.square(self.d)+np.square(self.d)))
        self.c = np.float32(0.5)
        self.U_old = np.array([0, 0])
        self.U_oldd = np.array([0, 0])

        # prepare Log arrays
        self.E1_log = np.array([])
        self.E2_log = np.array([])
        self.E3_log = np.array([])
        self.E4_log = np.array([])
        self.E5_log = np.array([])
        self.E6_log = np.array([])
        self.Un = np.float32([])
        self.U_log = np.array([])
        self.time = np.float64([])
        self.time_log = np.array([])
        self.now = np.float64([rospy.get_time()])
        self.old = np.float64([rospy.get_time()])
        self.begin = np.float64([rospy.get_time()])
        self.k = 0

        # prepare shutdown
        rospy.on_shutdown(self.shutdown)

        # prepare publisher
        self.pub = rospy.Publisher(self.name+'/cmd_vel', Twist, queue_size=1)
        self.velocity = Twist()

        # subscribe to z_values topic
        rospy.Subscriber(self.name+'/z_values', numpy_msg(Floats), self.controller)
        rospy.Subscriber(self.name+'/agents', Int32, self.n)

        # subscribe to controller_variables
        rospy.Subscriber('/controller_variables', numpy_msg(Floats), self.update_controller_variab
les)

    def n(self, data):  #This is used to extract the value of n (i.e. the number of robots the age
nt detected, published from the from the dataprocessor node)
```

```
61.             if self.running < 10:
62.                 self.n=data.data
63.
64.             if self.n==2:
65.                 self.mu_hat = 0.0
66.             elif self.n>2:
67.                 self.mu_hat = np.array([[0], [0]])
68.
69.             elif 10 < self.running < 1000:
70.                 self.shutdown()
71.
72.         def update_controller_variables(self, data):
73.             ''''' Update controller variables '''
74.             if self.running < 10:
75.                 # Assign data
76.                 self.controller_variables = data.data
77.
78.                 # Safe variables
79.                 self.running = np.float32(self.controller_variables[0])
80.                 self.d = np.float32(self.controller_variables[1])
81.                 self.dd = np.float32(self.controller_variables[2])
82.                 self.c = np.float32(self.controller_variables[3])
83.
84.
85.         def controller(self, data):
86.             ''''' Calculate U based on z_values and save error velocity in log arrays '''
87.             if self.running < 10:
88.                 # Input for controller
89.                 z_values= data.data
90.
91.                 Bx=np.array([])
92.                 By=np.array([])
93.                 D = np.array([])
94.                 E = np.array([])
95.                 Ed=[]
96.
97.                 for i in range(0,self.n):
98.                     Bx=np.append(Bx, z_values[1+3*i])
99.                     By=np.append(By, z_values[2+3*i])
100.                    D=np.append(D, z_values[3*i]**(-1))
101.                    E=np.append(E,z_values[3*i]-self.d)
102.                    Ed.append([E[i]])
103.
104.                # Formation shape control
105.                BbDz=np.append([Bx],[By], axis=0)
106.                Dzt=np.diag(D)
107.                Ed=np.asarray(Ed)
108.
109.                # Estimator
110.                self.now = np.float64([rospy.get_time()])
111.                self.DT = self.now-self.old
112.
113.                if isinstance(self.mu_hat,float):
114.                    if z_values[1]<0 and z_values[2]<0 and z_values[-2]<0 and z_values[-1]>0:
115.                        mu_hat_dot = 2*(Ed[-1] - self.mu_hat) #Estimate edge of last robot]
116.                        self.mu_hat = self.mu_hat + mu_hat_dot * self.DT
117.                        self.S1bDz = np.array([[z_values[-2]/z_values[-3]], [z_values[-1]/z_values[-
    3]]])
118.                        print "estimating last agent (Triangle)"
119.                    else:
120.                        mu_hat_dot = 2*(Ed[0] - self.mu_hat) #Estimate edge of 1st found robot
121.                        self.mu_hat = self.mu_hat + mu_hat_dot * self.DT
122.                        self.S1bDz = np.array([[z_values[1]/z_values[0]], [z_values[2]/z_values[0]]])

123.                        print "estimating first agent (Triangle)"
124.                else:
```

```
                    if z_values[1]<0 and z_values[2]<0 and z_values[-2]<0 and z_values[-1]>0 and z_values [-5]>0:
                        mu_hat_dot = 2*(np.array([Ed[-1]- self.mu_hat[0], Ed[-2]- self.mu_hat[1]]))
                        self.mu_hat = self.mu_hat + mu_hat_dot * self.DT
                        self.S1bDz = np.array([[z_values[-2]/z_values[-3], z_values[-1]/z_values[-3]], \
                                               [z_values[-5]/z_values[-6], z_values[-4]/z_values[-6]]])
                        print "estimating last and penultimate agents (Multiple)"

                else:
                    mu_hat_dot = 2*(np.array([Ed[0]- self.mu_hat[0], self.mu_hat[1]]))#Ed[-1]- self.mu_hat[1]]))
                    self.mu_hat = self.mu_hat + mu_hat_dot * self.DT
                    self.S1bDz = np.array([[z_values[1]/z_values[0], z_values[2]/z_values[0]], \
                                           [np.float64(0) , np.float64(0)]])
                    print "estimating first agent (Multiple)"

                print 'Nexus'+self.name
                if self.n==2:
                    U = self.c*BbDz.dot(Dzt).dot(Ed) - 2.9*self.S1bDz.dot(self.mu_hat).reshape((2, 1))

                else:
                    U = self.c*BbDz.dot(Dzt).dot(Ed) - 2.9*self.S1bDz.dot(self.mu_hat)

                print "U = ", U

                # Saturation
                v_max = 0.2
                v_min = 0.02
                for i in range(len(U)):
                    if U[i] > v_max:
                        U[i] = v_max
                    elif U[i] < -v_max:
                        U[i] = -v_max
                    elif -v_min < U[i]+self.U_old[i]+self.U_oldd[i] < v_min : # preventing shaking
                        U[i] = 0

                # Set old U values in order to prevent shaking
                self.U_oldd = self.U_old
                self.U_old = U

                # Append 0 to error if no robot is detected to be able to plot later
                if self.n > 3:
                    self.E3_log = np.append(self.E3_log, 0)
                if self.n > 4:
                    self.E4_log = np.append(self.E4_log, 0)
                if self.n > 5:
                    self.E5_log = np.append(self.E5_log, 0)
                if self.n > 6:
                    self.E6_log = np.append(self.E6_log, 0)

                #Append error and velocity in Log arrays
                self.E1_log = np.append(self.E1_log, Ed[0])
                self.E2_log = np.append(self.E2_log, Ed[1])
                if self.n > 2:
                    self.E3_log = np.append(self.E3_log, Ed[2])
                if self.n > 3:
                    self.E4_log = np.append(self.E4_log, Ed[3])
                if self.n > 4:
                    self.E5_log = np.append(self.E5_log, Ed[4])
                if self.n > 5:
                    self.E6_log = np.append(self.E6_log, Ed[5])

                self.Un = np.float32([np.sqrt(np.square(U[0])+np.square(U[1]))])
                self.U_log = np.append(self.U_log, self.Un)

```

```
187.                # Save current time in time log array
188.                if self.k < 1:
189.                    self.begin = np.float64([rospy.get_time()])
190.                    self.k = 10
191.                self.now = np.float64([rospy.get_time()])
192.                self.time = np.float64([self.now-self.begin])
193.                self.time_log = np.append(self.time_log, self.time)
194.                self.old = self.now
195.                # publish
196.                self.publish_control_inputs(U[0], U[1])
197.
198.            elif 10 < self.running < 1000:
199.                self.shutdown()
200.
201.        def publish_control_inputs(self,x,y):
202.            ''''' Publish the control inputs to command velocities '''
203.
204.            self.velocity.linear.x = x
205.            self.velocity.linear.y = y
206.
207.            self.pub.publish(self.velocity)
208.
209.        def shutdown(self):
210.            ''''' Stop the robot when shutting down the controller_1 node '''
211.            rospy.loginfo("Stopping Nexus_"+str(int(sys.argv[1]))+"...")
212.            self.running = np.float32(10000)
213.            self.velocity = Twist()
214.            self.pub.publish(self.velocity)
215.
216.            rospy.sleep(1)
217.            pl.close("all")
218.            pl.figure(0)
219.            pl.title("Inter-agent distance error measured by Nexus "+str(int(sys.argv[1])))
220.            pl.plot(self.time_log, self.E1_log, label="e1_nx"+str(int(sys.argv[1])), color='b')
221.            pl.plot(self.time_log, self.E2_log, label="e2_nx"+str(int(sys.argv[1])), color='y')
222.            if self.n > 2:
223.                pl.plot(self.time_log, self.E3_log, label="e3_nx"+str(int(sys.argv[1])), color='g')
224.            if self.n > 3:
225.                pl.plot(self.time_log, self.E4_log, label="e4_nx"+str(int(sys.argv[1])), color='r')
226.            if self.n > 4:
227.                pl.plot(self.time_log, self.E5_log, label="e5_nx"+str(int(sys.argv[1])), color='c')
228.            if self.n > 5:
229.                pl.plot(self.time_log, self.E6_log, label="e6_nx"+str(int(sys.argv[1])), color='m')
230.            pl.xlabel("Time [s]")
231.            pl.ylabel("Error [m]")
232.            pl.grid()
233.            pl.legend()
234.            pl.savefig("/home/mariano/Desktop/Plots/Nexus_Distance_"+str(int(sys.argv[1]))+".png")
235.
236.            pl.figure(1)
237.            pl.title("Input velocity Nexus "+str(int(sys.argv[1])))
238.            pl.plot(self.time_log, self.U_log, label="pdot_nx"+str(int(sys.argv[1])), color='b')
239.            pl.xlabel("Time [s]")
240.            pl.ylabel("Velocity [m/s]")
241.            pl.grid()
242.            pl.savefig("/home/mariano/Desktop/Plots/Nexus_Velocity_"+str(int(sys.argv[1]))+".png")
243.            pl.pause(0)
244.
245.    if __name__ == '__main__':
246.        try:
247.            rospy.init_node('controller_'+str(int(sys.argv[1])), anonymous=False)
248.            controller()
249.            rospy.spin()
250.        except:
251.            rospy.loginfo("Controller node_"+str(int(sys.argv[1]))+" terminated.")
```