UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Autonomous Systems

Course Project

Distributed Classification via Neural Networks and Formation Control

Professors:
Giuseppe Notarstefano
Ivano Notarnicola

Students: Alessandro Cecconi
Marco Bugo
Roman Sudin

Academic year 2021/2022

Abstract

In this report two different applications of distributed algorithms are discussed. The first one deals with the application of a distributed gradient tracking algorithm over a network of N agents, whose communication is represented by a weighted graph. The goal is to train N different Neural Networks, one for each agent, to solve a classification problem by recognizing a chosen digit from the MNIST dataset. Firstly, a single neural network has been modeled, then different agents have been added and the distributed gradient tracking has been implemented. The second one deals with a formation control problem where a set of agents, divided in leaders and followers, should perform a given set of layouts maintaining the given distances between each neighbor. In this case, the communication between agents is represented by a binomial undirected graph. At the beginning, a simple square formation has been implemented in python to study the mathematical quantities involved and to check the non-singularity of the chosen shape. Then, the results has been moved in ROS2 simulating the interaction between agents for different formations, either static and in motion, controlling them with different control inputs. To better visualize results the whole simulation has been implemented in RViz also. For this task we have used as reference [2].

Contents

List of Figures

List of Algorithms

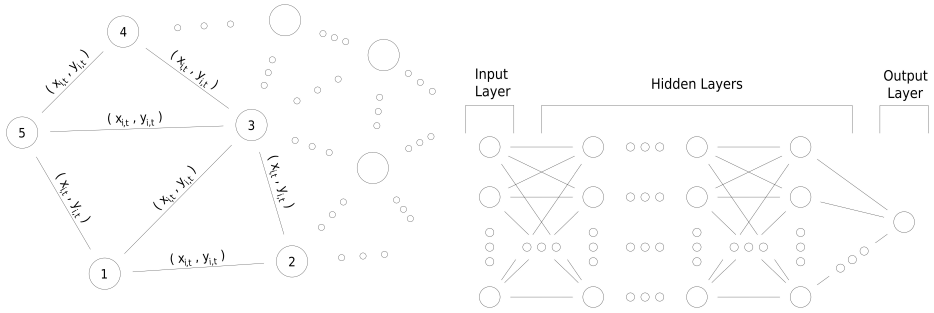
Chapter 1

Distributed gradient tracking for Neural Networks Training

1.1 Problem Statement

The goal is to properly define a network of N agents which cooperate in order to determine a classifier for a given set hand-written digits. Each agent i is equipped with a set of images of different digits and the associated labels coming from the MNIST dataset.
The problem has been solved by decoupling the task into two main objective:

- define the backpropagation algorithm of a single Neural Network from scratch, which is able to correctly classify the chosen digit;
- build a network of agents, each one with its own Neural Network and distribute the algorithm by means of the *Causual Distributed Gradient Tracking*.



(a) Scheme representing a hypothetical network of agents

(b) Diagram of the neural network used by each agent

1.2 Data manipulation

The dataset, downloaded from Keras MNIST, has been modified in order to use it in the neural network. The origin dataset is composed by images [28pixel $*$ 28pixel] of handwritten numbers with the associated label that indicates the written number. Firstly, since each value of the images ranges between $[0, 255]$, it has been normalized between $[0, 1]$ in order to not saturate the activation functions of the neurons. Secondly, the image matrix of size $(28, 28)$ has been reshaped as a unit vector of size $(784, 1)$ to match the dimension of the input of the neural network. Finally, given the desired number that should be properly classified by the

neural network, each label of the dataset has been changed as follows

$$y_i = \begin{cases} 1 & \text{if } label = digit \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

1.3 Data balancing

In order to have a training set that is able to train efficiently a machine learning model is useful to have a balanced dataset. To balance the training data, the class of *wrong* digits has been undersampled to make the two classes of the same order of magnitude. This passage is crucial to perform a good training of the model, otherwise it would reach high accuracy by classifying everything as a wrong digit instead of really trying to guess the chosen number.

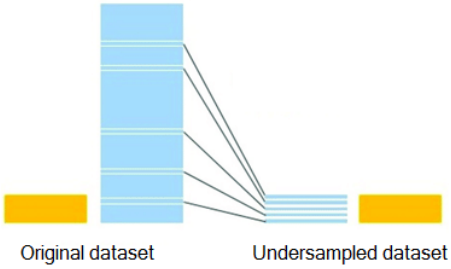


Figure 1.2: Graphical representation of undersampling

1.4 Neural Network Structure

The Neural Network has the same number of neurons for each layer, because using *numpy* arrays to store the variables it was not possible to change the dimensions once initialized at the beginning of the training. However, the task asks to classify a given input data, so the network dynamics has been modified to take it into account. In the last layer only the first neuron is trained, whereas the others are kept fixed at zero and unmodified. In order to do that, the inference and the adjoint dynamics functions in Python have been modified to force the wanted behaviour on the last layer.
For the initialization of the weights a random value has been assigned, because the results, obtained by initializing them at zero, have shown no particular differences.
The elementary unit of the neural network is a neuron, which is a computation unit whose output value is calculated with the following relation

$$x_l^+ = \sigma(x^T u_l) + u_{l,0} \tag{1.2}$$

where $x_l^+ \in \mathbb{R}$ is the output of the neuron, $x \in \mathbb{R}^d$ is the output of the previous layer, $u_l \in \mathbb{R}^d$ are the weights of that neuron and $u_{l,0} \in \mathbb{R}$ is the bias. With $\sigma(\cdot) : \mathbb{R}^{d+1} \to \mathbb{R}$ is the activation function. The implementation of a stand alone network has been really helpful to understand how does it works and to pick the activation and loss function which give best result in this case of study. We will say more about that in the next sections.
In 1 is shown the backpropagation algorithm that we have used and properly modified for our purposes

Algorithm 1 Backpropagation for NN training

for $k = 0, 1, 2, \dots$ **do**
  for image $i = 1, \dots, \mathcal{I}$ **do**
    Backward Simulation of the costate equation:
    for layer $t = T-1, \dots, 0$ **do**

$$\lambda_{i,t} = A_{i,t}^{k\,T} \lambda_{i,t+1}, \quad \lambda_{i,t} = \nabla J(x_{i,T}^k; y^i)$$
$$\Delta u_{i,t}^k = B_{i,t}^{k\,T} \lambda_{i,t+1}$$

      with $A_{i,t}^k = \nabla_x f(x_{i,t}^k, u_t^k)^T$ and $B_{i,t}^k = \nabla_u f(x_{i,t}^k, u_t^k)^T$
      Descent step on the control input:
    end for
    for layer $t = 0, \dots, T-1$ **do**

$$u_t^{k+1} = u_t^k - \alpha_k \sum_{i=1}^{\mathcal{I}} \Delta u_{i,t}^k$$

      Forward Simulation of the NN dynamics:
    end for
    for $t = 0, \dots, T-1$ **do**

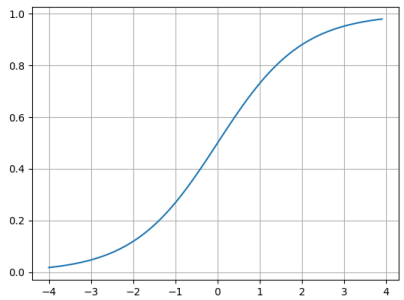
$$x_{i,t+1}^{k+1} = f(x_{i,t}^{k+1}, u_t^{k+1,})$$

    end for
  end for
end for

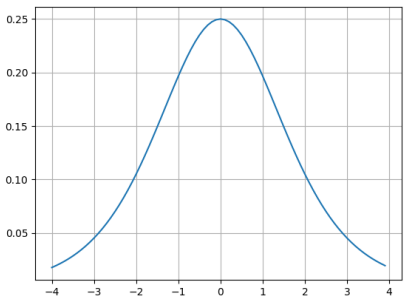
1.5 Activation function

For what concern the activation functions, several of them have been tested and implemented in Python to compare their behaviours. In particular, the following activation functions have been tested: *ReLu*, tanh and Sigmoid. The tests have shown that the activation function which performs better without return any numerical error, was the Sigmoid. The function and its derivative is shown in figures 1.3a and 1.3b, and it is defined as follows

$$S(z) = \frac{1}{1+e^{-z}} \quad \frac{d}{dz} S(z) = S(z)(1 - S(z)) \tag{1.3}$$



(a) Sigmoid



(b) Sigmoid derivative

1.6 Cost Function

Two different cost functions have been used to train the neural network and then the results have been compared. The first one used is the *Mean Squared Error* (MSE) Function

$$J_{MSE} = \sum_{k=1}^{K} ||\phi(u; D^k) - y^k||^2 \tag{1.4}$$

where $u \in \mathbb{N}^{l-1} \times \mathbb{N}^d \times \mathbb{R}^{d+1}$ is a tensor containing the weigths of the neural network, $D^k \in \mathbb{R}^{784}$ is the input data and $y^k \in \mathbb{N}$ is the lable associated to that input data. The prediction of the category of the input image is done by $\phi(\cdot,\cdot) : \mathbb{N}^{l-1}\ \mathbb{N}^d \times \mathbb{R}^{d+1} \times \mathbb{R}^d \to [0,1] \in \mathbb{R}$. The number of images used is denoted with $K \in \mathbb{N}$.
The second one is the *Binary Cross Entropy* (BCE) function

$$J_{BCE} = \sum_{k=1}^{K} y^k \log\left(\phi(u; D^k)\right) + (1 - y^k) \log\left(1 - \phi(u; D^k)\right) \tag{1.5}$$

The cost function is evaluated at each epoch in order to understand if the Neural Network is able to properly classify our input data.

1.7 Final structure

After several experiments the number of layers has been changed and in figure 1.4. Finally, the standalone neural network has been trained and tested reaching a really good accuracy on a balanced set (up to 98 %), that means that the standalone neural network is able to properly assign the chosen digit. This passage has taught how the neural network behaviour before implementing it in a distributed algorithm.

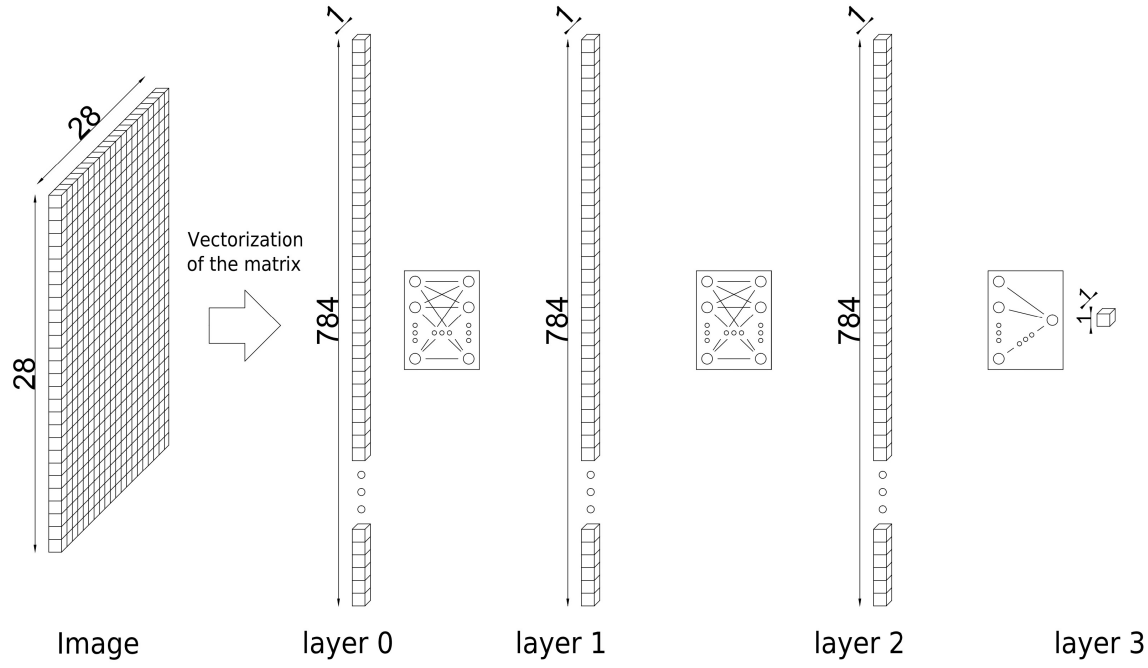


Figure 1.4: Neural network internal structure.

1.8 Consensus algorithm

After the testing of the neural network the Causal Distributed Gradient Tracking has been implemented. To implement the algorithm a connected graph has been created to define for each agent the neighbours with which share information. In particular, in figure 1.5 is shown the distributed graph with four agents connected through bidirectional edges. Each node has its own neural network to train on the basis of its own computation and on the information from the neighbours. In particular the nodes share each other computed gradient to optimize the learning process.
For what concern the dataset by which each agent trains its own neural network, it has been divided randomly between the agents such that everyone has the same number but different images to train and test. Through the division of the dataset the capability of the conseunsus algorithm is appreciated because, even if every agent has different images, they reach the same results for what concern the weights of neural network.

Algorithm 2 Causal Distributed Gradient Tracking for NN training

for epoch $k = 0, 1, 2, \dots$ **do**
  for agent $i = 1, \dots, N$ **do**
    for image $z = 1, \dots, \mathcal{I}$ **do**
      Backward simulation of the NN:
      for layer $t = T-1, \dots, 0$ **do**

$$\lambda^i_{z,t} = (A^{i,k}_{z,t})^\top \lambda^i_{z,t+1}, \quad \lambda^i_{z,t} = \nabla J(x^{i,k}_{z,T}; y^{i,z})$$
$$\Delta u^{i,k}_{z,t} = (B^{i,k}_{z,t})^\top \lambda^i_{z,t+1}$$

      end for
    end for
    Causal Gradient Tracking:

$$\mathbf{u}_i^{k+1} = \textstyle\sum_{j\in\mathcal{N}_i} a_{ij}\mathbf{u}_j^k + \mathbf{z}_i^k - \alpha\Delta\mathbf{u}_i^k$$
$$\mathbf{z}_i^{k+1} = \textstyle\sum_{j\in\mathcal{N}_i} a_{ij}\mathbf{z}_j^k - \alpha(\sum_{j\in\mathcal{N}_i} a_{ij}\mathbf{u}_j^k - \mathbf{u}_i^k)$$

  end for
  Forward Simulation of the NN:

$$\mathbf{x}_i^{k+1} = \mathbf{f}(\mathbf{x}_i^{k+1}, \mathbf{u}_i^{k+1})$$

end for

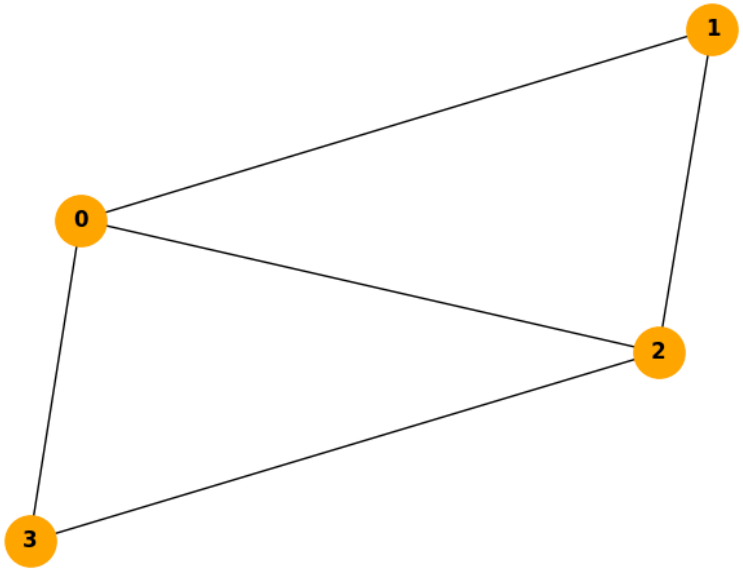


Figure 1.5: Graph for Distributed NN training with 4 agents

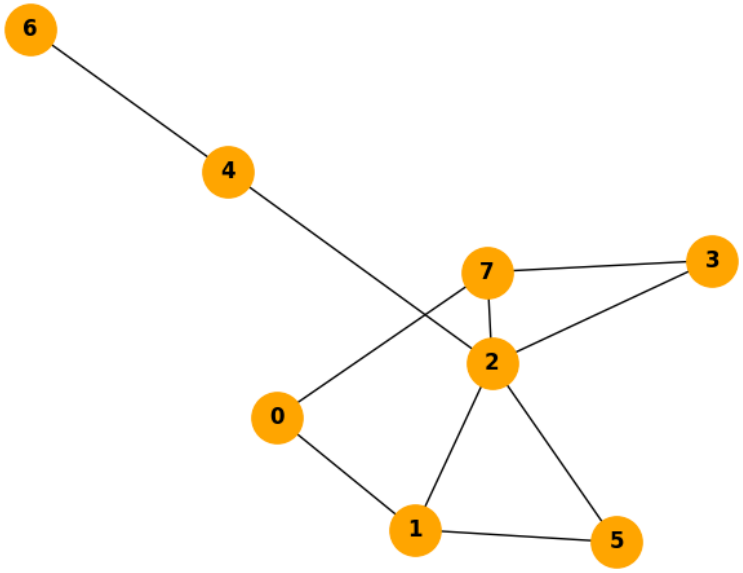


Figure 1.6: Graph for Distributed NN training with 8 agents

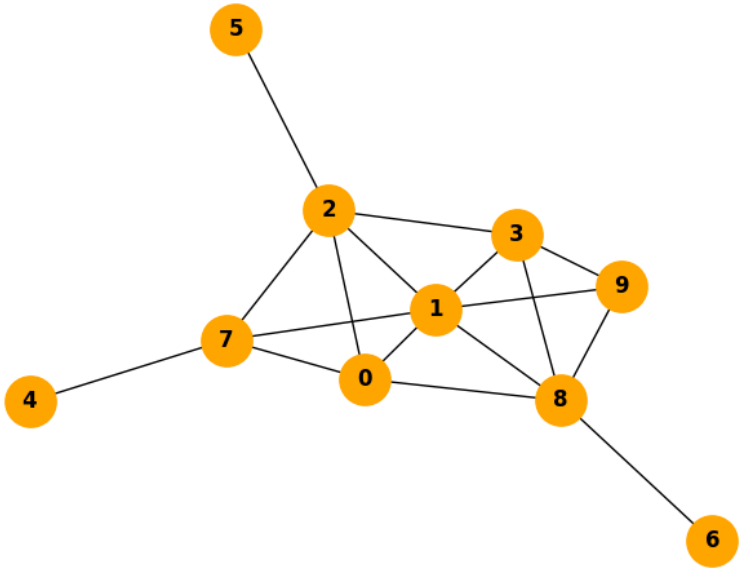


Figure 1.7: Graph for Distributed NN training with 10 agents

1.9 Results

Four main tests have been plotted with the parameters in table 1.1. From all the test good results has been produced with some peculiar differences. For each test it has been plotted: configuration of the connected graph (figures: 1.5, 1.6 and 1.7),sum of the Cost Function (figures: 1.8, 1.11, 1.14 and 1.17) and sum of the gradient (figures: 1.9, 1.9, 1.15 and 1.18). Firstly, surprisingly the distributed algorithm gives better results in terms of accuracy with respect to the stand alone neural network. Secondly, it has been noticed that the BCE is more sensitive with respect to the stepsize and the number of images for each agent. At the beginning the algorithm does not converge, but at the end it gives better results with respect to the MSE case. This is not surprising since the BCE is better suited for classification tasks. Finally, to see if the nodes weights reach consensus, for each node four random weights of four random nodes has been picked and plotted together. The nodes starting from random values show converge after some iterations when they reach the same value. For every case study the consensus has been reached, as can be appreciated in figures 1.10, 1.13, 1.16 and 1.19.
Below is shown an output example of a node after the training and then the test of the neural network

```
AGENT 1
The accuracy is 92.0 % where:
False positives 2
False negatives 0
LuckyNumber identified correctly 15 over 15
NOT LuckyNumber identified correctly 8 over 10
The effective LuckyNumbers in the tests are: 15
```

Cost Function	Agents	Images	Epochs	Stepsize	Layers
MSE	4	50	150	0.01	4
BCE	4	50	200	0.01	4
MSE	8	25	300	0.01	4
BCE	10	25	250	0.01	4

Table 1.1: Parameters

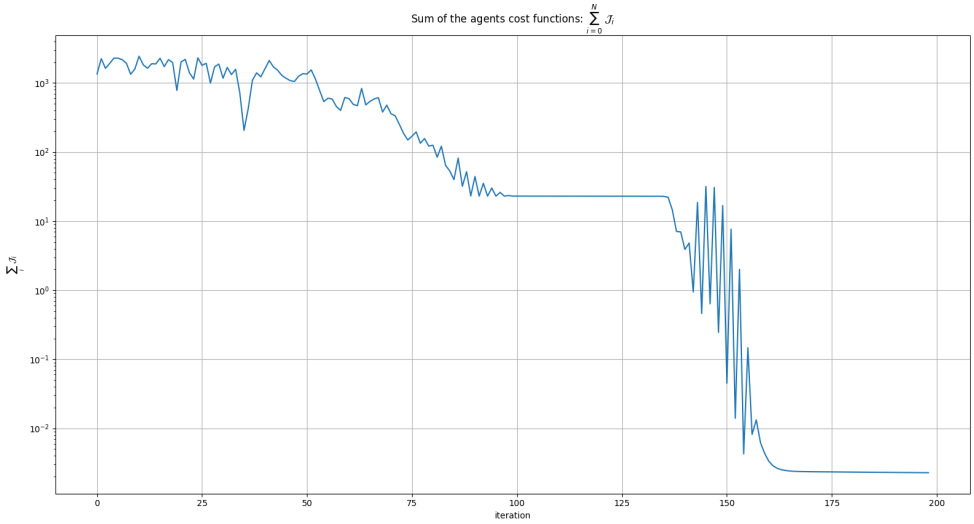


Figure 1.8: Total cost function $\sum_{i=1}^{N} J_i(\cdot)$ - BCE with 4 agents

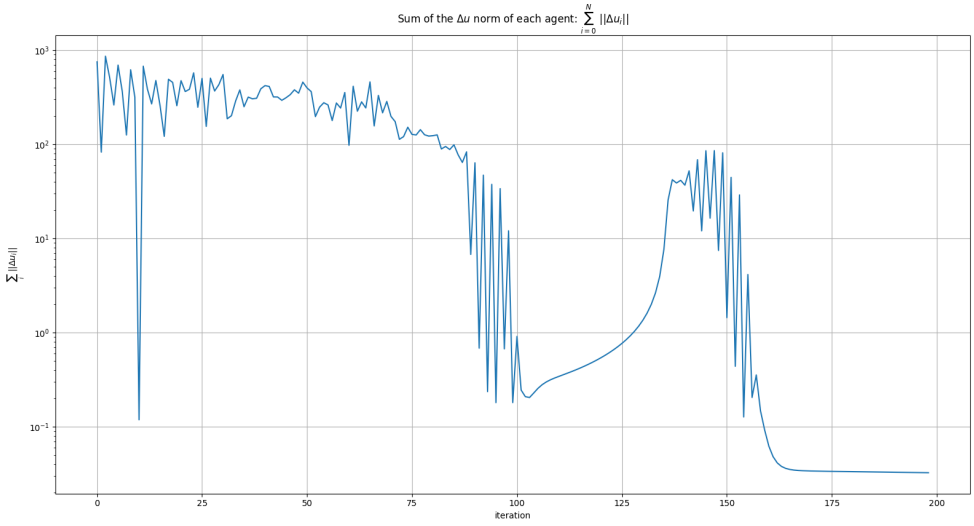


Figure 1.9: $\sum_{i=0}^{N} \Delta u_i$ - BCE with 4 agents

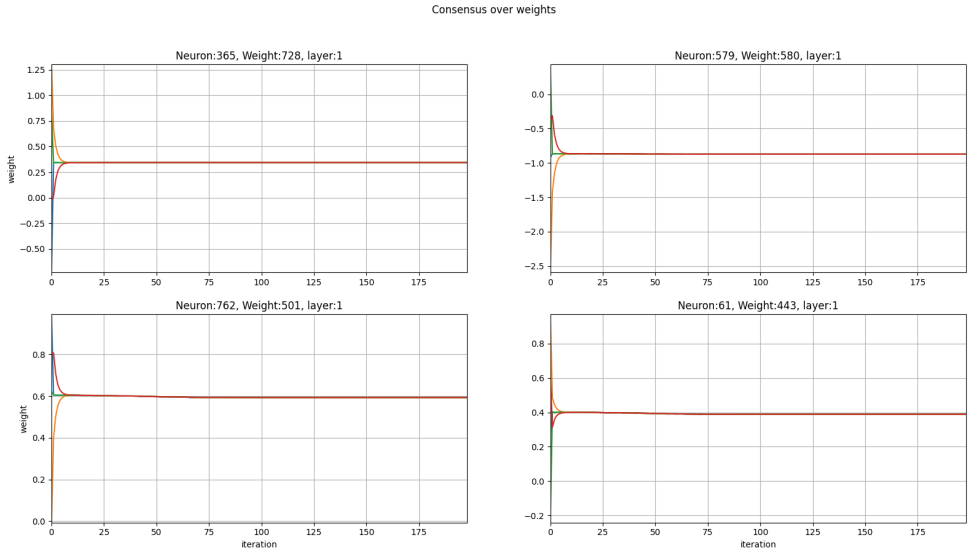


Figure 1.10: Neural Networks weights consensus - BCE with 4 agents

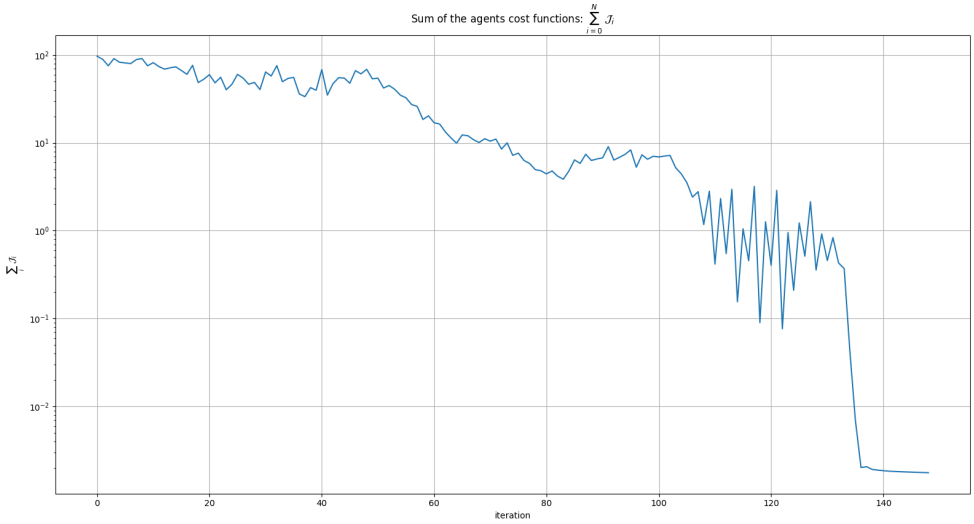


Figure 1.11: Total cost function $\sum_{i=1}^{N} J_i(\cdot)$ - MSE with 4 agents

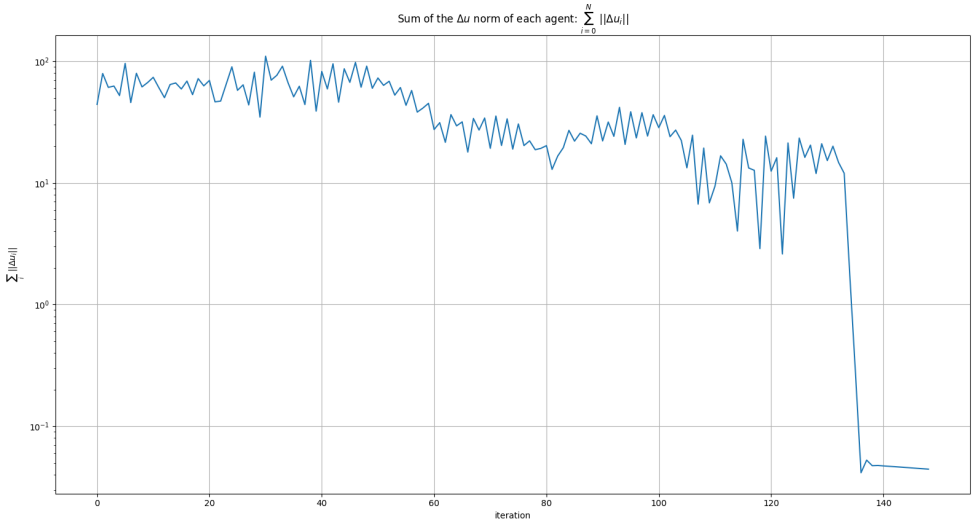


Figure 1.12: Sum of $\sum_{i=0}^{N} \Delta u_i$ - MSE with 4 agents

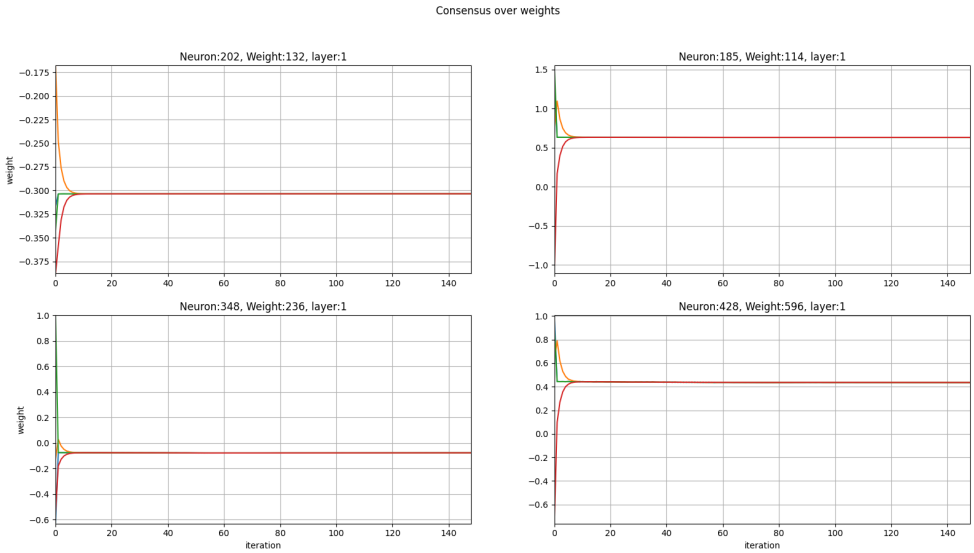


Figure 1.13: Neural Networks weights consensus - MSE with 4 agents

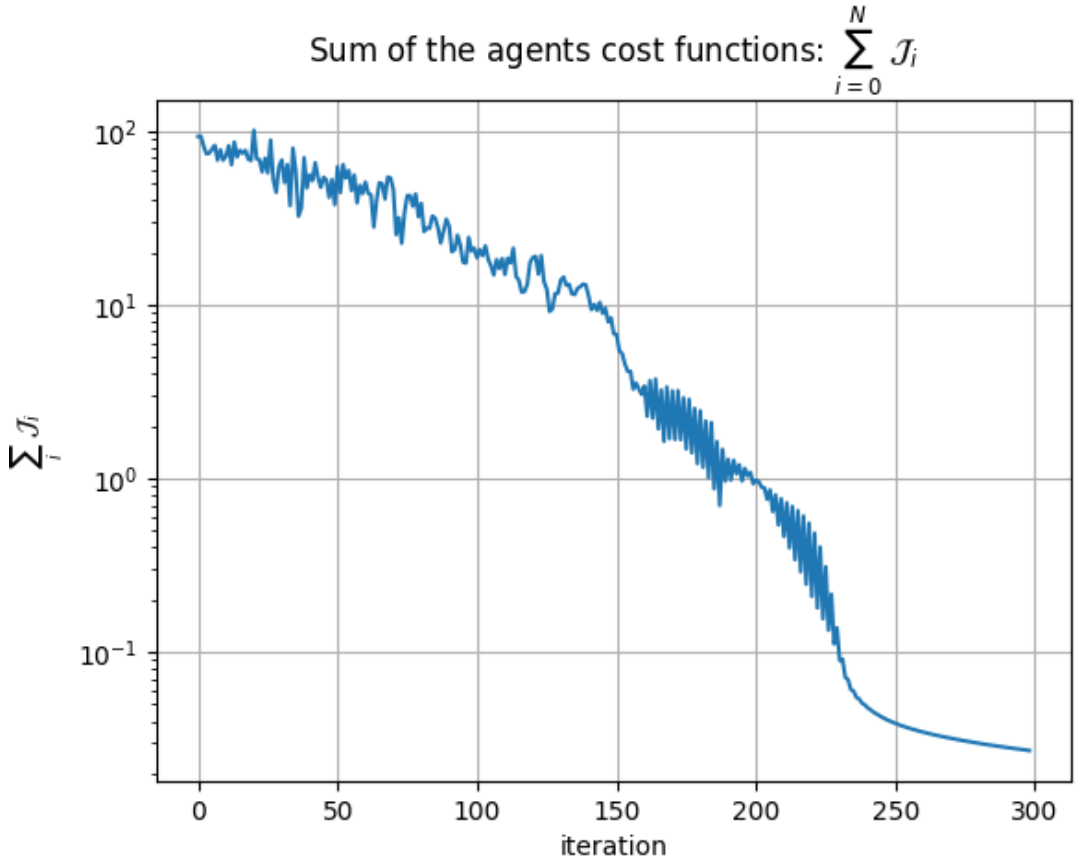


Figure 1.14: Total cost function $\sum_{i=1}^{N} J_i(\cdot)$ - MSE with 8 agents

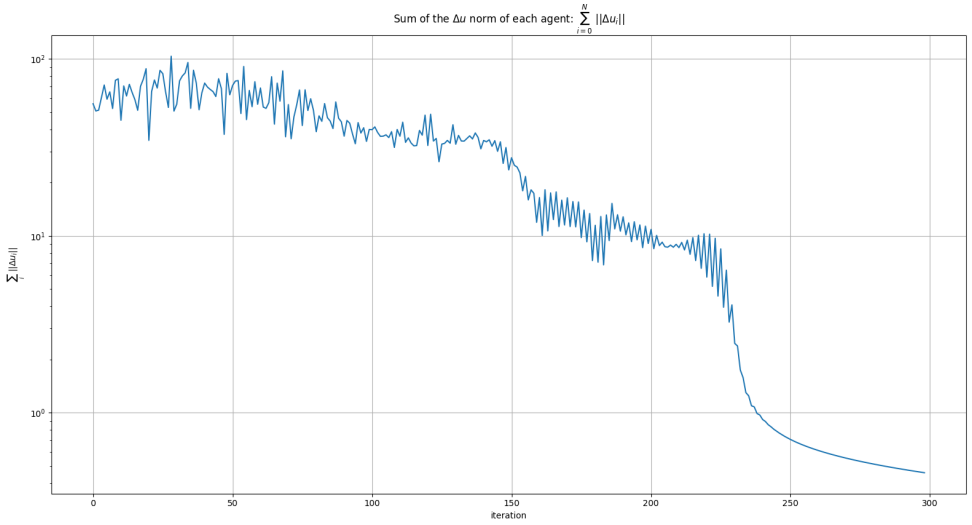


Figure 1.15: $\sum_{i=0}^{N} \Delta u_i$ - MSE with 8 agents

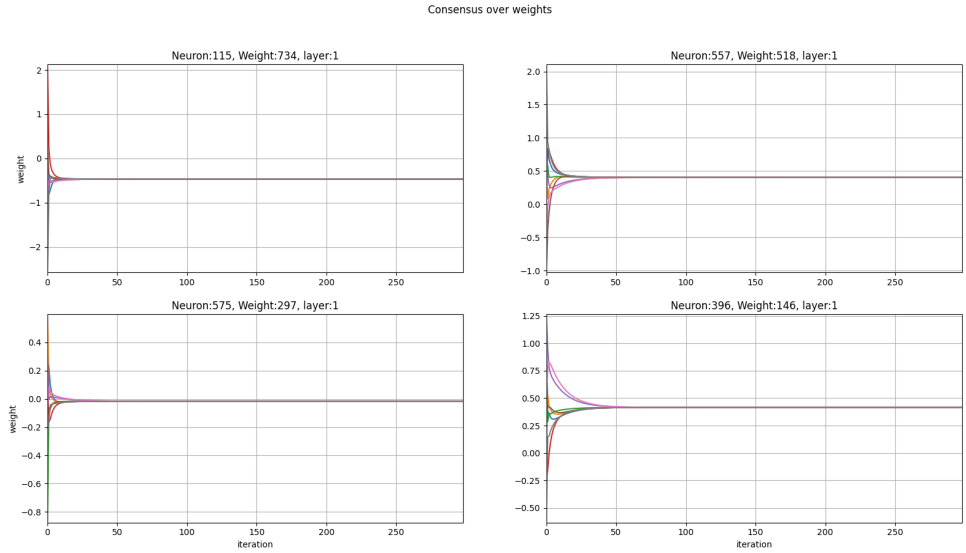


Figure 1.16: Neural Networks weights consensus - MSE with 8 agents

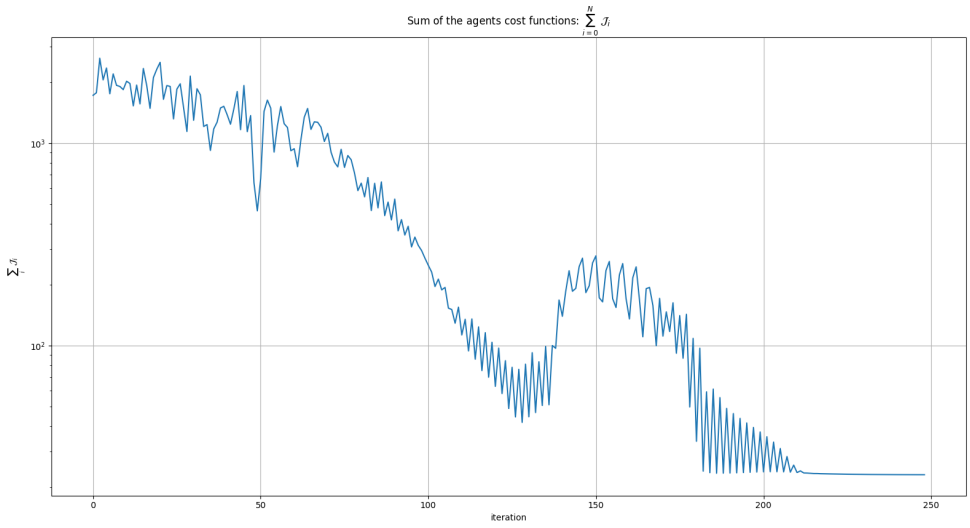


Figure 1.17: Total cost function $\sum_{i=1}^{N} J_i(\cdot)$ - BCE with 10 agents

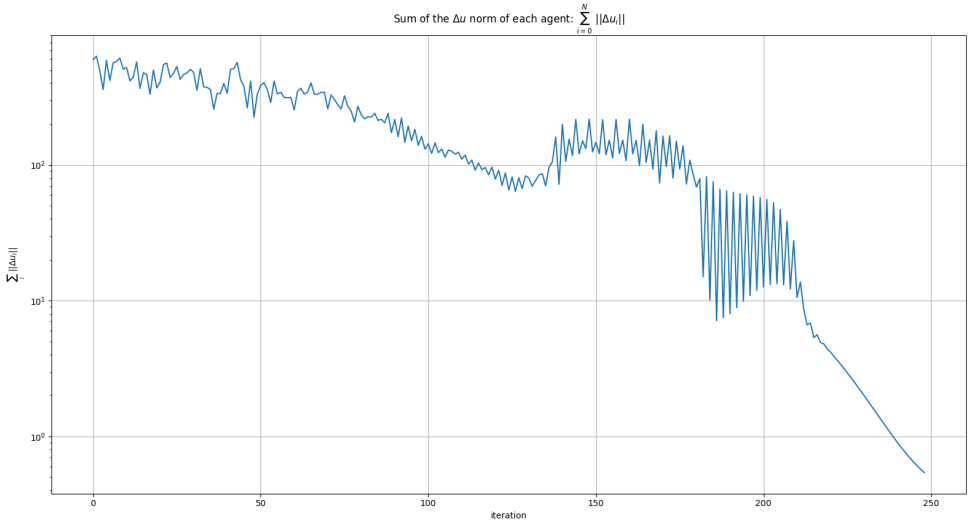


Figure 1.18: $\sum_{i=0}^{N} \Delta u_i$ - BCE with 10 agents

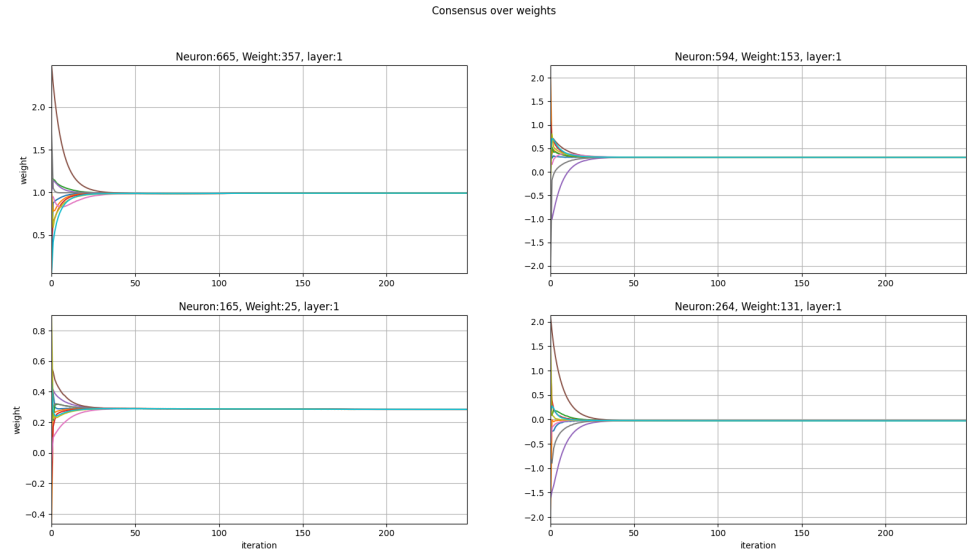


Figure 1.19: Neural Networks weights consensus - BCE with 10 agents

Chapter 2

Formation Control in ROS2

2.1 Problem Statement

In this task a network of N robotic agents has been considered, whose state vector has the structure

$$x_i(t) = \begin{bmatrix} p_i(t) \\ v_i(t) \end{bmatrix} \tag{2.1}$$

with $i = 1, \ldots, N$, $p_i(t) \in \mathbb{R}^d$ and $v_i(t) \in \mathbb{R}^d$. Two kind of robots have been considered: leaders and followers. The motion (i.e. position and velocity) of each leader is given a *a priori*, instead each follower is modeled as a double-integrator dynamics

$$\begin{aligned} \dot{p_i}(t) &= v_i(t) \\ \dot{v_i}(t) &= u_i(t) \end{aligned} \tag{2.2}$$

with $i \in \mathcal{V}_f$, where $\mathcal{V}_f$ is the set of followers.
The information flows among agents by a fixed undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. By mapping the position p_i to vertex i, the formation is denoted as $\mathcal{G}(p)$. If an edge between agent i and agent j exists, agent i can access the information of agent j (i.e. $(i, j) \in \mathcal{E}$).
To apply the *Bearing-Based Formation Maneuver Control* the following quantity has been defined

$$g_{ij} \triangleq \frac{p_j - p_i}{||p_j - p_i||} \tag{2.3}$$

which is called the *bearing unit vector* of agent j with respect to agent i. Once g_{ij} has been calculated, it has been used to define the *Orthogonal Projection Matrix* $P_{g_{ij}}$ as

$$P_{g_{ij}} = I_d - g_{ij} g_{ij}^T \tag{2.4}$$

This matrix geometrically projects any vector onto the orthogonal compliment of $P_{g_{ij}}$. Based on these quantities, it is possible to formally state the properties of the target formation, starting from the *Bearing Laplacian Matrix* $\mathcal{B}(\mathcal{G}(p^*))$

$$[\mathcal{B}(\mathcal{G}(p^*))]_{ij} = \begin{cases} \mathbf{0}_{dxd}, & i \neq j, (i,j) \notin \mathcal{E} \\ -P_{g_{ij}^*}, & i = j, (i,j) \in \mathcal{E} \\ \sum_{k \in \mathcal{N}_i} P_{g_{ik}^*}, & i = j, i \in \mathcal{V} \end{cases} \tag{2.5}$$

which represent each ij block shape, with $\mathcal{B}(\mathcal{G}(p^*)) \in \mathbb{R}^{Nd \cdot Nd}$. A function for each of these quantities 2.3, 2.4 and 2.5 has been defined in a python file called *Functions.py* which has been used throughout the entire task. In particular, a tensor which stores every matrix $P_{g_{ij}}$ from any node i to any other node j (with $i \neq j$) has been defined, to make easier the computation

of the control action that will be discussed in the next section.
Based on the Bearing Laplacian $\mathcal{B}(\cdot)$, it is possible analyze the existence and uniqueness of the target formation, using the result to check if the generated graph have the right shape to model the formation we want to follow. In fact, the matrix $\mathcal{B}(\cdot)$ can be partitioned in

$$\mathcal{B} = \begin{bmatrix} \mathcal{B}_{ll} & \mathcal{B}_{lf} \\ \mathcal{B}_{fl} & \mathcal{B}_{ff} \end{bmatrix}$$

It can be shown that the formation is unique if and only if the sub-matrix $\mathcal{B}_{ff}$ is not singular. Under this condition, it is possible to uniquely determine position of velocity of the followers in the target formation

$$p_f^*(t) = -\mathcal{B}_{ff}^{-1}\mathcal{B}_{fl}p_l(t), \quad v_f^*(t) = -\mathcal{B}_{ff}^{-1}\mathcal{B}_{fl}v_l(t)$$

This condition is always checked when the launch file is called, to be sure that is fulfilled.

2.2 Followers Control Inputs

The goal is to control agents translation and scale following a desired path while maintaining the given formation pattern. Leaders move with constant velocity profiles (i.e. also zero) or with a piece-wise continuous acceleration profile.
For what concern the followers,different cases have been taken into account and will be discussed in details in the next sections.
If the leaders have constant velocity the control input will be

$$u_i(t) = -\sum_{j\in\mathcal{N}_i} P^*_{g_{ij}}[k_p(p_i(t) - p_j(t)) + k_v(v_i(t) - v_j(t))] \tag{2.6}$$

where $\mathcal{N}_i$ represent the set of neighbors of the agent i and k_p, k_v are strictly positive constant gains. As mentioned before, this control action is applied if and only if the agent is a follower.
In the case of time-varying leaders velocity, the control action applied to the followers is

$$u_i(t) = -K_i^{-1}\sum_{j\in\mathcal{N}_i} P^*_{g_{ij}}[k_p(p_i(t) - p_j(t)) + k_v(v_i(t) - v_j(t)) - \dot{v}_j(t)] \tag{2.7}$$

where $K_i = \sum_{j\in\mathcal{N}_i} P^*_{g_{ij}}\ \forall i \in \mathcal{V}_f$ is a non-singular matrix if the formation is unique. A term which takes into account the neighbors' velocity should be added, but since this information is not available from the system state,it has been calculated numerically without enlarging the state dimension. In order to do that, in the *Functions.py* python file has been defined a function which calculates the numerical derivative as

$$\dot{v_j}(t) \simeq \frac{v_{j,t} - v_{j,t-1}}{dt}$$

as suggested in [1] in chapter III. Each followers' neighbors velocity at the current iteration t is stored, that it will be used at $t+1$ to calculate the approximate derivative.
In the last case, a constant input disturbance action w_i acting on each followers has been supposed. This action modifies the dynamics in 2.2 as

$$\begin{aligned} \dot{p}_i(t) &= v_i(t) \\ \dot{v}_i(t) &= u_i(t) + w_i \end{aligned} \tag{2.8}$$

Under this new dynamics the control action becomes

$$u_i(t) = -\sum_{j\in\mathcal{N}_i} P^*_{g_{ij}}\left[k_p(p_i(t) - p_j(t)) + k_v(v_i(t) - v_j(t)) - k_i\int_0^t (p_i(\tau) - p_j(\tau))d\tau\right] \tag{2.9}$$

The integral action has been introduced by discretizing the position error using the following rule

$$e_{p,t} = \begin{bmatrix} e_{p_x,t-1} \\ e_{p_y,t-1} \end{bmatrix} + \begin{bmatrix} e_{p_x,t} \\ e_{p_y,t} \end{bmatrix} \cdot dt$$

making a sort of numerical integration. As done before for velocities, all the position errors for each pair of agents at every iteration have been stored in a tensor to use them in the next iteration.
The same reasoning has been applied for time-varying moving leaders even in the case of constant disturbances acting on followers input, by just adding the integral term in 2.7.

2.3 Leaders acceleration profile and Formation Definition

Leaders' positions and accelerations are assumed to be known *a priori* as stated in the reference paper [2]. To define the leaders' motion, a fifth order polynomial trajectories generated by the *robotics-toolbox-python* has been used in order to satisfy the condition of piece-wise continuous acceleration. The equation which defines the polynomial is

$$\mathbf{q}(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{2.10}$$

Since the polynomial order is of degree 5, the boundary conditions necessary are at least 6, in particular $\mathbf{p}(t_{in}) = \mathbf{v}(t_{in}) = \mathbf{a}(t_{in}) = \mathbf{v}(t_f) = \mathbf{a}(t_f) = \mathbf{0}$ and $\mathbf{p}(t_f) = \mathbf{10}$, where $t_{in} = 0$. For sake of simplicity, the same conditions are applied for the motion along x and y. In figure 2.1 an example of trajectory computed, from which the $\ddot{\mathbf{q}}(t)$ is provided as input to the leaders.
For what concern the formation definition, the shapes have been defined by using *GeoGebra*. Some of them are shown in figure 2.2 and 2.3. After defining the shape, an arbitrary number of points have been chosen on it, representing the number of agents used in the simulation environment. Then, all the reference positions $\mathbf{p}^* = \begin{pmatrix} p_x^* \\ p_y^* \end{pmatrix}$ have been stored in a python dictionary as follows

```
formations = {'square': [[1, 5], [5, 5], [1, 1], [5, 1]]}
```

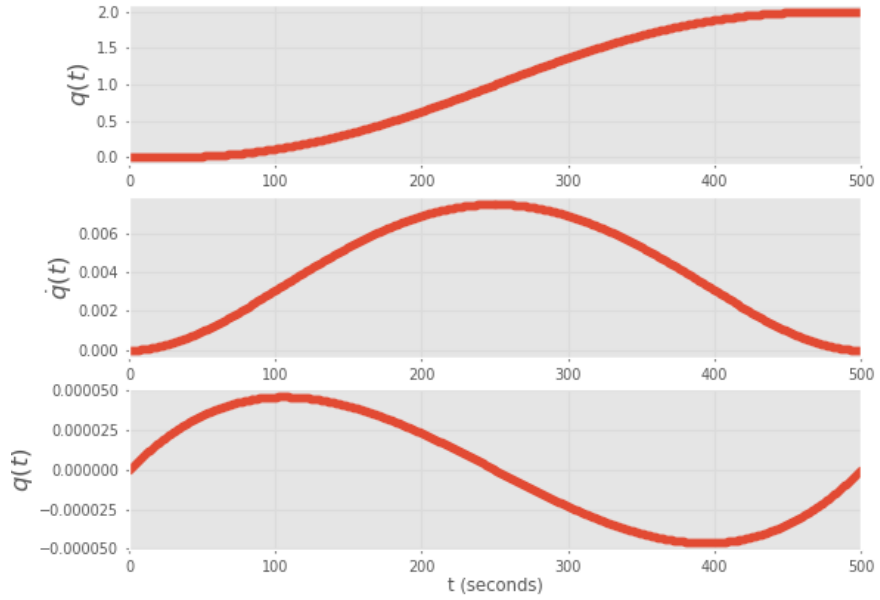
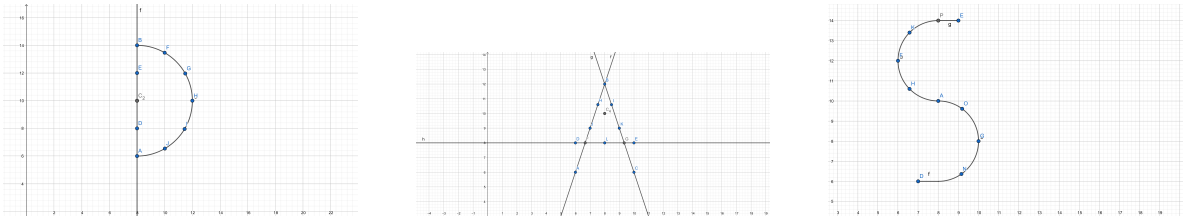
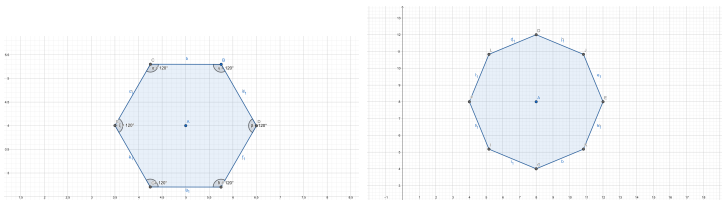


Figure 2.1: Sample of $q(t)$, $\dot{q}(t)$ and $\ddot{q}(t)$ generated using the fifth-order polynomial function



(a) D letter in GeoGebra (b) A letter in GeoGebra (c) S letter in GeoGebra

Figure 2.2: Letters for time-varying formation built in GeoGebra



(a) Exagon (b) Octagon

Figure 2.3: Formations shape in GeoGebra

2.4 ROS2 Implementation and Results

The whole algorithm has been implemented using the *ROS*2 environment exploiting its distributed property where a node and a topic for each agent can be executed. Every agent is automatically launched by a launch file. The launch file computes all the necessary information needed for the nodes (e.g desired formation, acceleration profile, $P_{g_{ij}}$, and so on) then launch each node with the necessary arguments.

Each node executes the file *agent.py* for what concern the still and moving formation of generic shapes (square, octagon, group) and the file *agent_lett.py* for what concern the letters formation task. These files manage the communication between nodes, and the dynamic of the nodes distinguish the leaders from the followers.
As already said, before the launch of each node the non-uniqueness of the sub-matrix $\mathcal{B}_{ff}$, otherwise the graph is generated again. The launch files have been designed in order to be as parametric and reusable as possible, for example it has been defined a launch file capable to: launch different formations, define static or moving leaders, zero or random initial conditions and enable or disable an extra integral action. The agent dynamics is applied by following the forward-Euler discretization for each agent state

$$x_{i,t+1} = x_{i,t} + dt \cdot \dot{x}_i(t) \tag{2.11}$$

where dt is the sampling time.

2.4.1 Formations with constant leader velocity

Several formations with different number of agents have been defined. Leaders are initialized in their reference positions, instead the followers need to reach the target formation.
For the follower initialization two different cases have been implemented: a zero initial position and a random initial position. In both cases the followers are able to converge to their references, in figures 2.4 and 2.11 some of the results are shown. Each followers implement the control action defined in 2.6. The results point out that the convergence of the algorithm is strictly related to the number of leaders and to the graph connectivity.

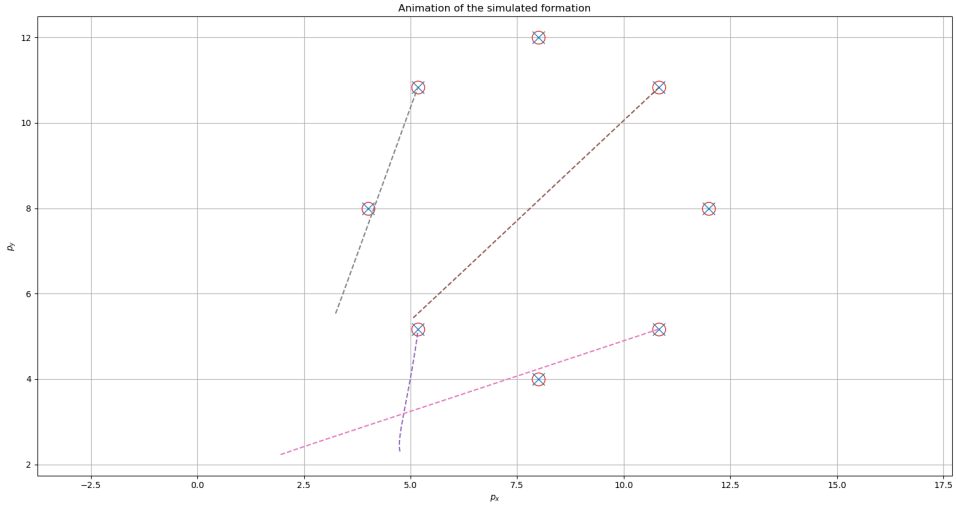


Figure 2.4: Simulation result of octagon formation - 4 leaders (still)

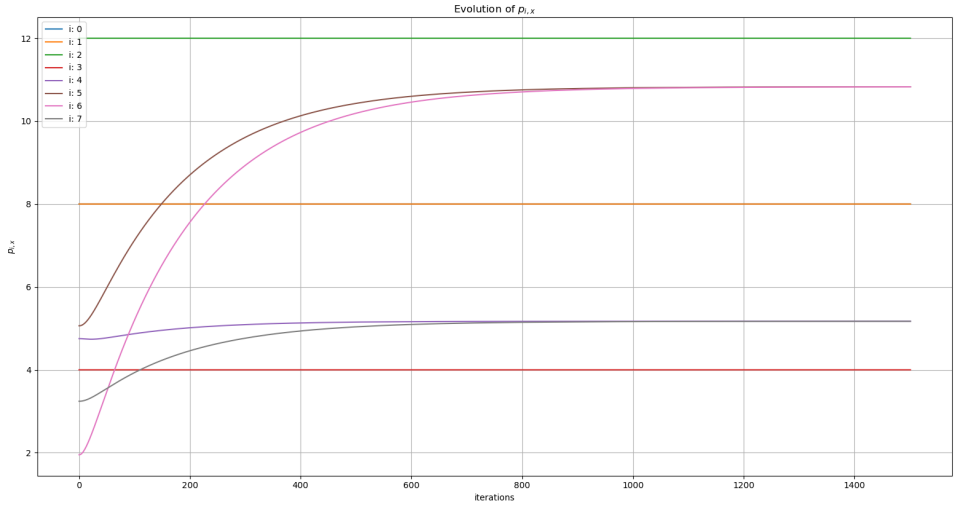


Figure 2.5: Evolution of $p_{i,x}$ during octagon formation - 4 leaders (still)

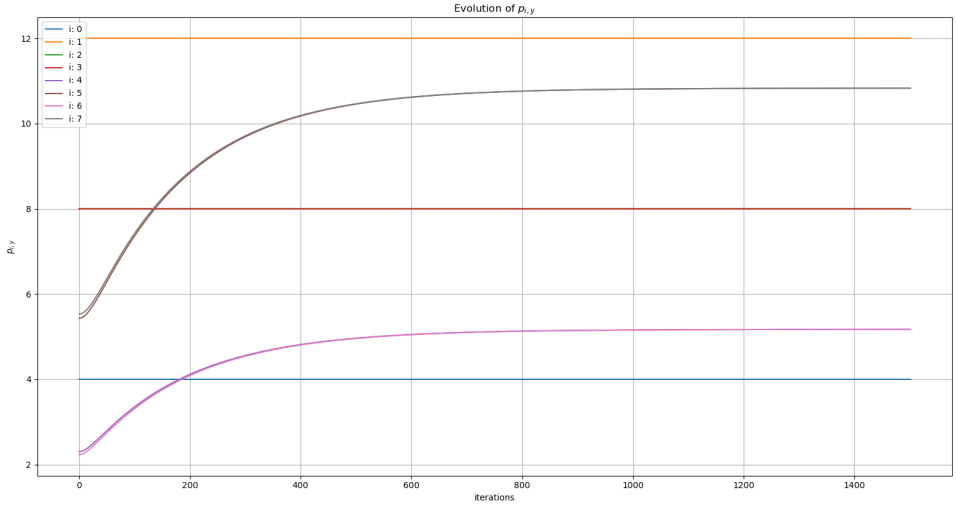


Figure 2.6: Evolution of $p_{i,y}$ during octagon formation - 4 leaders (still)

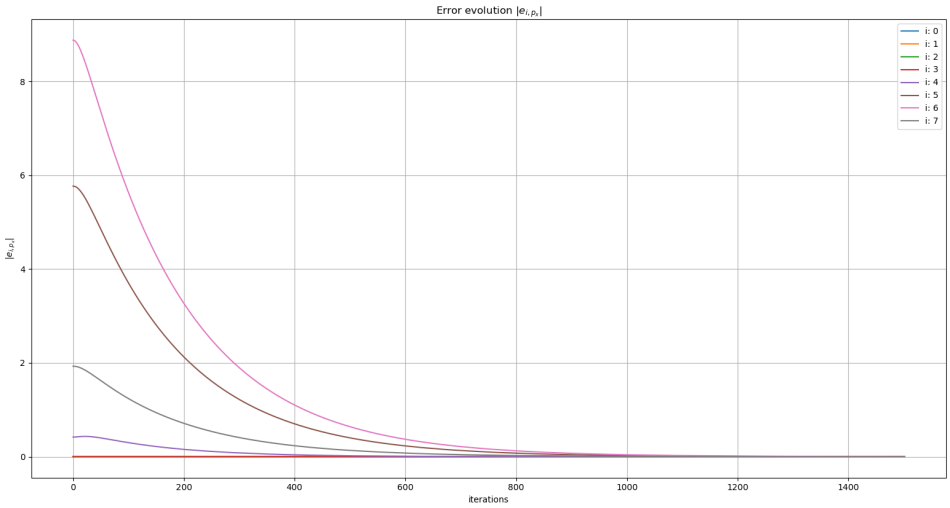


Figure 2.7: Evolution of $|e_{i,p_x}| = |p_{i,x} - p^*_{i,x}|$ during octagon formation - 4 leaders (still)

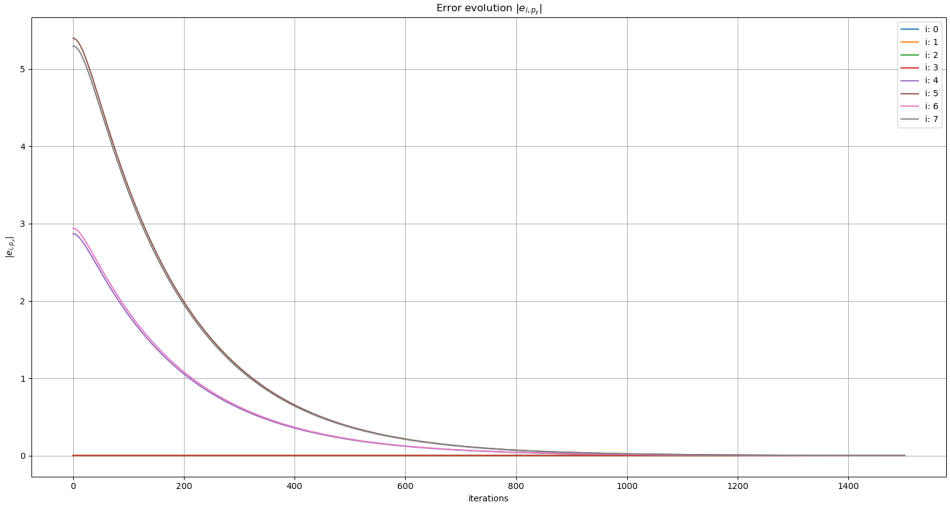


Figure 2.8: Evolution of $|e_{i,p_y}| = |p_{i,y} - p^*_{i,y}|$ during octagon formation - 4 leaders (still)

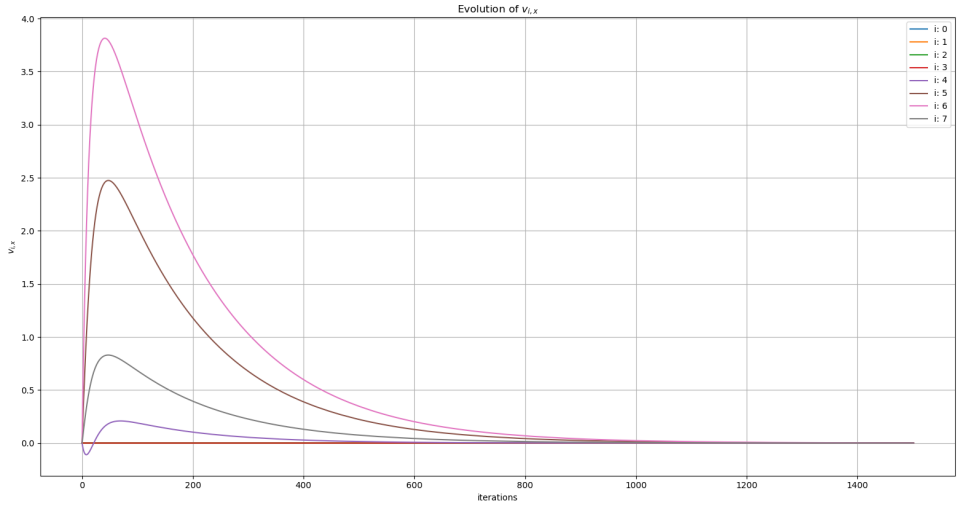


Figure 2.9: Evolution of $v_{i,x}$ during octagon formation - 4 leaders (still)

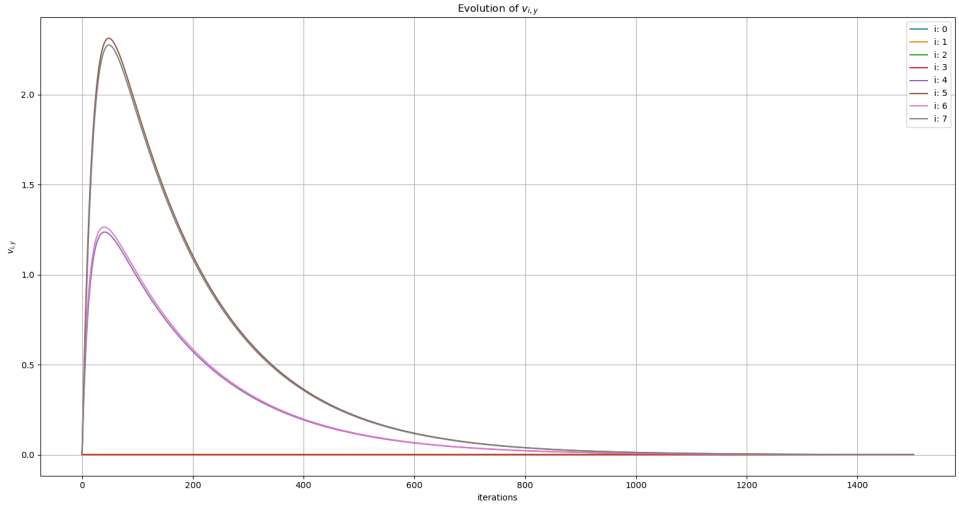


Figure 2.10: Evolution of $v_{i,y}$ during octagon formation - 4 leaders (still)

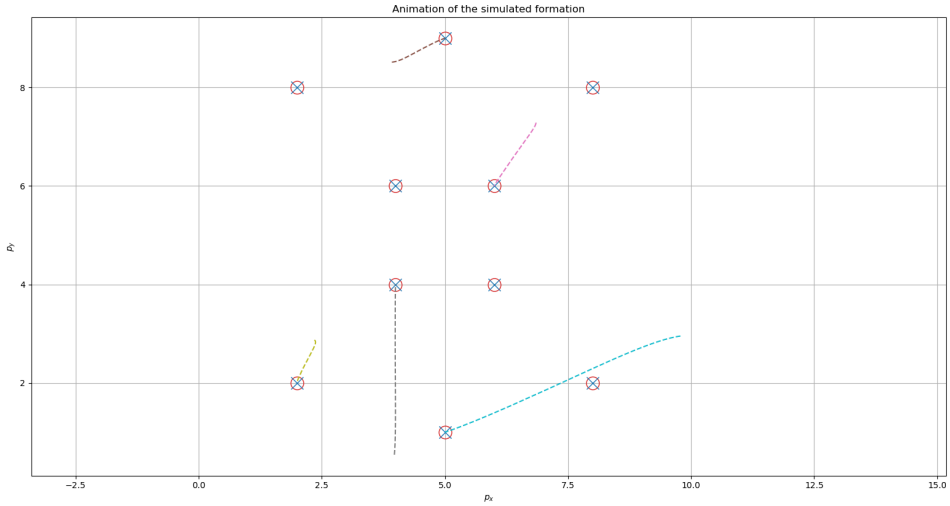


Figure 2.11: Simulation result of group formation - 4 leaders (still)

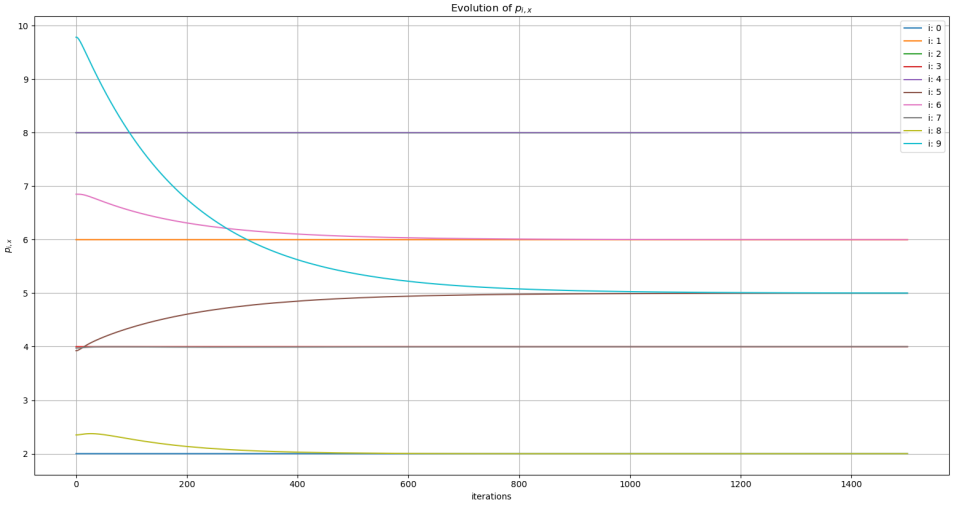


Figure 2.12: Evolution of $p_{i,x}$ during group formation - 4 leaders (still)

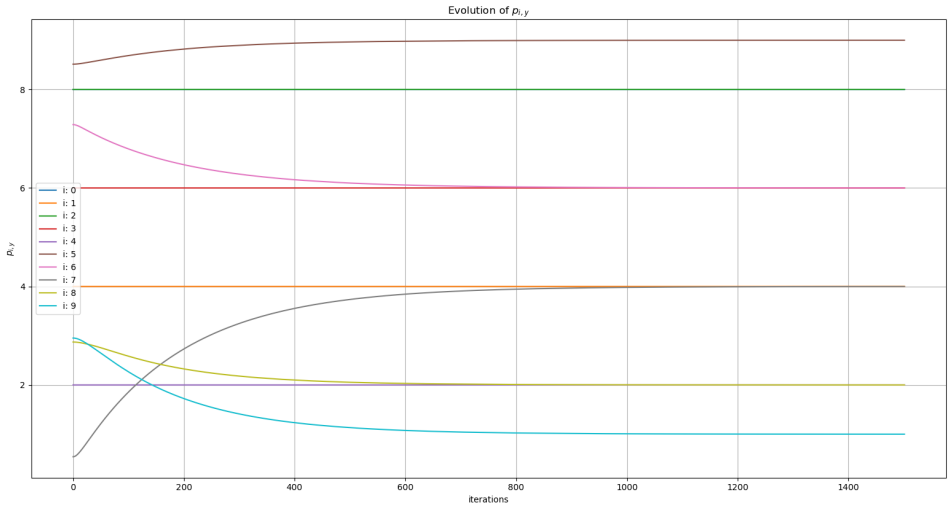


Figure 2.13: Evolution of $p_{i,y}$ during group formation - 4 leaders (still)

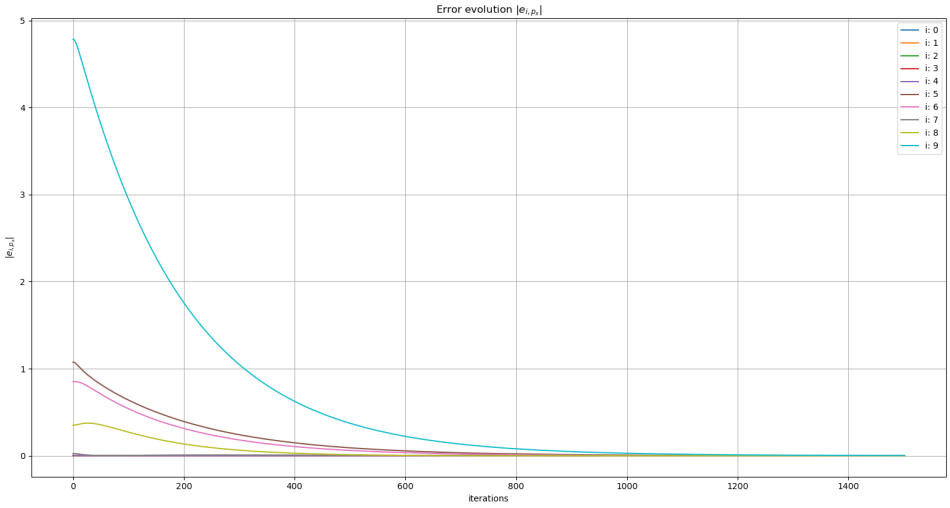


Figure 2.14: Evolution of $|e_{i,p_x}| = |p_{i,x} - p^*_{i,x}|$ during group formation - 4 leaders (still)

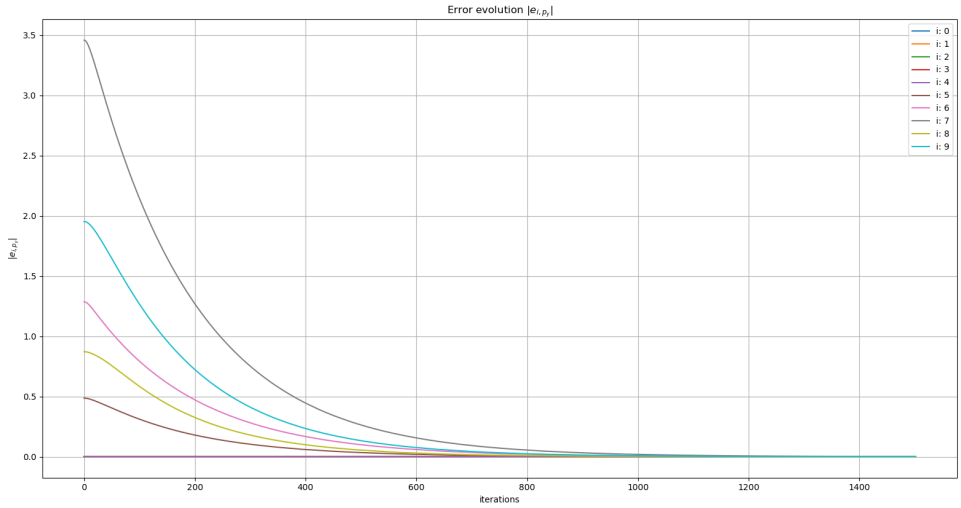


Figure 2.15: Evolution of $|e_{i,p_y}| = |p_{i,y} - p^*_{i,y}|$ during group formation - 4 leaders (still)

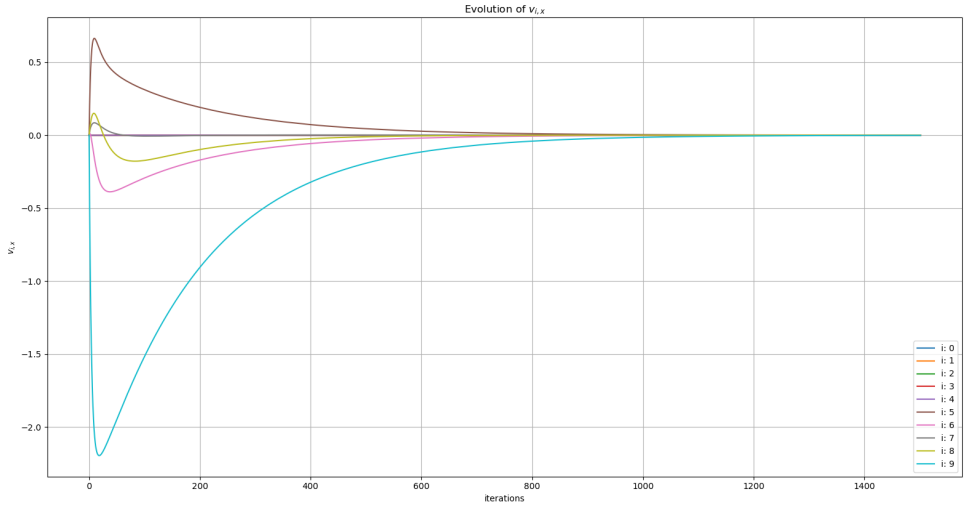


Figure 2.16: Evolution of $v_{i,x}$ during group formation - 4 leaders (still)

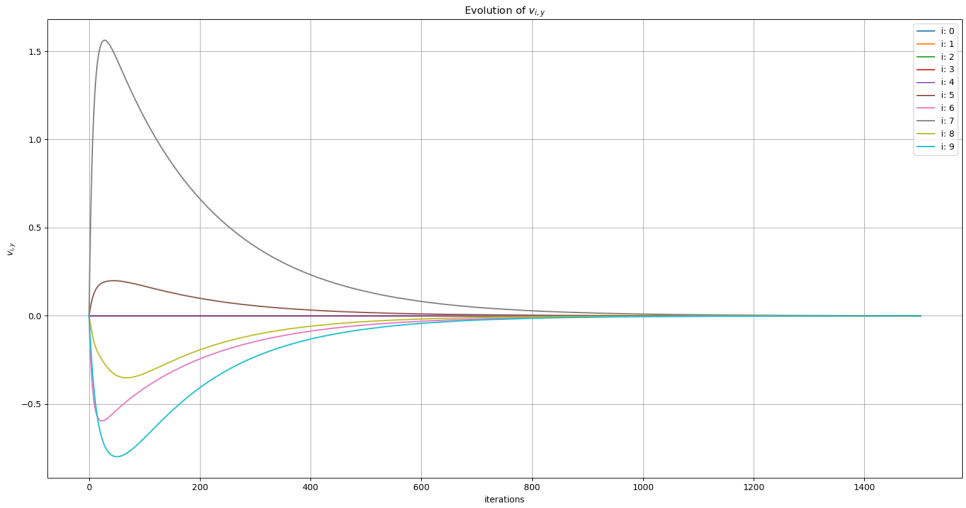


Figure 2.17: Evolution of $v_{i,y}$ during group formation - 4 leaders (still)

2.4.2 Formations with variable leader velocity leader

The leaders follow the acceleration profile $\ddot{q}(t)$ as shown in 2.1 and defined in equation 2.10. For the leaders the control law in 2.7 have been applied, where also accelerations of neighbours are taken into account. They have been estimated by storing the previous velocity values of each agent's neighbors, using them to compute the acceleration numerically as previously said. This approach has been chosen in order to remain consistent with the paper notation and not augment the system state. However, it would have been the simplest choice even if it would have had an impact on the code efficiency. Some of the results are visible in figures 2.18, 2.23 and 2.28.

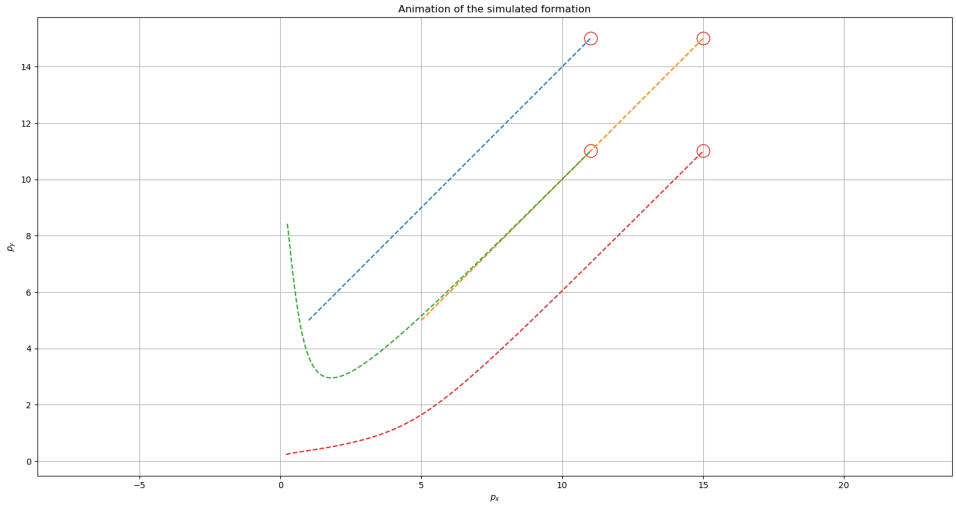


Figure 2.18: Simulation result of square formation - 2 leaders (moving)

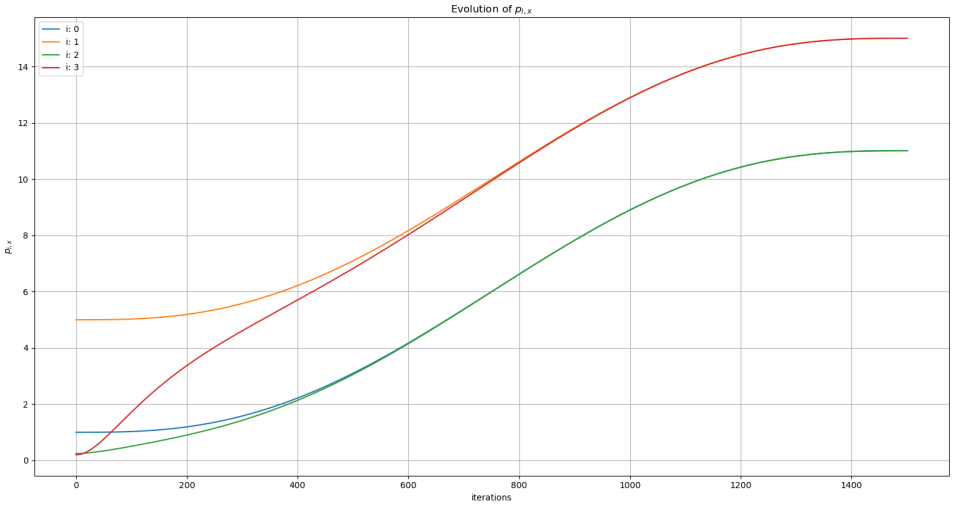


Figure 2.19: Evolution of $p_{i,x}$ during square formation motion - 2 leaders (moving)

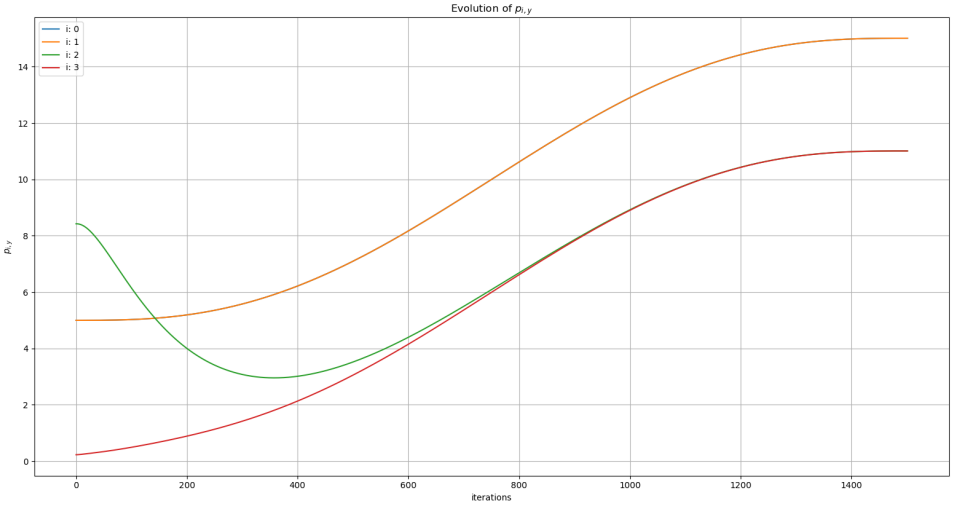


Figure 2.20: Evolution of $p_{i,y}$ during square formation motion - 2 leaders (moving)

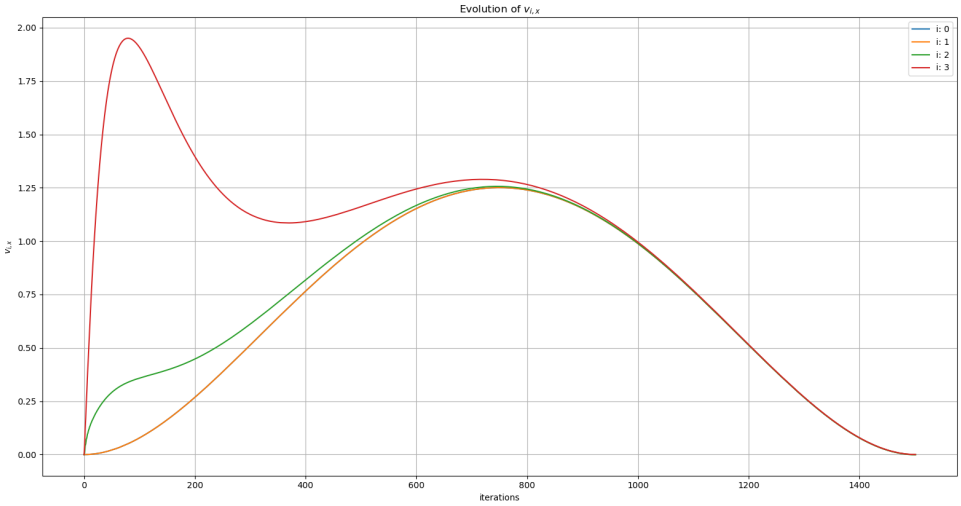


Figure 2.21: Evolution of $v_{i,x}$ during square formation motion - 2 leaders (moving)

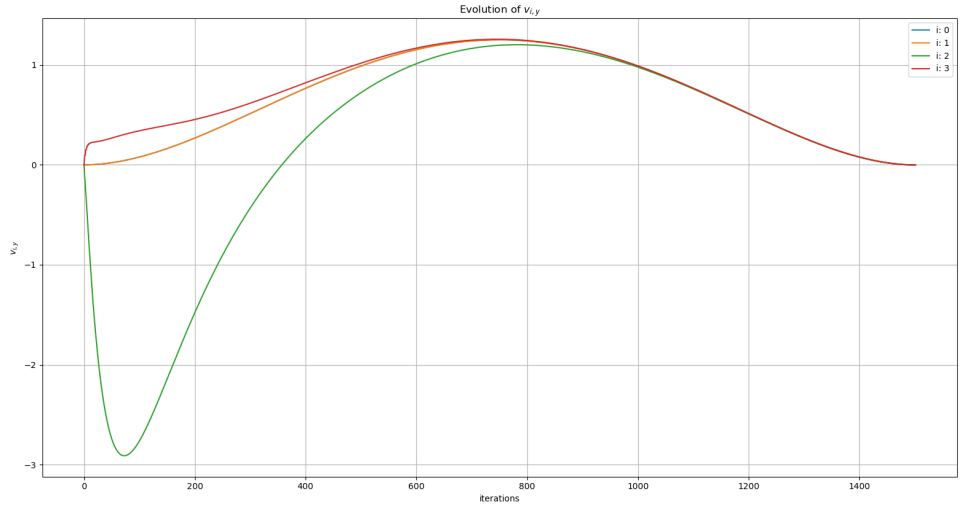


Figure 2.22: Evolution of $v_{i,y}$ during square formation motion - 2 leaders (moving)

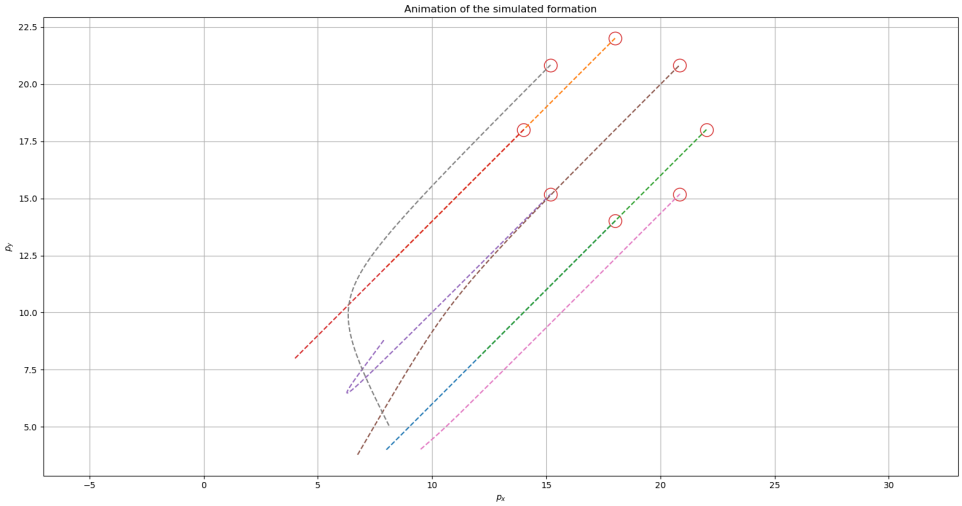


Figure 2.23: Simulation result of octagon formation - 4 leaders (moving)

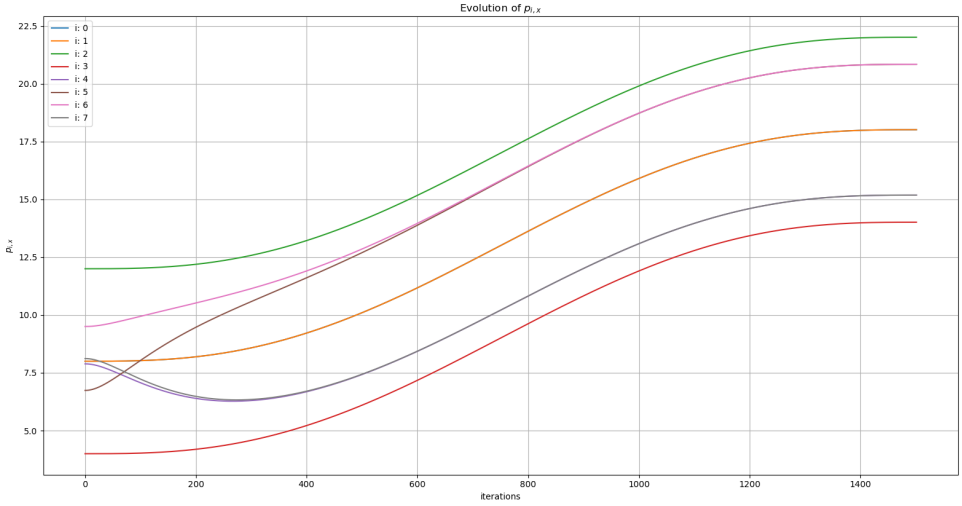


Figure 2.24: Evolution of $p_{i,x}$ during octagon formation - 4 leaders (moving)

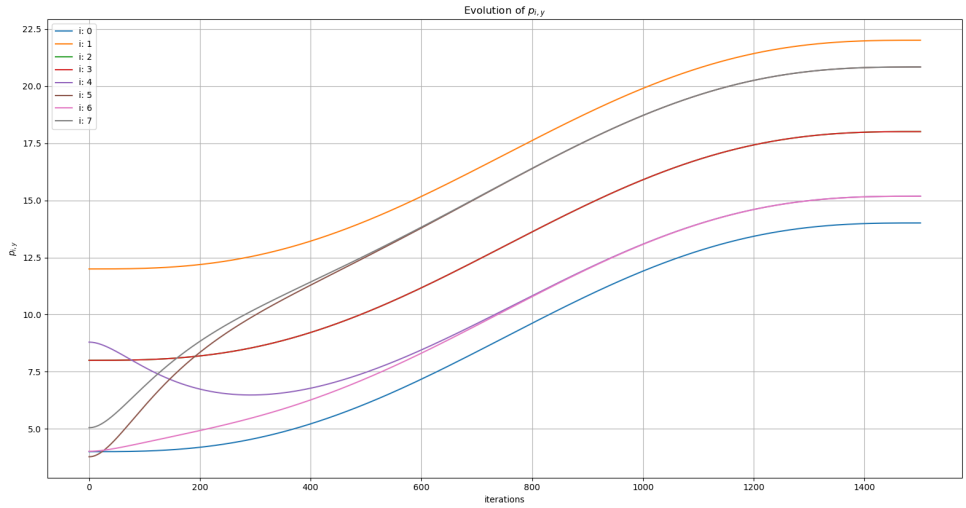


Figure 2.25: Evolution of $p_{i,y}$ during octagon formation - 4 leaders (moving)

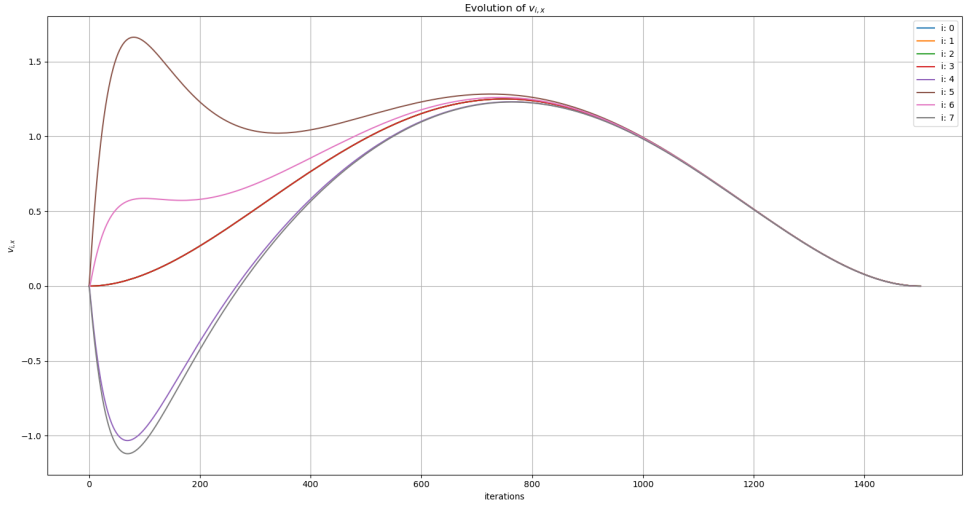


Figure 2.26: Evolution of $v_{i,x}$ during octagon formation - 4 leaders (moving)

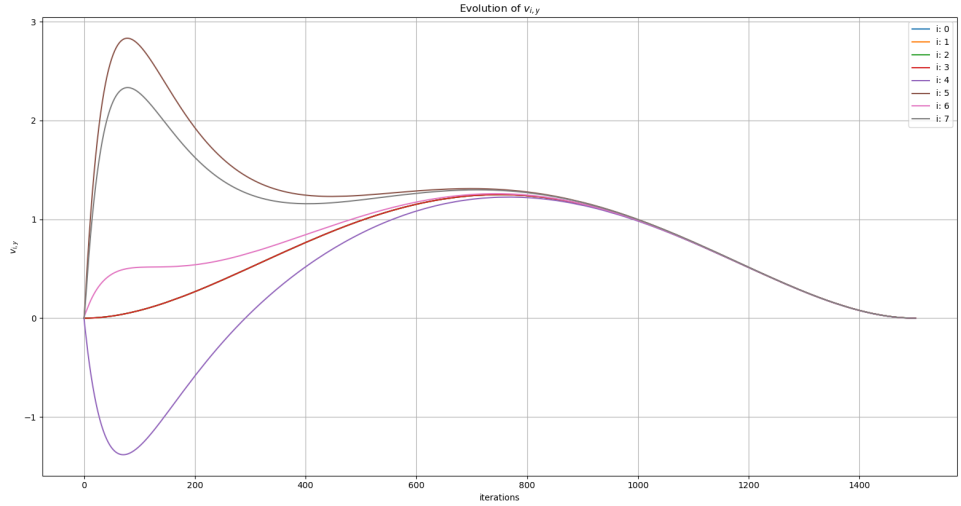


Figure 2.27: Evolution of $v_{i,y}$ during octagon formation - 4 leaders (moving)

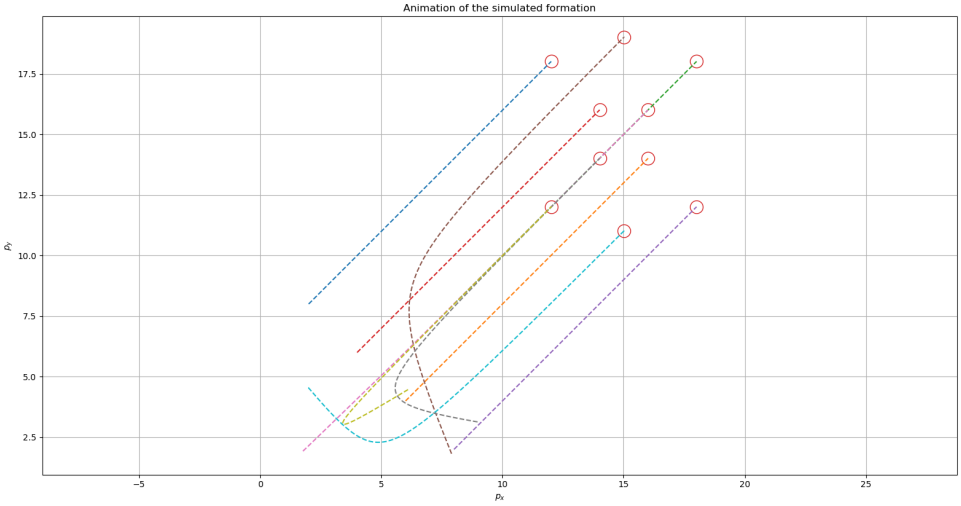


Figure 2.28: Simulation result of group formation - 4 leaders (moving)

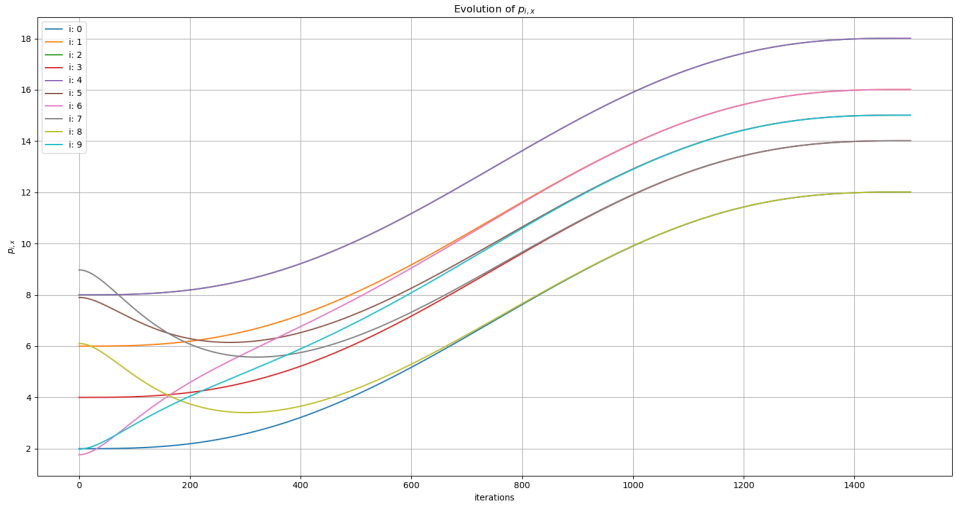


Figure 2.29: Evolution of $p_{i,x}$ during group formation - 4 leaders (moving)

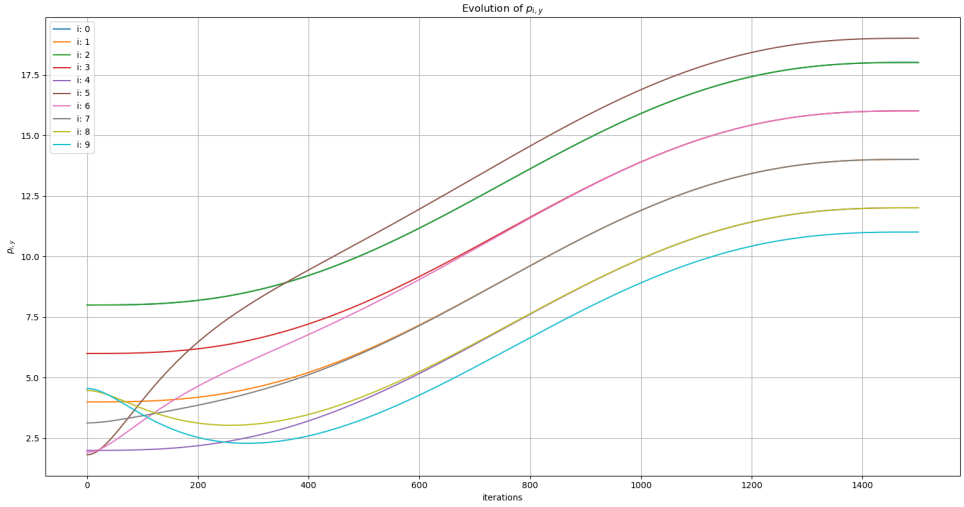


Figure 2.30: Evolution of $p_{i,y}$ during group formation - 4 leaders (moving)

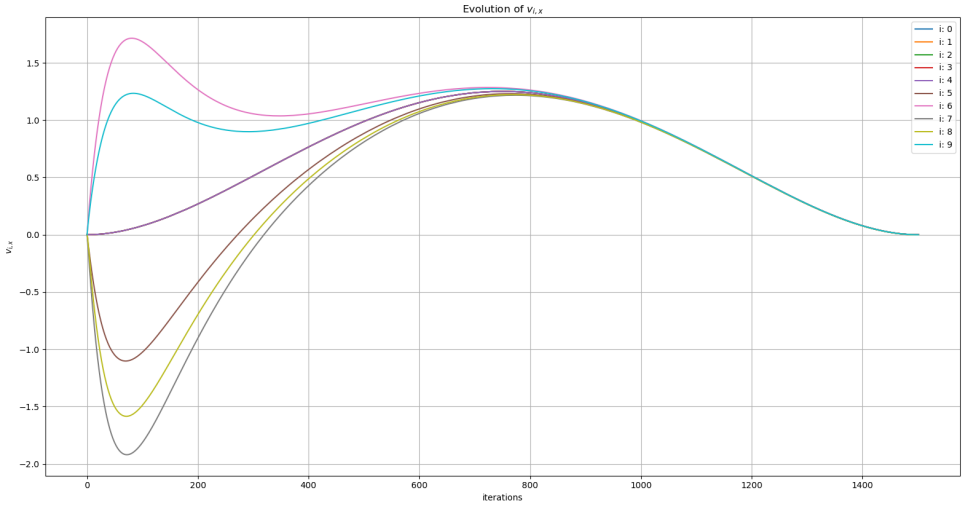


Figure 2.31: Evolution of $v_{i,x}$ during group formation - 4 leaders (moving)

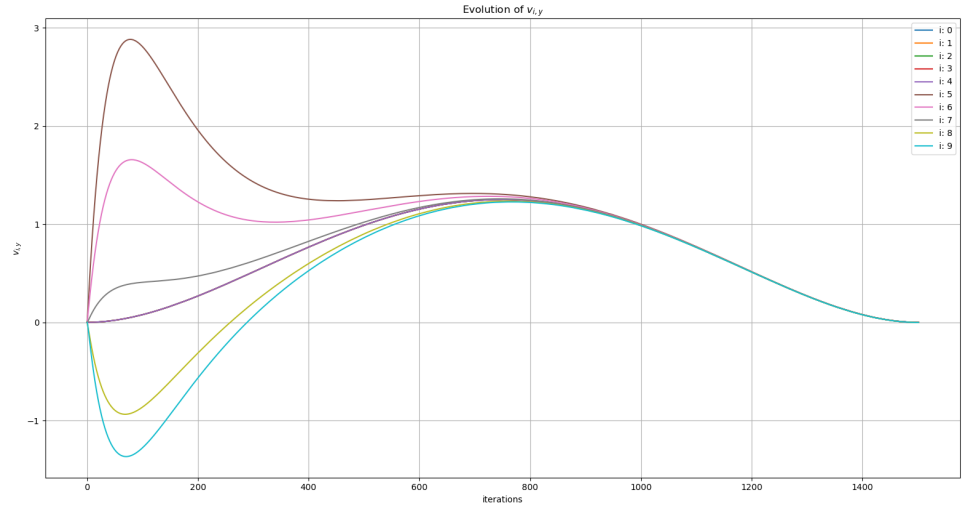


Figure 2.32: Evolution of $v_{i,y}$ during group formation - 4 leaders (moving)

2.4.3 Variable formation to draw given letters

For what concern the last point, the same reasoning discussed before has been applied in a more complex scenario where it has been necessary to draw consecutively letters of a given word. This is a challenging task since leaders and followers should reach the reference positions of the given letter, and once that change, start following the next one. In order to do that, based on the same polynomial function discussed in 2.10, different functions have been computed, one for each trait between two consecutive letters, in order to define the acceleration profiles that must be imposed as control input to the leaders. In this way, they are able to move from any given initial position to the reference one, and then move again to reach the next letter. Obviously, during the motion of the leaders, the followers follow the leaders mantaining their trajectory. The function which computes the acceleration profiles is called *acc_profile()* and is

in file *functions_lett.py*. The results have been showed in figures 2.33, 2.34 and 2.35 where it is also possible to see that we have implemented the possibility to see the real-time simulations using RViz.

Figure 2.33: D letter of word "DAS" in RViz

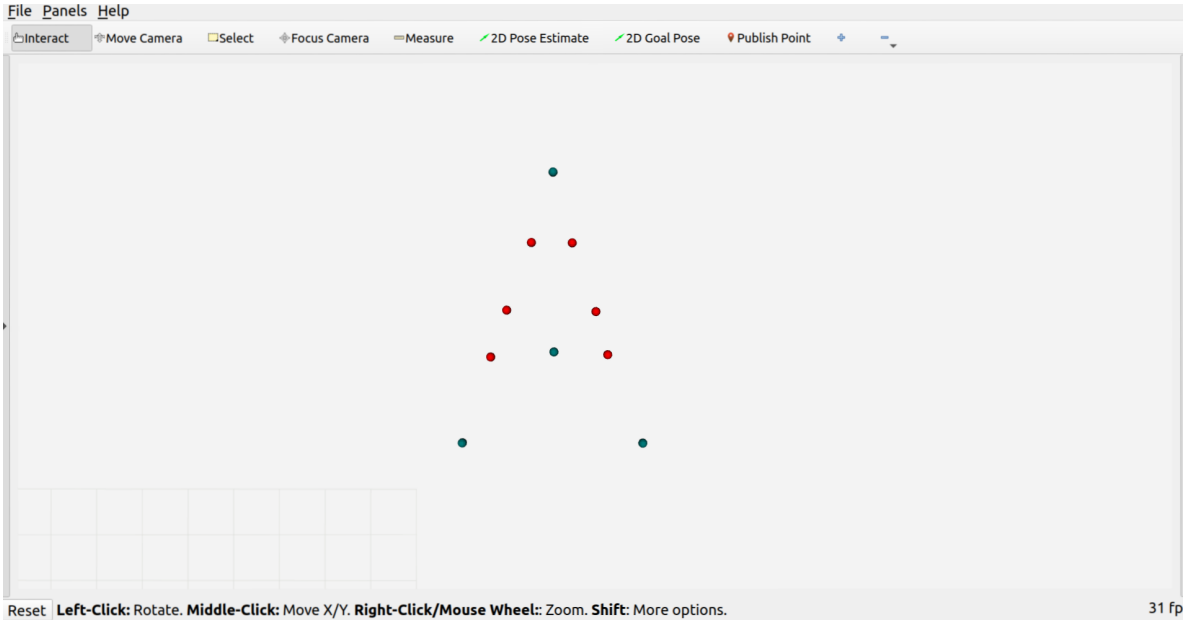


Figure 2.34: A letter of word "DAS" in RViz

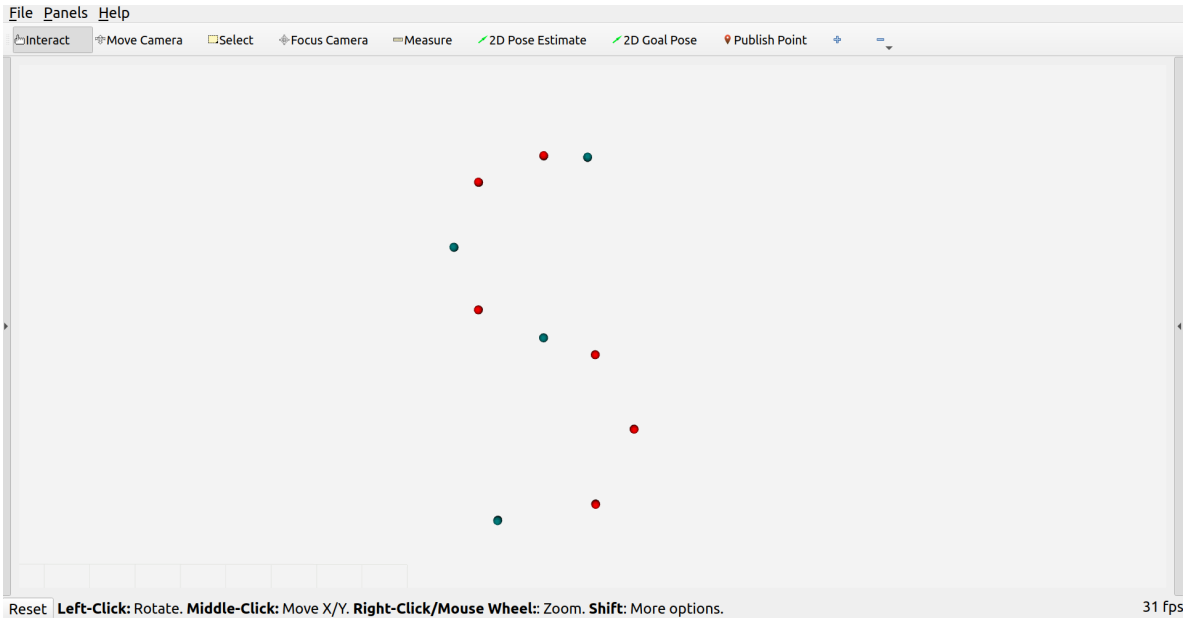


Figure 2.35: S letter of word "DAS" in RViz

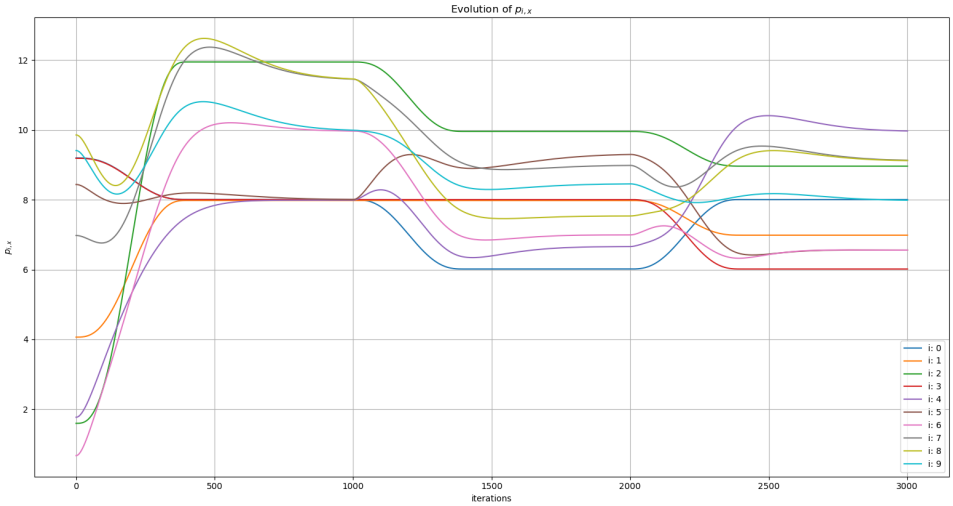


Figure 2.36: Evolution of $p_{i,x}$ during word "DAS" formation

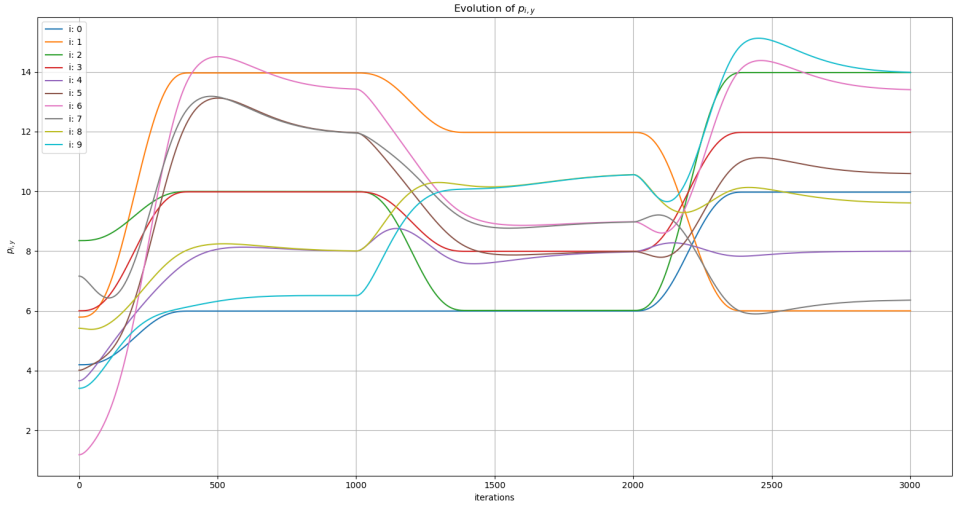


Figure 2.37: Evolution of $p_{i,y}$ during word "DAS" formation

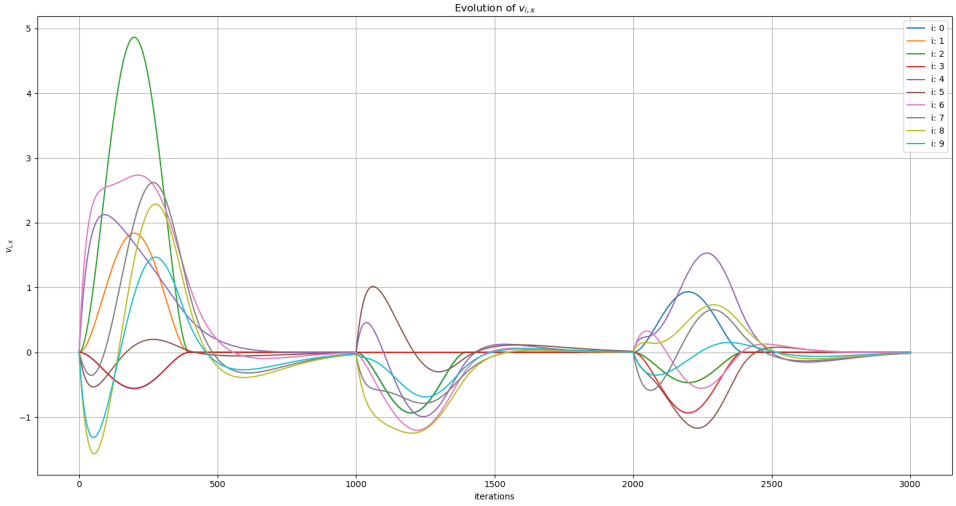


Figure 2.38: Evolution of $v_{i,x}$ during word "DAS" formation

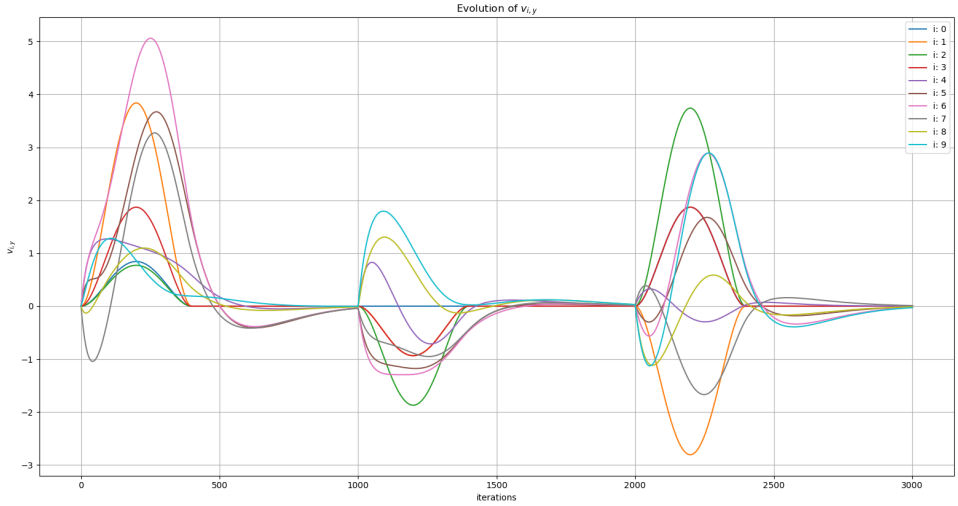


Figure 2.39: Evolution of $v_{i,y}$ during word "DAS" formation

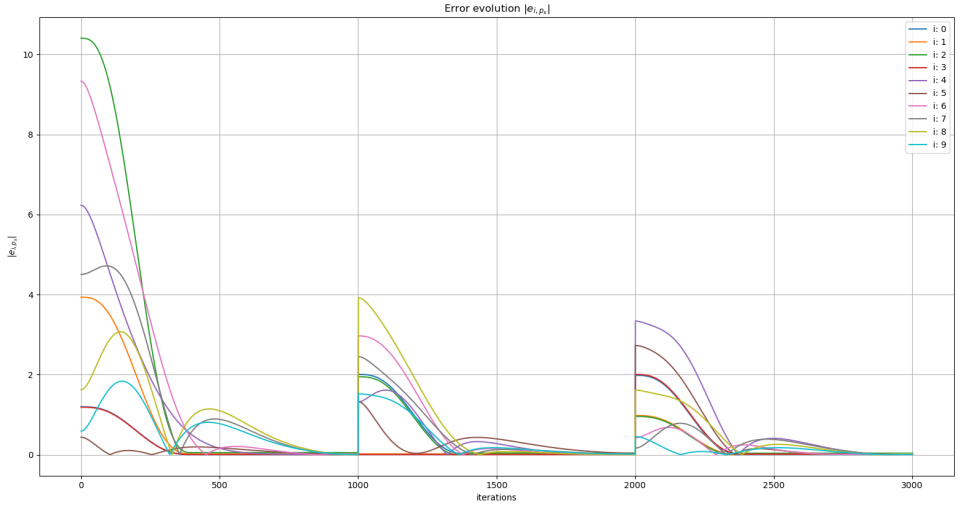


Figure 2.40: Evolution of $|e_{i,p_x}| = |p_{i,x} - p^*_{i,x}|$ during word "DAS" formation

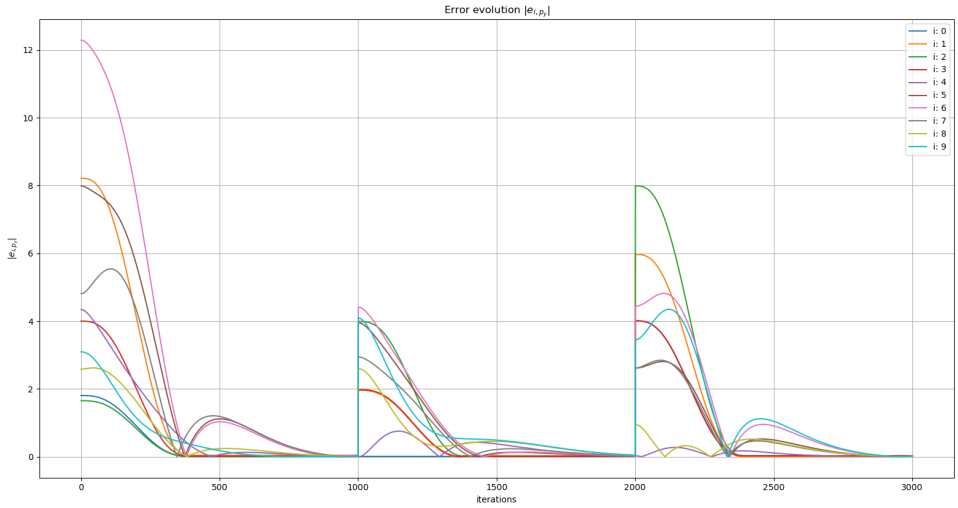


Figure 2.41: Evolution of $|e_{i,p_y}| = |p_{i,y} - p^*_{i,y}|$ during word "DAS" formation

2.4.4 Plus: Integral Action

Also integral action in 2.9 has been implemented, as presented in the reference paper. Very good results have been obtained also in this case, because agents converge to their reference position even in presence of a constant disturbance action on the input acceleration. However, the number of iterations needed to converge are about twice the case without disturbance. As can be seen in figure 2.42, there is an initial overshoot in the followers caused by the constant term that is slowly compensated as iterations increase.

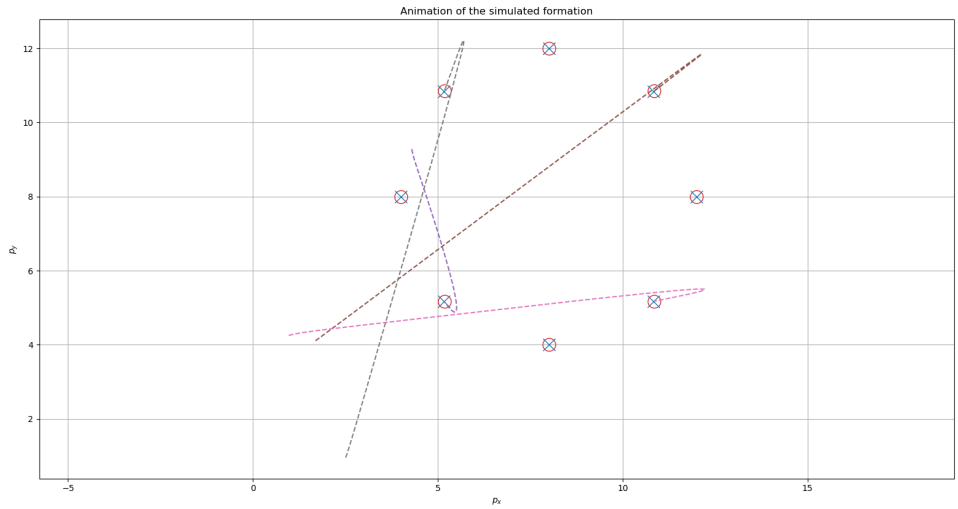


Figure 2.42: Simulation result of octagon with **integral action** - 4 leaders (still)

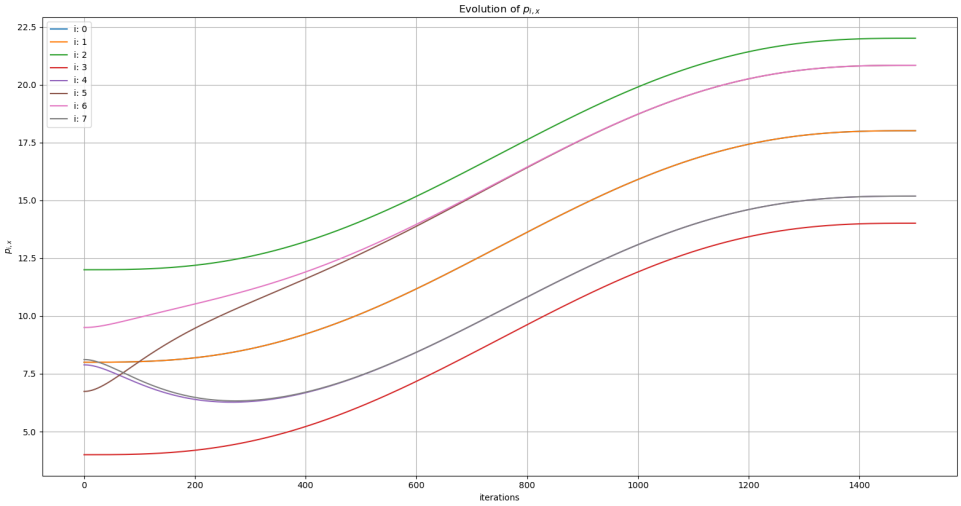


Figure 2.43: Evolution of $p_{i,x}$ during octagon with **integral action** - 4 leaders (still)

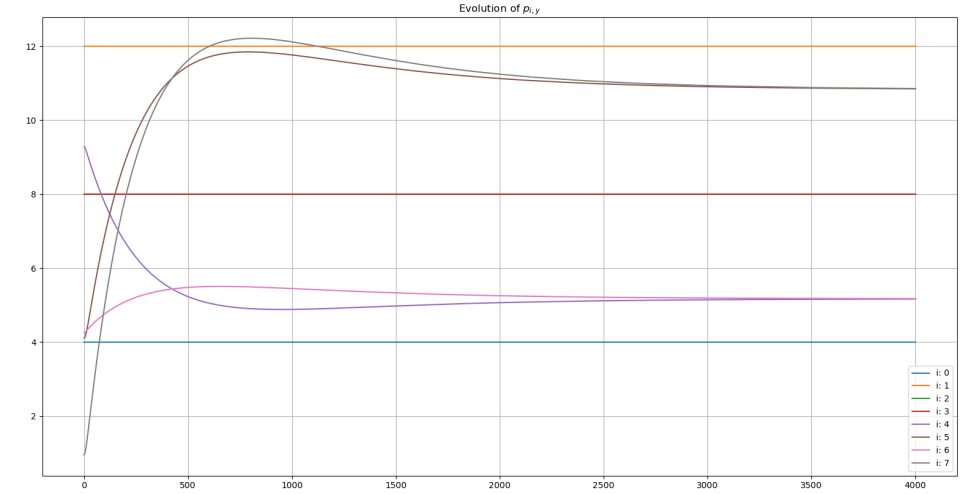


Figure 2.44: Evolution of $p_{i,y}$ during octagon with **integral action** - 4 leaders (still)

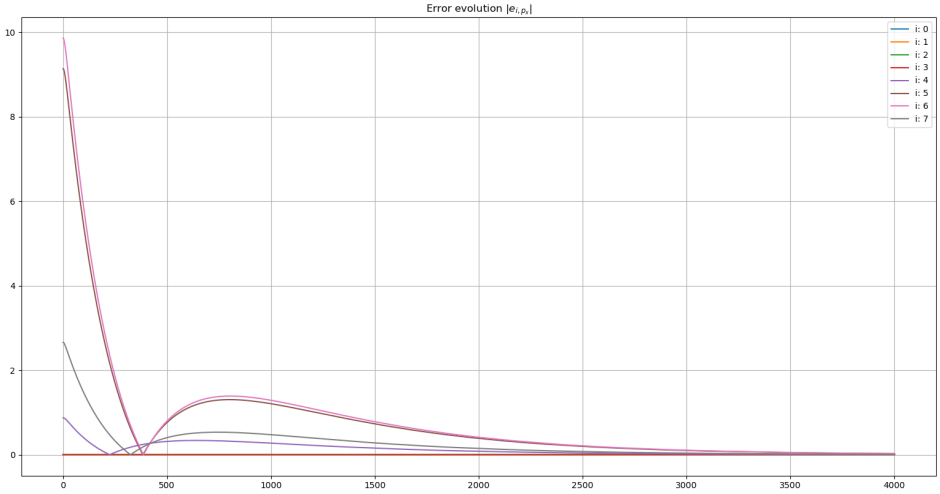


Figure 2.45: Evolution of $|e_{i,p_x}| = |p_{i,x} - p^*_{i,x}|$ during octagon with **integral action** - 4 leaders (still)

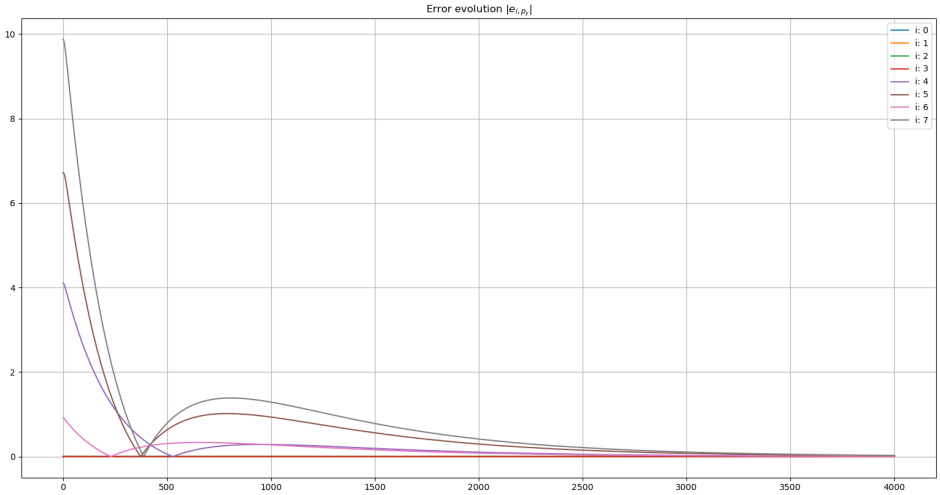


Figure 2.46: Evolution of $|e_{i,p_y}| = |p_{i,y} - p^*_{i,y}|$ during octagon with **integral action** - 4 leaders (still)

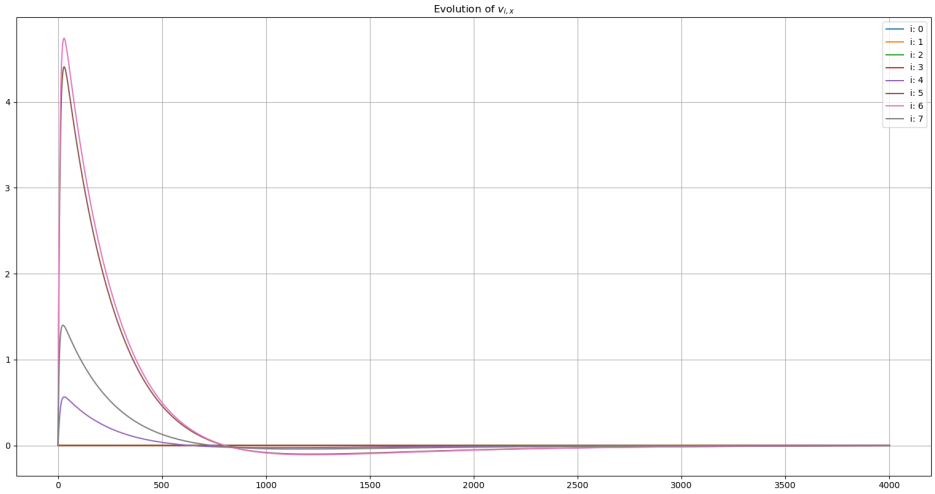


Figure 2.47: Evolution of $v_{i,x}$ during octagon with **integral action** - 4 leaders (still)

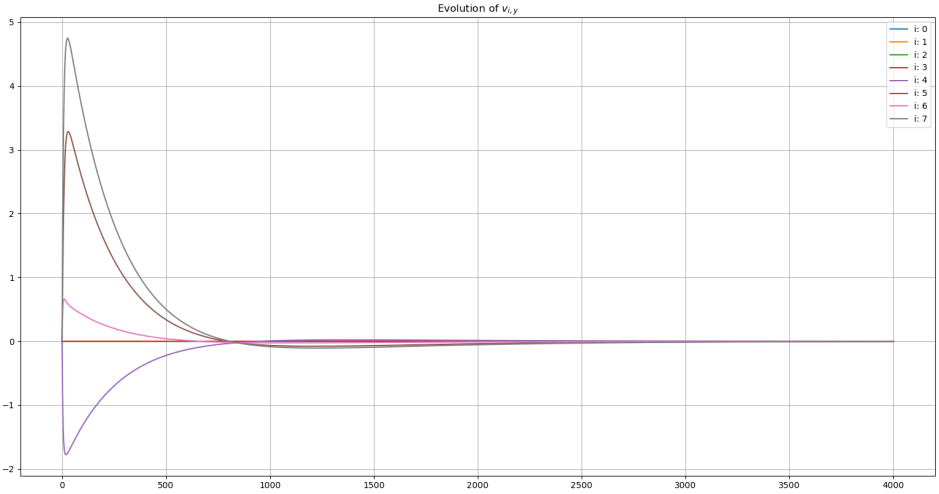


Figure 2.48: Evolution of $v_{i,y}$ during octagon with **integral action** - 4 leaders (still)

Conclusions

Building from scratch the Neural Network has been useful to understand how does it work starting from its backpropagation algorithm which is not common, because usually Neural Networks are used with the interal structure already defined in proper frameworks (e.g. Keras). Moreover, the distributed part obtained by using the causal gradient tracking, gives amazing results in terms of convergence and training of the NN. As possible improvement it could be useful to optimize the computation time of the code, for example by using a more efficient construct instead of the numpy arrays, using data structure with variable dimensions between different layers (e.g. lists).
For what concern the Formation Control, the results obtained show that the agents are able to follow a desired trajectory maintaining the given shape pretty well. It would be of interest introduce some communication error in order to verify the robustness of this method. Also the integral action worked, but the convergence has been reached in a huge amount of iterations and with still leaders. Further improvements may consider efficiency in the code (which has not been the main objective of this work) to make it work on real hardware.

Bibliography

[1] Wei Ren and Randal W Beard. *Distributed consensus in multi-vehicle cooperative control*, volume 27. Springer, 2008.

[2] Shiyu Zhao and Daniel Zelazo. Translational and scaling formation maneuver control via a bearing-based approach. *IEEE Transactions on Control of Network Systems*, 4(3):429–438, 2015.