UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Autonomous Systems

DISTRIBUTED CONTROL APPLICATIONS

Professors:
Giuseppe Notarstefano
Ivano Notarnicola

Students:
Laura Calzoni
Francesca Paradiso
Matteo Periani

Academic year 2022/2023

Abstract

The project consists in two main tasks. The first one involves a data analytics application while the second one deals with the control for multi-robot systems in ROS2.
Task 1 concerns the classification of the image dataset `fashion_mnist`, supposing to have a team of N agents and that each agent can access only to a private subset of it. The problem is faced firstly approaching to a general consensus optimization problem, then realizing a (simpler) centralize training and concluding realizing its distributed version. The distributed training is based on the Gradient Tracking method. The Sigmoid function has been used as activation function while the Binary Cross-Entropy (BCE) has been chosed as loss function. At the end, the accuracy is computed to evaluate the quality of the obtained solution.
Task 2 instead demands the implementation of a distributed realization for potential-based Formation Control. The algorithm must include a solution for avoiding collisions between agents, as well as a control design in order to steer the formation toward different target positions or trajectories. The effectiveness of the implemented algorithm is tested for multiple desired 3D formations, with a different number of agents.

Contents

Introduction **5**

1 Distributed Classification via Neural Networks **6**
1.1 Distributed Optimization 6
1.1.1 Python implementation 8
1.1.2 Results 8
1.2 Centralized Training 10
1.2.1 Python Implementation 12
1.2.2 Results 12
1.3 Distributed Training 13
1.3.1 Python implementation 14
1.3.2 Results 15

2 Formation Control **17**
2.1 Desired formations 17
2.2 Problem setup 19
2.2.1 Discretization 20
2.2.2 ROS2 implementation 20
2.2.3 Results 21
2.3 Collision avoidance 23
2.3.1 ROS2 implementation 24
2.3.2 Results 24
2.4 Moving Formation 25
2.4.1 ROS2 implementation 26
2.4.2 Results 26
2.5 Obstacle avoidance 27
2.5.1 ROS2 implementation 28
2.5.2 Results 28

Conclusions **29**

Introduction

Motivations

The aim of this project is to show the effectiveness of distributed algorithms in different frameworks.
A distributed algorithm is designed to solve computational problems in a distributed computing environment, where groups of interconnected nodes work together to achieve a common goal. These algorithms are often used in situations where a centralized approach is not feasible or desirable, such as in large-scale systems or cloud computing environments.
Distributed systems offer several advantages over centralized systems. First of all, since they distributes the computational load across multiple nodes, they are able to handle increasing workloads and larger volumes of data. Secondly, they are build to enable parallel processing, which significantly improves performance compared to centralized algorithms, specially in terms of computational time. By utilizing the combined resources of multiple machines, distributed systems can achieve higher performance and faster response times. Furthermore, due to their decentralized nature and redundancy, distributed systems tend to be more reliable. Indeed, in this case, if one node fails or becomes unavailable, others can continue to provide services, minimizing the impact on overall system availability.
However, distributed design solutions also introduce complexities in terms of design, implementation, management and resource allocation and it must be guaranteed the consistency and correctness of the results despite potential failures or delays in the nodes' communication. Some common problems addressed by distributed algorithms include distributed consensus, leader election, distributed mutual exclusion, distributed resource allocation, and distributed data storage and retrieval.

Contributions

The development of our project finds place within the studies on the efficiency and effectiveness of distributed algorithms, in particular for distributed training of neural networks and for formation control tasks.

Chapter 1

Distributed Classification via Neural Networks

During the past years, the field of Artificial Intelligence grows exponentially and Neural Networks have become the standard way to address difficult task. Problems like classification, image segmentation, language processing and so on, are nowadays easily solved by NN. In addition, the growth of the problems' complexity and of the dataset size led to the use of distributed algorithms to handle these situations. In particular, distributed classification via Neural Networks refers to the process of performing classification tasks using a distributed computing system, where multiple NN models collaborate to make predictions.

In this chapter of the project is requested to solve firstly a distributed optimization problem and successively to solve a binary image classification task, both in centralized and distributed setup.

1.1 Distributed Optimization

Distributed optimization refers to the process of optimizing a function or finding the optimal solution to a problem by dividing the computational workload across multiple computing nodes. Consider a team of N agents, let's denote the local optimum estimate of agent $i \in \{1...N\}$ at time $k \geq 0$ with $u_i^k \in R^d$ and thus represent with $u^k \in \mathbb{R}^{dN}$ the stack vector of the agents estimates. Call N_i the set of neighbors of agent i, according to the agents graph.

The cost is usually designed as a quadratic function, in order to have a

convex optimization problem with only one global minimum, and results:

$$J_i(u) = \frac{1}{2}u^T Qu + Ru$$

with gradient :

$$\frac{\partial J_i}{\partial u} = Qu + R$$

where $u \in \mathbb{R}^d$,$R \in \mathbb{R}^N$ and $Q \in \mathbb{R}^{Nd^2}$ is positive definite.
The associated optimization problem can be written as

$$\min_u \sum_{i=1}^{N} J_i(u)$$

where each function J_i is local and private. Each agent, without sharing data, should contribute in equal amount to the optimization problem and, to converge all to an unique optimal solution $\mathbf{x}^*$, it's necessary to reach consensus.
In standard setups, the optimization problem can usually be solved updating the solution's estimate according to the negative gradient direction of the objective function $-\nabla J$. However, in a distributed environment, the total gradient corresponds to:

$$\nabla J = \sum_{i=1}^{N} \nabla J_i$$

which is a global term (since it involves all the states), not known by the single agents.
For this reason, a distributed optimization problem can be approached using the distributed *gradient tracking algorithm*, in which the tracking of the global gradient is performed in order to be used as descent direction for the states' updates. It reads:

$$u_i^{k+1} = \sum_{j \in N_i} a_{ij} u_j^k - \alpha s_i^k$$

$$s_i^{k+1} = \sum_{j \in N_i} a_{ij} s_j^k + \nabla J_i(u_i^{k+1}) - \nabla J_i(u_i^k)$$

with

$$x_i^0 \in \mathbb{R}$$

$$s_i^0 = \nabla J_i(x_i^0)$$

where $\alpha \in \mathbb{R}$ are the stepsize, $u_i^k \in \mathbb{R}^d$ are the agent minimum estimates, $s_i^k \in \mathbb{R}^d$ is the local agent estimate of the cost function gradient according to the consensus algorithm and $a_{ij} \in \mathbb{R}$ are the graph weights.

The weights matrix of the agents graph is computed using the *Metropolis Hasting* method, which is reported below:

$$a_{ij} = \begin{cases} \frac{1}{\max(d_i, d_j)} & \text{if } (i,j) \in E \text{ and } i \neq j \\ 1 - \sum_{h \in N_i \setminus \{i\}} a_{ih} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

where $d_i \in \mathbb{R}$ is the degrees (number of neighborhoods) of the agent i and E is the edge set of the graph.

1.1.1 Python implementation

In the executable *task1_1.py* we implemented the whole algorithm, whose parameters are:

- `NN`: the number of the agents
- `MAX_ITERS`: the maximum number of iterations per agent
- `d`: the size of the input
- `graph_type`: the type of the graph (Cycle, Star, Path)

The agents graph and the weights matrix are computed by the helper functions in the file *utils.py*. The distributed algorithm explained before is implemented through a for-loop, where at each iteration (loop cycle) every agent computes the cost function and its gradient, updates its local minimum estimates via the distributed gradient algorithm and stores the new value. At the end, the consensus error, the cost evolution and the norm of the gradient are plot to check the consistency of the results.

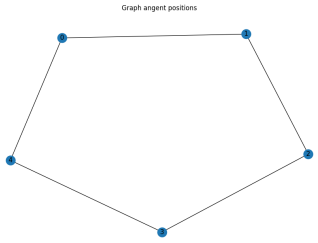
To run the executable, go in `task1` directory and type:

```
$ python3 task1_1.py <NN> <MAX_ITERS> <d> <graph_type>
```

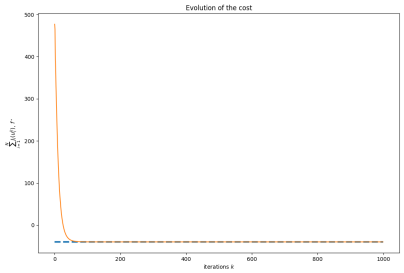
1.1.2 Results

The following plots represent the results of the distributed optimization. We test our algorithm with the different graph types (Cycle, Star, Path), with the others parameters fixed, in the following way:

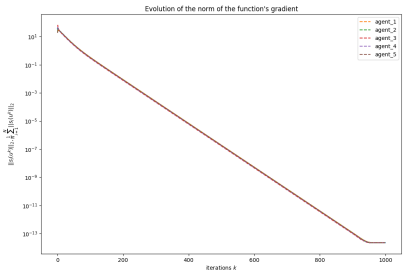
- `NN` $= 5$
- `MAX_ITERATIONS` $= 1000$
- `d` $= 3$
- $\alpha = 0.01$



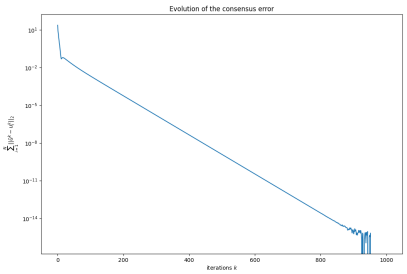
(a) graph_type



(b) cost evolution

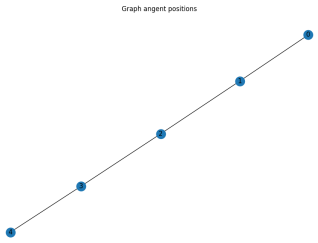


(c) norm of the gradient evolution

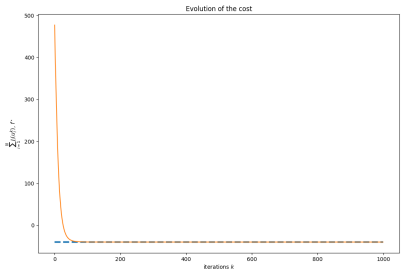


(d) consensus error

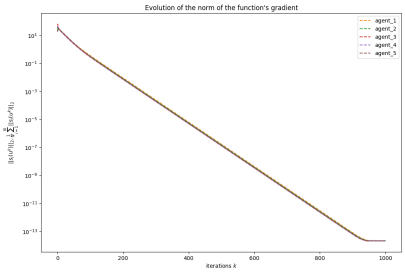
Figure 1.1: *graph_type = Cycle*



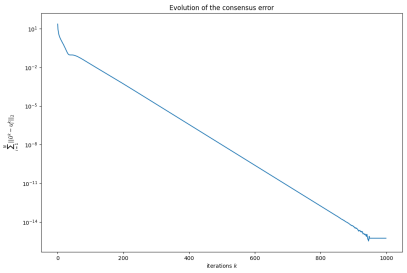
(a) graph_type



(b) cost evolution

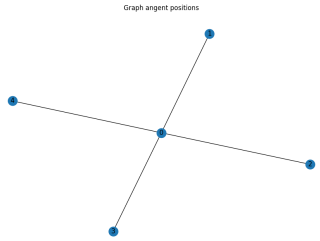


(c) norm of the gradient evolution

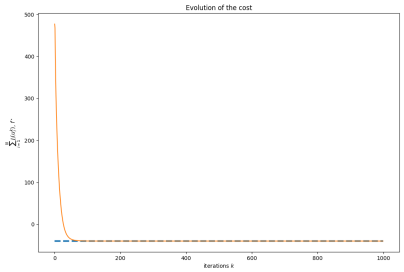


(d) consensus error

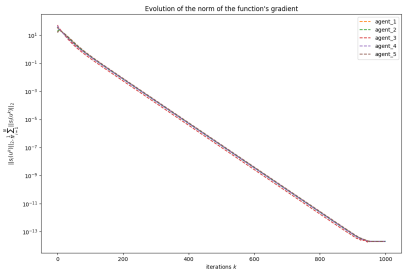
Figure 1.2: *graph_type = Path*



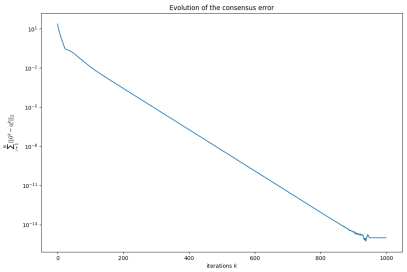
(a) graph_type



(b) cost evolution



(c) norm of the gradient evolution



(d) consensus error

Figure 1.3: *graph_type = Star*

In all of the trials, we can see that the cost function J correctly decreases as the gradient's estimates approach to zero. Furthermore, we can see from the consensus error plots, that agent reach consensus, so we are sure the agents states properly converge to the true global minimum.

1.2 Centralized Training

In this step, we were required to train a small Neural Network to perform a binary classification task over the *fashion mnist* dataset. Firstly, we freely selected a single category (Sneakers) among those present and we transformed the whole dataset into binary one, setting to *1* the elements that correspond to the chosen category and assigning *0* to the others. Then, we reduced the number of samples, in order to have a balanced dataset with half of the items with label one and the rest with zero. At the end, we down-sampled the images to speed up the computation, and we shuffled the samples, ensuring that model remains general and does not overfit. This process is applied to both the train and the test set.

We designed a flat network with linear layers, that performs at each step the matrix product between the input data and the weights of the corresponding

layer. In particular, considering our dataset built with $((D^1, y^1), ..., (D^I, y^I))$ I multiple samples of labelled data, every input sample $D^i \in \mathbb{R}^d$ is elaborated by the network as follows:

$$x_{h,0} = [D]_h$$

$$x_{h,t+1} = \sigma(x_t^\top u_{h,t}) \quad \forall h = 1, \dots, d, \; \forall t = 1, \dots, T$$

where $d \in \mathbb{N}$ is the number of neurons, $T \in \mathbb{N}$ is the number of layer, $[D]_h$ is the h-component of a data sample and $u_{h,t} \in \mathbb{R}^d$ are the weights of the network (ignoring the bias w.l.o.g).
As activation function we used the *sigmoid* function $\sigma(\xi)$:

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}$$

The network is correctly trained when the condition $x_T^i \approx y^i$ is satisfied, where $y^i \in \{0, 1\}$ is the label associated to the train data D^i. For this purpose, we had to found the best weight vector $\mathbf{u}^* \in \mathbb{R}^{d^2 T}$, by solving the following optimization problem:

$$\mathbf{u}^* = arg\min_{\mathbf{u}} \sum_{i=1}^{I} J(x_{i,T}; y^i) = arg\min_{\mathbf{u}} \sum_{i=1}^{I} J(\phi(\mathbf{u}, x_{i,0}); y^i)$$

where ϕ is the *shooting map* and J is the *loss function*. A suitable loss function could be the *Binary Cross-Entropy*, that states:

$$J = \sum_{i=1}^{I} (y^i \log(\phi(\mathbf{u}, D^i)) + (1 - y^i) \log(1 - \phi(\mathbf{u}, D^i)))$$

and whose gradient can be calculated as:

$$\frac{\partial J}{\partial \hat{y}} = \frac{y^i}{\hat{y}^i} - \frac{1 - y}{1 - \hat{y}^i}$$

calling $\hat{y}^i = \phi(\mathbf{u}, D^i)$.
So, the optimization problem can be solved using the (batch) gradient method through the following formula:

$$\mathbf{u}^{k+1} = \mathbf{u}^k - \alpha \sum_{i=1}^{I} \nabla J(\phi(\mathbf{u}, x_{i,0}); y^i), \quad k \in \mathbb{N}$$

where $\alpha \in \mathbb{R}$ is the step-size.

1.2.1 Python Implementation

In the executable *task1_2.py* we defined all the algorithm. The algorithm parameters are:

- `TARGET`: the number of the class to treat as positive
- `MAX_EPOCHS`: the he maximum number epochs
- `TRAIN_SIZE`: the size of train set
- `TEST_SIZE`: the size of test set
- `STEP_SIZE`: the learning rate
- `T`: the number of network layers
- `d`: the number of layer neurons

The data acquisition and pre processing are done by the helper function that could be found in the file *data_preparation.py*. The neural network structure is defined in the file *modelling_nn.py*, where the number of layers T and the number of neurons d parameters are defined. The training of the network explained before is implemented through a for-loop where, at each iteration (or loop cycle), every training data is firstly elaborated through the network to compute the loss (forward-pass) and then the weights are optimized, based on the obtained result (backward-pass). At the end of the training, the accuracy over train and test sets is computed to evaluate the network performances. Finally, the evolution of the loss function and the norm of its gradient are plotted to check the correctness of the implementation.

To run the relative code, go in `task1` directory and type:

```
$ python3 task1_2.py <MAX_EPOCHS> <TRAIN_SIZE> <TEST_SIZE> <STEP_SIZE>
```

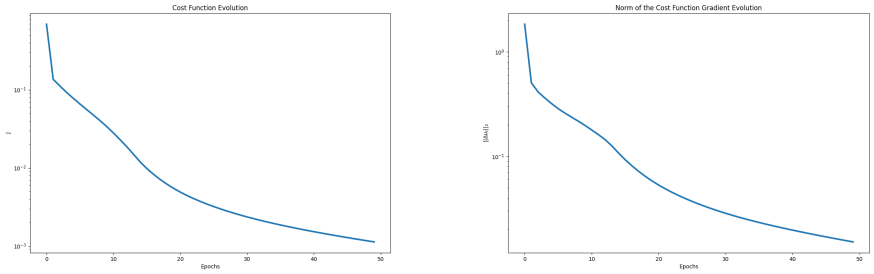
1.2.2 Results

The following plots represent the results of the centralized training. We test our algorithm fixing the train size and the image resolution in order to speed up the training process. We set:

- `TARGET` $= 7$
- `TRAIN_SIZE` $= 512$
- `TEST_SIZE` $= 128$
- `MAX_EPOCHS` $= 50$

- `STEP_SIZE` $= 0.1$

For the network configuration, we use 3 layers, each with 400 neurons (in this way we have a 20x20 images). At the end, only the output of the first neuron is used to compute the loss and the gradient, masking the other in order to not effect the train process.



(a) loss evolution (b) norm of the gradient evoution

Figure 1.4: Train plot

In the following table it is reported the network accuracy over the train and the test data:

	Train	Test
accuracy	98.35 %	96.78%

We decide to not test our algorithm with different parameters because the total computation is quite heavy. Furthermore, we know that since we trained the network with balanced dataset, both its size and the target class chosen don't impact too much on the train process. The step-size, instead, in this case only influences the number of iterations the network need to learn how to classify the data, but not the performances. Increasing the number of epochs we will obtain a better generalization over the data.

1.3 Distributed Training

The last assignment fro this chapter is to distribute the training of the previous network over multiple agents setup. The objective of this exercise is to simulate a scenario where multiple agents aim to solve a shared classification task having the knowledge of their local data only.

The data configuration is the same of the section 1.2, the only difference is that train and test data are equally split across the agents. What changes is the how the optimization is performed. Indeed, in a distributed setup the classic network training methods cannot be used, as distributed optimization algorithms are needed to compute the weights update.

Applying the general distributed optimization problem to Neural Network training, the system should minimize the sum of the agent local loss functions J_i:

$$\min_u \sum_{i=1}^{N} J_i(u)$$

with

$$J_i = \sum_{h=1}^{I} (y^h \log(\phi(\mathbf{u}, D^h)) + (1 - y^h) \log(1 - \phi(\mathbf{u}, D^h)))$$

where N is the number of agents, I is the number of training data for each agent and D^h is the h-th sample.
The optimization problem cam be solved via *distributed gradient tracking* (already implemented in 1.1), in which, the weights of the network are updates as follows:

$$\mathbf{u}_i^{k+1} = \sum_{j \in N_i} a_{ij} \mathbf{u}_j^k - \alpha \mathbf{s}_i^k$$

$$\mathbf{s}_i^{k+1} = \sum_{j \in N_i} a_{ij} \mathbf{s}_j^k + \nabla J_i(\mathbf{u}_i^{k+1}) - \nabla J_i(\mathbf{u}_i^k)$$

where $\mathbf{u}_i^k \in \mathbb{R}^{Tdd+1}$ is the i-th agent's update of the network weights' estimates, $a_{ij} \in \mathbb{R}$ are the agents weights, $\mathbf{s}_i^k \in R^{Tdd+1}$ is the estimate of the loss function gradient of the i-th agent.

1.3.1 Python implementation

The distributed training is implemented in the executable *task1_3.py*. The algorithm parameters are:

- `N_AGENTS`: number of the agents
- `TARGET`: number of the classes to treat as positive
- `N_EPOCHS`: maximum number of epochs
- `TRAIN_SAMPLES_PER_AGENT`: size of the local agent train set
- `TEST_SAMPLES_PER_AGENT`: size of the local agent test set
- `STEP_SIZE`: learning rate
- `T`: number of network layers
- `d`: number of neurons per layer

The data acquisition and pre-processing are done by the helper function that could be found in *data_preparation.py*. The neural network structure is defined in *modelling_nn.py*, as before. The distributed training is implemented through a for-loop as explained in the previous section, where at each iteration the forward pass of data is followed by the backward-pass of weights. As before, the accuracy over train and test sets of a selected agent is computed to check the system performance.

To run this task, in `task1` directory type:

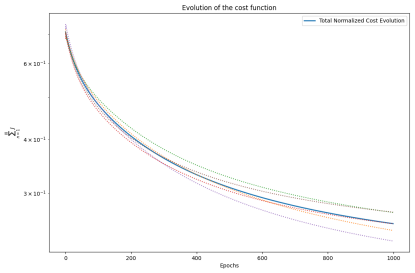
```
$ python3 task1_3.py <NN> <MAX_EPOCHS> <TRAIN_SAMPLE> <TEST_SAMPLE> <STEP_SIZE>
```

1.3.2 Results

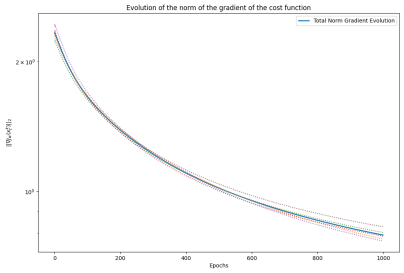
The following plots represent the results of the distributed training. We test our algorithm fixing all the parameters due heavy computation requirements. The algorithm parameters are:

- `N_AGENTS` = 5
- `TARGET` = 7
- `N_EPOCHS` = 1000
- `TRAIN_SAMPLES_PER_AGENT` = 256
- `TEST_SAMPLES_PER_AGENT` = 64
- `STEP_SIZE` = 0.005

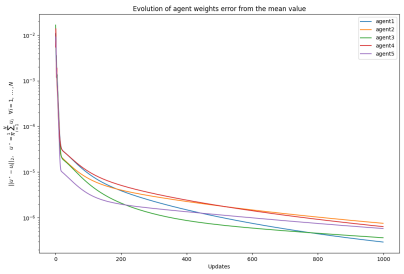
For the network configuration, we use 3 layers, each with 400 neurons (in this way we have a 20x20 images).



(a) loss evolution



(b) norm of the gradient evolution



(c) consensus error evolution

Figure 1.5: Train plot

In the following table it is reported the network accuracy over the train and the test data of the $agent_0$:

	Train	Test
accuracy	92.97 %	92.19%

Chapter 2

Formation Control

In recent years, the concept of formation control has gained significant attention as an effective way to orchestrate the movements and behaviors of a group of robots. In particular, formation control involves the coordination of multiple agents to achieve a desired spatial arrangement, resulting in a coherent and synergistic behavior of the group. Unlike individual robot control, where each agent operates independently, formation control leverages inter-robot communication, collaboration, and shared objectives to achieve common goals. In a distributed environment, this result is obtained with agents that are able to make decisions based on local information only.

For this project, among the various typologies, distance-based control has been approached, in which inter-agent distances are actively controlled in order to achieve the desired formation. Each agent can sense the relative positions of its neighbors with respect to its own local coordinate systems. The reference formation is specified in terms of desired distances between any pair of agents and can be treated as a given rigid body. Thus, to achieve the desired position by controlling the inter-agent distances, the communication graph between agents needs to be rigid.

2.1 Desired formations

As mentioned, graph rigidity characterizes the requirement on undirected interaction graphs in distance-based formation control. Considering an undirected graph $\mathcal{G} = (\mathcal{I}, \mathcal{E})$ and N agents $x_i \in \mathbb{R}^n$, then $x \in \mathbb{R}^{Nn}$ is said to be a realization of $\mathcal{G}$ in $\mathbb{R}^n$, while the pair $(\mathcal{G}, x)$ is called a framework of $\mathcal{G}$ in $\mathbb{R}^n$. Suppose that N is greater than $n+1$, then the framework $(\mathcal{G}, x)$ is said to be minimally rigid if the cardinality of the edge set is $|E| = Nn - n(n+1)/2$. In order to verify the performances of the implemented control, we tested different geometric formation patterns, with a different number of agents

each, all at least minimally rigid. It must be considered that the more the connections between agents are, the more the formation is facilitated. As a consequence, we tried to reach a good trade-off between the complexity of the desired spatial arrangement and the degrees of connectivity of the communication graph. Since the agents were 3-dimensional (i.e $n = 3$), we designed several solid shapes:

- pyramid with a square base
  - number of agents N = 5
  - base edge L = 2
  - height H = 3
  - diagonal of the base D = $L\sqrt{2}$
  - lateral edge l = $\sqrt{H^2 + L^2/2}$

$$distances = \begin{bmatrix} 0 & L & 0 & L & l \\ L & 0 & L & D & l \\ 0 & L & 0 & L & l \\ L & D & L & 0 & l \\ l & l & l & l & 0 \end{bmatrix}$$

- cube
  - number of agents N = 8
  - base edge L = 3
  - diagonal of the base d = $L\sqrt{2}$
  - diagonal of the cube D = $L\sqrt{3}$

$$distances = \begin{bmatrix} 0 & L & d & L & L & 0 & D & d \\ L & 0 & L & 0 & d & L & 0 & D \\ d & L & 0 & L & D & d & L & 0 \\ L & 0 & L & 0 & 0 & D & d & L \\ L & d & D & 0 & 0 & L & d & L \\ 0 & L & d & D & L & 0 & L & 0 \\ D & 0 & L & d & d & L & 0 & L \\ d & D & 0 & L & L & 0 & L & 0 \end{bmatrix}$$

- regular octahedron
  - number of agents N = 6
  - edge L = 1
  - diagonal D = $L\sqrt{2}$

$$distances = \begin{bmatrix} 0 & L & L & L & L & D \\ L & 0 & L & 0 & L & L \\ L & L & 0 & L & D & L \\ L & 0 & L & 0 & L & L \\ L & L & D & L & 0 & L \\ D & L & L & L & L & 0 \end{bmatrix}$$

- regular pentagon (fixing the third dimension at zero)
  - number of agents N = 5
  - edge L = 1
  - diagonal $D = L(\sqrt{5}+1)/2$

$$distances = \begin{bmatrix} 0 & L & L & D & 0 \\ L & 0 & D & L & D \\ L & D & 0 & 0 & L \\ D & L & 0 & 0 & L \\ 0 & D & L & L & 0 \end{bmatrix}$$

2.2 Problem setup

Consider a team of N robots. Let's denote the position of robot $i \in \{1...N\}$ at time $t \geq 0$ with $x_i(t) \in R^3$ and thus represent with $x(t) \in R^{3N}$ the stack vector of the continuous positions. Call N_i the set of neighbors of agent i, according to the desired formation.
The dynamics of each agent can be written as the sum of derivatives of local potential function V_{ij} as:

$$\dot{x_i}(t) = -\sum_{j \in N_i} \frac{\partial V_{ij}}{\partial x_i} \qquad \forall i = \{1, .., N\}$$

being N_i the set of neighbours of the i-th agent.
Choosing as potential function:

$$V_{ij}(x) = \frac{1}{4}(\|x_i - x_j\|^2 - d_{ij}^2)^2$$

with $d_{ij} \in \mathbb{R}$ the assigned relative distances between two robots, the agents' dynamics are given by:

$$\dot{x_i}(t) = -\sum_{j \in N_i} (\|x_i(t) - x_j(t)\|^2 - d_{ij}^2)(x_i(t) - x_j(t)) \qquad \forall i = \{1, .., N\}$$

Thus, the equilibrium of the whole system is reached in a configuration for which $\forall i = \{1, ..., N\}, \quad \dot{x_i}(t) = 0$. It can be seen that, depending on the initial condition, the algorithm can converge either to:

- the desired formation if

$$\sum_{j \in N_i} (\|x_i(t) - x_j(t)\|^2 - d_{ij}^2) = 0 \quad \forall i = \{1, .., N\}$$

  since in this way $\|x_i(t) - x_j(t)\|^2 = d_{ij}^2 \quad \forall i = \{1, .., N\}, \forall j \in N_i$

or to

- a randez-vous if

$$\sum_{j \in N_i} (x_i(t) - x_j(t)) = 0 \quad \forall i$$

  since in this way $x_i(t) = \sum_{j \in N_i} x_j(t), \ \forall i = \{1, .., N\}$, and all the agent states collapse on each other.

With this approach, the formation is represented as a configuration of virtual attractive local potentials. Each robot perceives them and adjusts its motion accordingly, in order to follow the negative direction of the gradient. Indeed, since the local potentials are defined monotonically increasing with the distance between the desired and the actual inter-agent relative position, following the negative gradient each robot is attracted towards its target location in the formation, with a force proportional to the position error.

2.2.1 Discretization

After have focused on the general algorithm, we implemented its discretized version exploiting the forward Euler method. Denoting with $p_i^k \in \mathbb{R}^3$ the discretized position at time $k \in \mathbb{N}$ of the i-th robot, the agents' dynamics result:

$$p_i^{k+1} = p_i^k - dt \sum_{j \in N_i} (\|p_i^k - p_j^k\|^2 - d_{ij}^2)(p_i^k - p_j^k) \qquad \forall i = \{1, .., N\}$$

with a time-step dt equal to 0.01. All the agents' states have been randomly initialized.

2.2.2 ROS2 implementation

In order to be able to implement the formation control as a distributed algorithm, we exploited ROS2. In the executable *the_agent_formation.py* we defined each agent as a class of type *Node* that receives the following parameter from the corresponding launch file:

- `agent_id` = its own index
- `n_p` = the dimension of each agent
- `dt` = the discretization time step, equal to the visualization frequency of the Rviz simulation environment, which is $\frac{1}{100}$
- `max_iters` = the maximum number of iteration per agent

Furthermore, from the terminal we provide the information about the desired formation to achieve, from the ones presented before. The i-th agent node publishes its own position on its topic (`topic_i`) and reads from `topic_j` ($\forall$ `j` in `neigh`) the position of its neighbours.
The distributed algorithm explained before is implemented in the function `timer_callback`, which is run by each agent using its private data at every clock. Each clock of the timer corresponds to an iteration in time and at the end of every cycle the agent updates its position. The published message includes both the id of the relative agent and its updated position. Such message will then be read in the `listener_callback` function (of the agent's neighbours) and added to the list of the neighbours' positions with the corresponding indexes.
In the `listener_callback`, we also synchronize the robots by waiting for all the expected messages to be received.
The launch file *task2_formation.launch.py* launches the Rviz visualization, all the Agent nodes (with the proper launch parameters, as explained before) and each *visualizer.py* file, responsible for the agents representation in Rviz

To run the formation, go in `task2` directory and, after having built the package, type:
`$ export formation_shape=cube` (any desired shape)
`$ ros2 launch task2 task2_formation.launch.py`

2.2.3 Results

The following figures represent the result of the formation process. In particular in *Figure 2.1* we can see the agents' positioning in the space and how they all successfully reach formation. *Figure 2.2* instead, show the error between each agent and its neighbours during such phase.

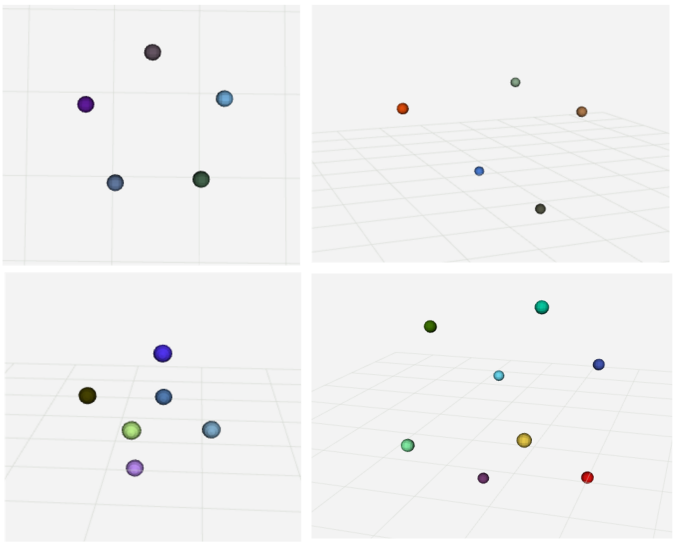


Figure 2.1: *Rviz visualization of the agents in formation*

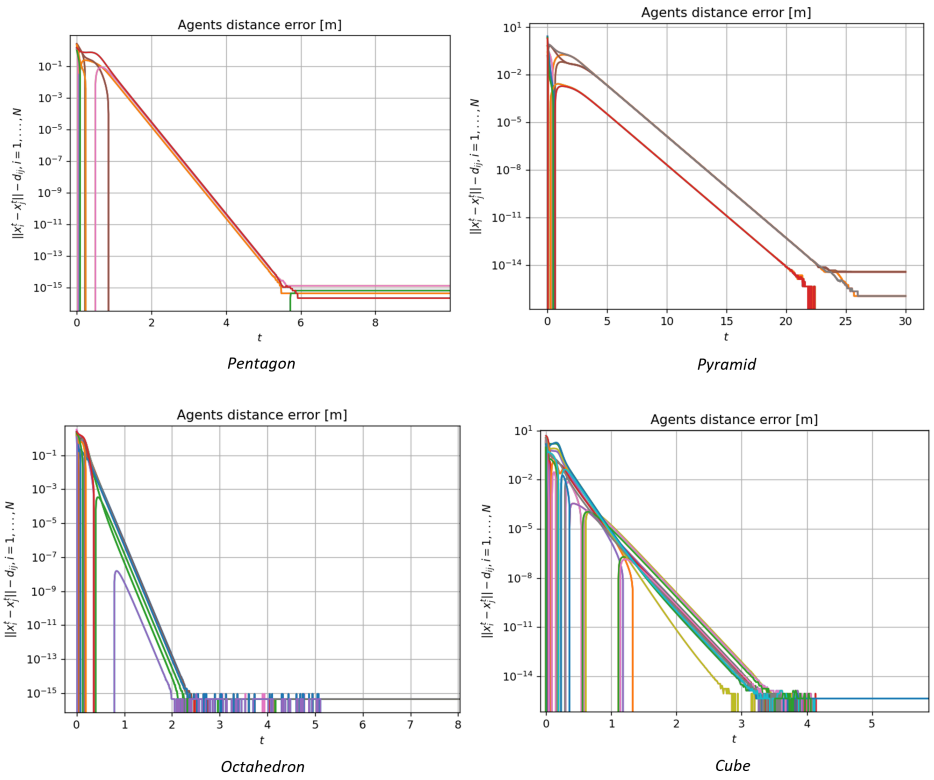


Figure 2.2: *Formation error*

Below, in *Figure 2.3*, it's visualized the behaviour in time of the potential function associated to each desired formation. It can be seen that it correctly decreases, since the agents follows the direction of its negative gradient.

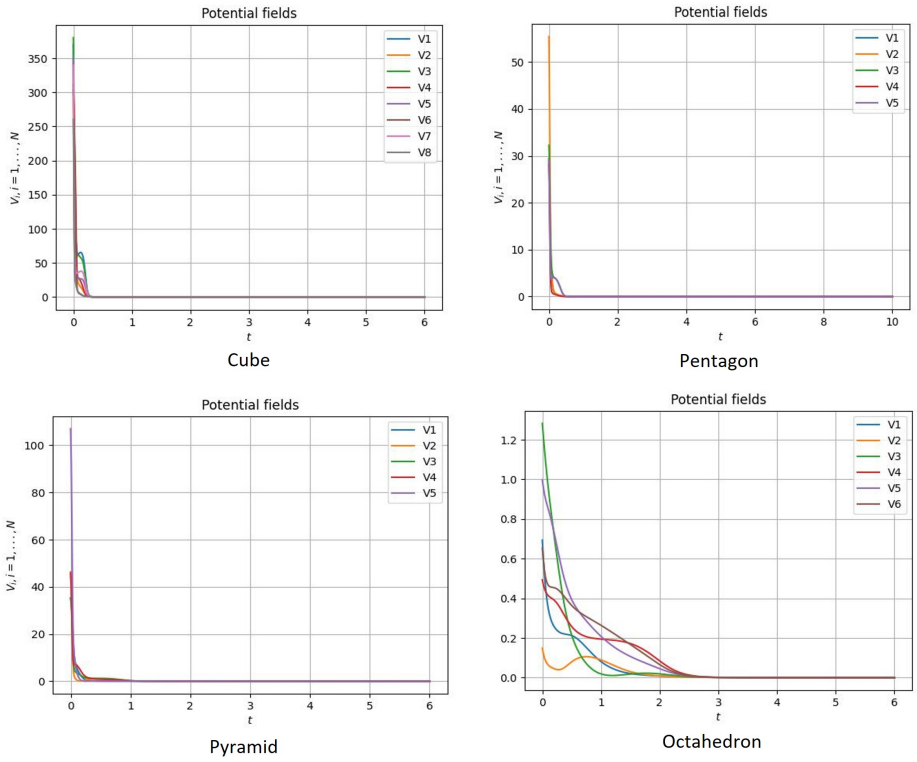


Figure 2.3: *Trend over time of the potential associated to each desired formation*

2.3 Collision avoidance

As mentioned above, in reaching the formation the agents may collide between each other. This behaviour should be obviously avoided, since in real application it would correspond to physical robots crashing to each other. For this purpose, we modified the previous control law by introducing a barrier function, which adds a repulsive potential to the total field. The repulsive potential should act as a barrier that pushes the robots away when they get too close to each other, with a force that augments as their relative distance decreases. Consequently, a proper choice for the collision avoidance barrier function V_{ij}^{barr} (between the i-th agent and its j-th neighbour) could be the following:

$$V_{ij}^{barr}(p^k) = -\log(\|p_i^k - p_j^k\|^2)$$

Indeed, the more an agent approximates to one of its neighbors the higher is the barrier and its potential contribution. The drawback of this approach is that collisions between two non-neighbouring agents cannot be prevented, as they cannot known each other state.
The partial derivative of each repulsive potential with respect to its corresponding agent' state results:

$$\frac{\partial V_{ij}^{barr}}{\partial p_i} = -\frac{2(p_i^k - p_j^k)}{\|p_i^k - p_j^k\|^2}$$

and thus, the dynamics of each agent becomes, $\forall i = \{1, .., N\}$:

$$p_i^{k+1} = p_i^k - dt \sum_{j \in N_i} \left((\|p_i^k - p_j^k\|^2 - d_{ij}^2)(p_i^k - p_j^k) - \frac{2(p_i^k - p_j^k)}{\|p_i^k - p_j^k\|^2} \right)$$

In order to be able to verify the effectiveness of the implemented approach, we initialized the agents states randomly between 0 and 0.1, to place them near to each other and possibly forcing collisions to happen.

2.3.1 ROS2 implementation

In the executable *the_agent_collision.py* we just changed the initialization of the agents and added the barrier function in the `timer_callback` code, as reported in the section above. The corresponding launch file is called *task2_collision.launch.py* and to launch it run in the command line:

```
$ export formation_shape = octahedron    (any desired shape)
$ ros2 launch task2 task2_collision.launch.py
```

2.3.2 Results

Figure 2.4 shows the formation error between each agent and its neighbours in the collision avoidance case. As we can see the error rapidly decreases as the agents reach the formation sooner.

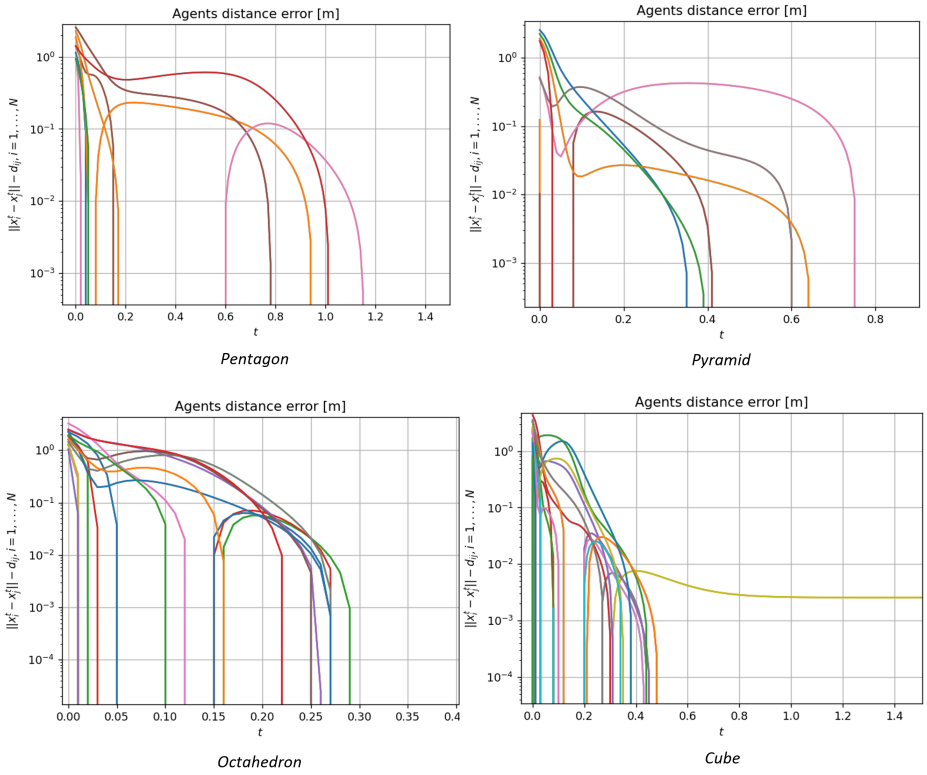


Figure 2.4: *Formation error with collision avoidance*

2.4 Moving Formation

As required, the next step was to realize a moving formation, declaring one (or more) agent as leader and implementing a control law to steer the formation toward a target position. Firstly, we randomly select a leader and assign to it a desired, fixed position. In order to be able to reach the reference point, a term proportional to the position error must be added to dynamics of the leader, as follows:

$$p_i^{k+1} = p_i^k + dt \left(- \sum_{j \in N_i} (\|p_i^k - p_j^k\|^2 - d_{ij}^2)(p_i^k - p_j^k) + K_p(p_i^k - p^{des}) \right)$$

being $p^{des} \in \mathbb{R}^3$ the desired position (invariant in time) and $K_p \in \mathbb{R}$ the proportional gain. The control law of the followers, instead, remains unchanged. In this way, as the leader starts moving toward the goal, the other agents follow and the formation is achieved and maintained during the motion. It must be taken into account that, the bigger K_p the faster the leader is steered to the target, but the greater would be also the oscillations generated during the motion and around the set-point. We found that a value

of 5 is a good trade-off. We have then tested the results for different target points.
As second test, we chose to design a reference curve in space and control the leader so that it both reaches the trajectory and moves along it. Namely, we designed circular path at a fixed height, whose equation is given by:

$$p^{des}(k) = [3 + 2\cos(0.01k), 3 + 2\sin(0.01k), 5]$$

In this case, we set the gain $K_p = 20$.

2.4.1 ROS2 implementation

Now, in *the_agent_leader_follower.py* the Agent node also contains the definition of the desired target. While the set-point is assigned differently in each formation, the circular trajectory is calculated in the `timer_callback` in the same way for every case. The id of the leader is given as an internal parameter of the class, as well as the value of the proportional gain. The launch file *task2_leader_follower.launch.py*, apart from containing the previous executables, it also includes a `goal_visulizer` to plot the reference point in the Rviz environment. The type of reference to reach (i.e. a fixed position or the time-variant trajectory) must be specified by terminal:

```
$ export formation_shape=pyramid     (any desired shape)
$ export reference=point     (desired target type)
$ ros2 launch task2 task2_leader_follower.launch.py
```

2.4.2 Results

The figure below shows the formation error of each agent with its neighbours in the case of a moving leader. As we can see some agents take more time reaching a stable pose with respect to their neighbours as they are trying to maintain the formation while the leader is moving towards the goal.

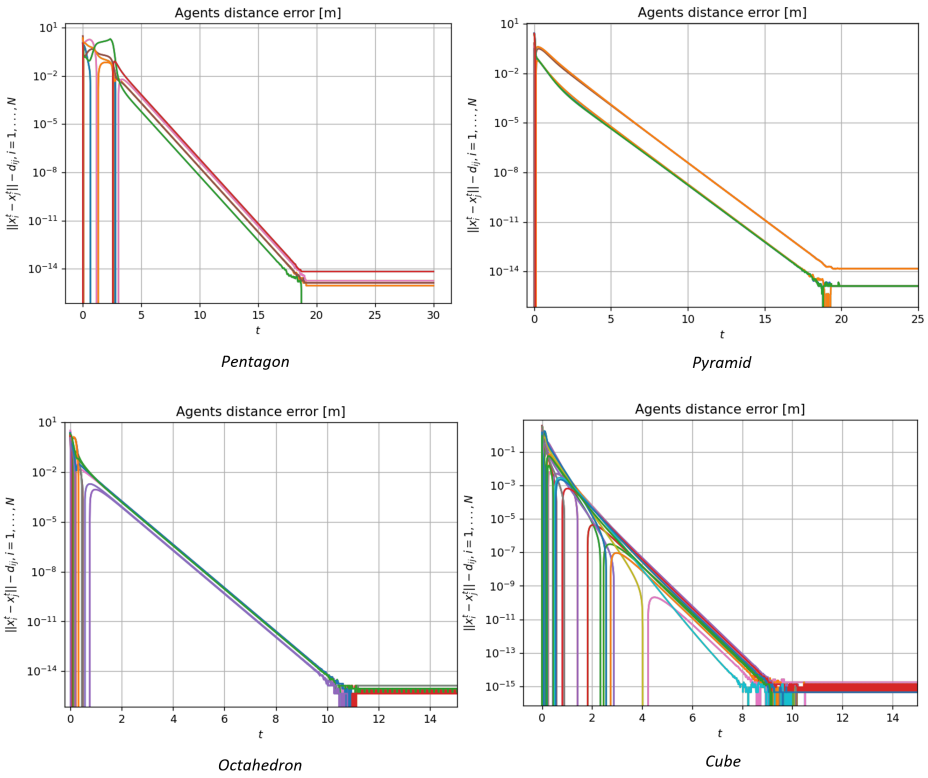


Figure 2.5: *Formation error with moving leader*

The following videos show how the formation is reached and maintained while the leader follows the circular trajectory. As an example, we reported just the cases of the cube and the octahedron.

2.5 Obstacle avoidance

As a conclusion for our project, we included a way to avoid also fixed obstacles. We added a repulsive potential for each obstacles, implementing a barrier functions for each of them. Calling $p^{obs} \in \mathbb{R}^3$ the obstacle position, chosen freely by us, the barrier function V_{io} can be designed as:

$$V_{io}(p^k) = -\log(\|p_i^k - p^{obs}\|^2)$$

similarly to the one devoted to the collision avoidance. Therefore, if we suppose that all the agents can sense the obstacles, the control law to implement results, $\forall i = \{1, .., N\}$:

$$p_i^{k+1} = p_i^k - dt \sum_{j \in N_i} \left((\|p_i^k - p_j^k\|^2 - d_{ij}^2)(p_i^k - p_j^k) - \frac{2(p_i^k - p^{obs}}{\|p_i^k - p^{obs}\|^2} \right)$$

If we don't weight the two potential contributions differently, the agents would try to avoid the obstacles maintaining the formation during the motion, as much as possible. So, in order to speed up the convergence, we weighted the potential of the formation inversely proportional to the distance from the goal. In this way, when the agents are still far from the target, they care less to reach the desired formation, so they avoid the occlusions in the path more easily.

Finally, we merged all the contributions together, in order to achieve a moving formation, both avoiding collisions between agents and obstacles. The total agents' dynamics becomes:

- for the leader:

$$p_i^{k+1}=p_i^k-dt\left[\sum_{j\in N_i}\left(\frac{(\|p_i^k-p_j^k\|^2-d_{ij}^2)(p_i^k-p_j^k)}{\|p_i^k-p^{des}\|}-\frac{2(p_i^k-p_j^k)}{\|p_i^k-p_j^k\|^2}\right)-\frac{2(p_i^k-p^{obs})}{\|p_i^k-p^{obs}\|^2}+K_p(p_i^k-p^{des})\right]$$

- for the followers:

$$p_i^{k+1}=p_i^k-dt\left[\sum_{j\in N_i}\left(\frac{1}{\|p_i^k-p^{des}\|}(\|p_i^k-p_j^k\|^2-d_{ij}^2)(p_i^k-p_j^k)-\frac{2(p_i^k-p_j^k)}{\|p_i^k-p_j^k\|^2}\right)-\frac{2(p_i^k-p^{obs})}{\|p_i^k-p^{obs}\|^2}\right]$$

2.5.1 ROS2 implementation

In the script *the_agent_obstacle_avoidance.py* we implemented the whole control together. Thus, the Agent node includes all the previous parameters, plus the obstacle position, which has been located in different points according to the type of formation. In the `timer_callback` we modified the expression of the robots' dynamics as explained before. To launch its corresponding launching file *task2_obstacle_avoidance.launch.py* type:

```
$ export formation_shape=octahedron     (any desired shape)
$ export reference=trajectory     (desired target type)
$ ros2 launch task2 task2_obstacle_avoidance.launch.py
```

2.5.2 Results

From the simulation it can be seen that the agents try reaching the given formation while moving in the space in order for the leader the reach the goal pose/trajectory. Thanks to the weights given, we can see how at first, while the leader is still far from the goal, the formation burden will be less than the proportional controller and obstacle avoidance ones. For this reason we will see the agents moving quickly toward the goal without maintaining the formation, and accomplishing this one just after the leader is pretty near from the goal pose.

Conclusions

The aim of this project was to work on two main tasks: a Distributed Classification problem on the *fashion mnist* dataset and a Formation Control problem.
The setup of the former task was achieved by solving a consensus problem implementing a *Gradient Tracking* algorithm on a quadratic optimization function. Such procedure was then tested both on a cycle and on a star path graph with different number of agents and assigned *Metropolis-Hastings* weights. Then, we proceded by selecting the *sneakers* category as the one to be recognised among all the others and we preprocessed the dataset in order to make it suitable for a binary classification problem. As a last step before the distributed training, we built a multi-sample Neural Network with *sigmoid* function as activation function and a *Binary Cross-Entropy* loss.
Such NN is then used by different agents in parallel during the Distributed training. Each agent computes the training on its given subset of the dataset by solving a *distributed Gradient Tracking* optimization problem.
As for what it regards the second task, we implemented a discrete-time Formation Control algorithm in order to be able to make different number of agents reach different shapes in the 3D space. The Formation algorithm was then integrated with a logarithmic barrier function in order to implement the collision avoidance between the agents during the formation process. As a last step, we also implemented a leader-follower control law where one declared leader aims at reaching a desired position/trajectory in the space thanks to a proportional control law. As a consequence, all the other agents follow it by trying to maintain the formation given.
The results obtained for the different shapes are pretty good, in fact, for all of them we are able to reach formation in a matter of few seconds. Moreover, the implementation of the barrier function not only avoid the agents' collision but also speed up the formation process. In the end, its interesting to notice how, with the leader-follower control law, the formation error decreases more slowly as the followers are trying to maintain the formation while the leader moves. As an additional step, we also implemented a collision avoidance algorithm, by joining the former three separate steps in one, and defining an obstacle in the space.