UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Autonomous Systems

Course Project

Professors:
Giuseppe Notarstefano
Ivano Notarnicola

Students:
Gianluca Di Mauro
Andrea Perna
Meisam Tavakoli

Academic year 2023/2024

Abstract

This report details the development and outcomes of implementing two primary tasks in distributed autonomous systems. The first task focuses on in-depth data analytics, specifically distributed classification and optimization, utilizing the gradient tracking algorithm for distributed optimization and subsequent classification of a generated dataset. The second task addresses the control of multi-robot systems in ROS 2, employing the Aggregative Tracking framework to maintain robot formation while navigating towards targets. Both tasks were successfully executed in a Python environment, yielding promising results.

Contents

Introduction

Distributed autonomous systems are pivotal in modern technology, enabling applications from industrial automation to smart grids. Central to these systems are multi-agent systems, where numerous autonomous agents collaborate to achieve shared goals. These systems are scalable, robust, and capable of functioning despite individual agent failures, making them crucial for applications such as sensor networks and robotic swarms. In particular, distributed systems with collaborative agents excel at solving complex problems through coordinated efforts. Consensus-based distributed algorithms are essential, ensuring agents across the network agree on parameters or decisions, thus enabling synchronized actions and coherent behavior. This project focuses on distributed classification and distributed robotics, employing aggregative gradient algorithms and gradient tracking algorithms to address distributed optimization problems. These consensus-based algorithms optimize global objectives while managing local computations and communications efficiently. By achieving consensus, they allow agents to converge on optimal solutions despite delays, noise, and other uncertainties. The importance of these algorithms lies in their ability to integrate individual agent contributions into a cohesive whole, facilitating efficient and accurate information processing across networks. This report explores the role of these algorithms in distributed classification and robotics, highlighting their significance in developing robust, scalable, and efficient distributed autonomous systems.

Chapter 1

Task1 - Distributed Classification via Logistic Regression

1.1 Distributed Optimization

1.1.1 Theoretical Framework

The first task focuses on implementing Gradient Tracking (GT), a Machine Learning algorithm tailored for solving decentralized optimization problems. In such problems, each agent is aware only of a local objective function, and the collective aim is to minimize the sum of these local objective functions across all agents to reach a consensual solution z^* of the following problem:

$$min_z \sum_{i=1}^{N} l_i(z)$$

The Gradient Tracking algorithm is defined as follows:

$$z_i^{k+1} = \sum_{j \in N_i} a_{ij} z_j^k - \alpha s_i^k \qquad z_i^0 \in \mathbb{R}$$

$$s_i^{k+1} = \sum_{j \in N_i} a_{ij} s_j^k + \nabla \ell_i \left(z_i^{k+1} \right) - \nabla \ell_i \left(z_i^k \right) \quad s_i^0 = \nabla \ell_i \left(z_i^0 \right)$$

The effectiveness of GT depends on the following assumptions:

- Let $a_{ij}, i, j \in \{1, ..., N\}$ be nonnegative entries of a weighted adjacency matrix A associated to the undirected and connected graph G, with $a_{ii} > 0$ and A doubly stochastic.
- For all $i \in \{1, ..., N\}$, each cost function $l_i : \mathbb{R}^d \to \mathbb{R}$ is strongly convex with coefficient $\mu > 0$ and has Lipschitz continuous gradient with constant L > 0

1.1.2 Problem Set-Up

To address this task, preliminary steps were undertaken before implementing the main Gradient Tracking (GT) algorithm. Initially, two functions were defined for graph initialization. The first function generates a random but symmetric adjacency matrix, representing a connected graph with N nodes. The second function computes graph's weights using the Metropolis-Hastings method, which is an algorithm that generates a Markov chain on a graph [1] [2].

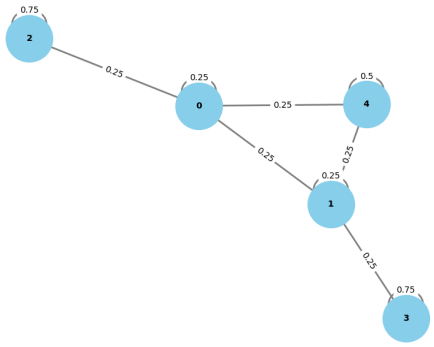
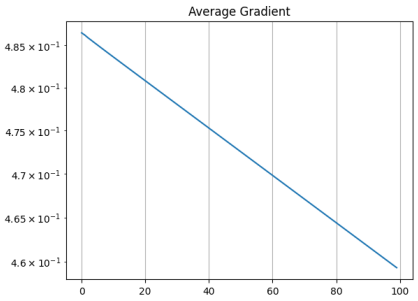


Figure 1.1: Graph Connectivity, 5 Agents

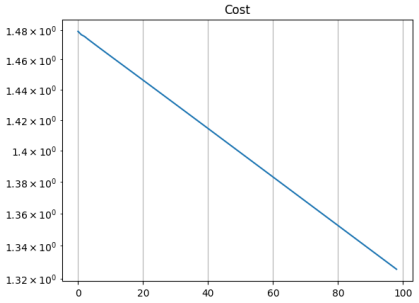
The core component of this task is the implementation of the Gradient Tracking algorithm, shown in 2.5.5. For this implementation, a matrix-based approach has been chosen, which is particularly suitable for handling graph structures, especially when large. The algorithm operates on the adjacency matrix of the graph, which is passed to the "gradient_tracking" function. This latter also takes several other parameters, such as the number of iterations and nodes in the graph. The adjacency matrix A, derived from the Metropolis-Hastings algorithm, is used to calculate the transition probabilities for the Markov chain. Parameters a and b are used in the quadratic function that the algorithm seeks to minimize, which represents the cost that the algorithm aims to reduce. The gradient of this function, computed at each iteration, guides the algorithm in the direction of the steepest descent of the cost. The gradient_tracking function, in particular, operates by iteratively updating agents' states Z and gradients S. This function, also computes the cost at each iteration, providing a measure of the algorithm's progress. By the end of the specified number of iterations, the function returns the final state of the decision variables and the cost at each iteration. The decision variables of the agents, in this particular framework (cost-coupled optimization), should be all equal at the end of the algorithm, i.e., they should have reached consensus. For this subtask, scalar decision variables have been taken into account.

1.1.3 Simulations

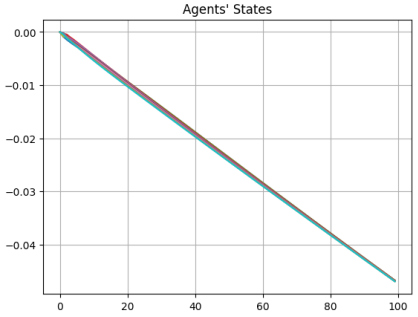
The results of an experiment with 10 agents are shown below, with a step-size $\alpha = 1e^{-3}$ in N=100 iterations. The choice of α is crucial: if it was set too high, or the number of iterations was not sufficient, then convergence wouldn't happen and cost would rise, as shown in 1.4a and 1.4b, respectively.



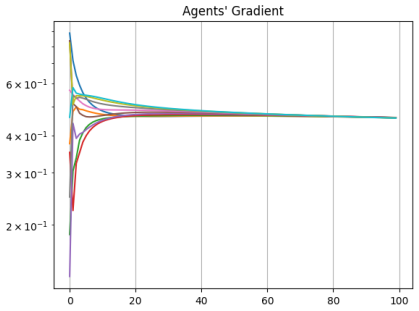
(a) Evolution of Gradient, α=1e-3



(b) Evolution of Cost, α=1e-3

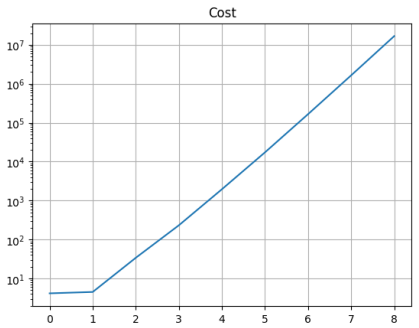


(a) Agents' States, α=1e-3



(b) Agent's Gradient, α=1e-3

(a) Agents' States, α=0.8



(b) Evolution of Cost, α=0.8

1.2 Centralized Classification

1.2.1 Dataset and Conic Generation

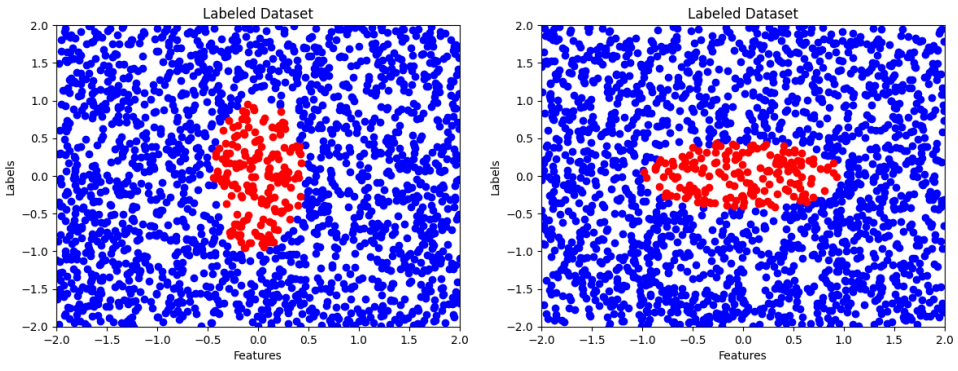
The second task involved regards Centralized Classification, which begins with generating a dataset of $\mathcal{M} \in \mathbb{N}$ points: $\mathcal{D}^m \in \mathbb{R}^d, m = 1, \ldots, \mathcal{M}$. Each point $\mathcal{D}^m$ is binary labeled according to a separating function in the form:

$$\left\{ x \in \mathbb{R}^d \mid w^\top \varphi(x) + b = 0 \right\}.$$

where $\varphi : \mathbb{R}^d \to \mathbb{R}^q$ is a nonlinear function and $w \in \mathbb{R}^q, b \in \mathbb{R}$ parameters. The labeling criteria are defined as:

$$\begin{aligned} w^\top \varphi\left(\mathcal{D}^m\right) + b \geq 0, \quad & \text{if } p^m = 1, \\ w^\top \varphi\left(\mathcal{D}^m\right) + b < 0, \quad & \text{if } p^m = -1. \end{aligned}$$

Here, w determines the shape of the function, while b represents its bias or "shift". Various function shapes were tested, yielding different results, which will be discussed later. Several auxiliary functions were defined for this task. First, a function to determine the conic parameters for the chosen separating function was implemented. This function computes the appropriate values for w and b, as well as the center coordinates and intersection points with the conic's axes. These parameters are then utilized by the generate_dataset() function, which generates and labels the points to create the dataset. This dataset is subsequently split into training and test sets using the standard hold-out method provided by scikit-learn's train_test_split() function [3]. A notable feature of generate_dataset() is the option to generate points around the center of the conic by setting the centered_data parameter to True. This involves computing the conic's center and semi-axes to generate a more balanced and heterogeneous dataset. Examples of generated and labeled datasets are shown in Figures 2.32a and 2.32b.



(a) vert_ellipse (b) hor_ellipse

Figure 1.5: Datasets Generation

1.2.2 Centralized Gradient Method

After generating the dataset, a centralized gradient method (CGM) was employed to implement a logistic regressor for classifying the points. The optimization problem can be formulated as follows:

$$\min_{w,b} \sum_{m=1}^{\mathcal{M}} \log\left(1+\exp\left(-p^m\left(w^\top \varphi\left(\mathcal{D}^m\right)+b\right)\right)\right) \tag{1.1}$$

where w and b are the optimization variables. Using classical gradient descent, the loss is minimized at each step by computing the gradient for both variables w and b through an auxiliary function that computes the loss and gradients, and store them over the iterations. The loss function used for the CGM is shown in 1.1. The execution of the CGM is halted by a control variable, "min_grad_norm", which determines the minimum gradient norm value at which the execution stops. The pseudocode for such an algorithm is shown in 2.5.5. The CGM is executed on the training set, to find the optimal w and b values. Aftwerward, classification is performed by using a sigmoid function, of the form:

$$\frac{1}{(1+e^{-Z})} \tag{1.2}$$

which maps the points to the range (0,1), where Z is an array containing the dot product between w and each point D plus the bias b. The ultimate learning procedure reads:

$$w^\top \varphi\left(\mathcal{D}^m\right)+b \tag{1.3}$$

Gradient and cost related to a centralized classification experiment are shown below, where it is possible to appreciate the effectiveness of CGM in finding the optimal decision variables.

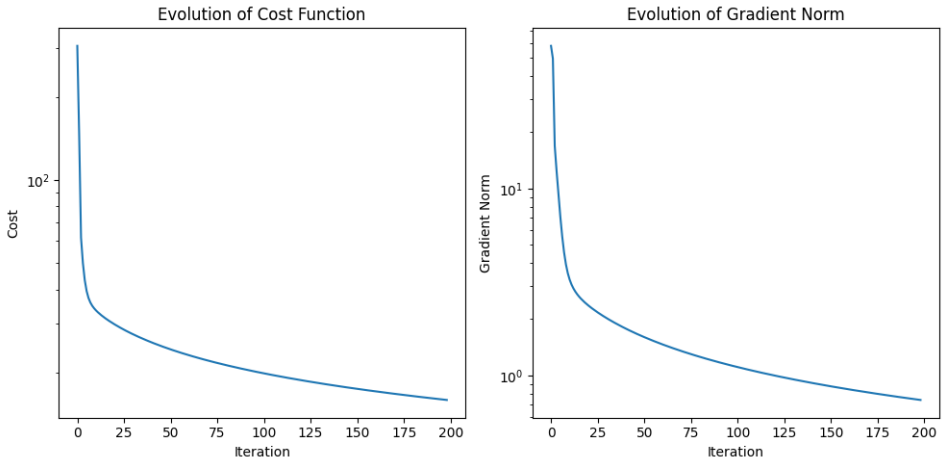
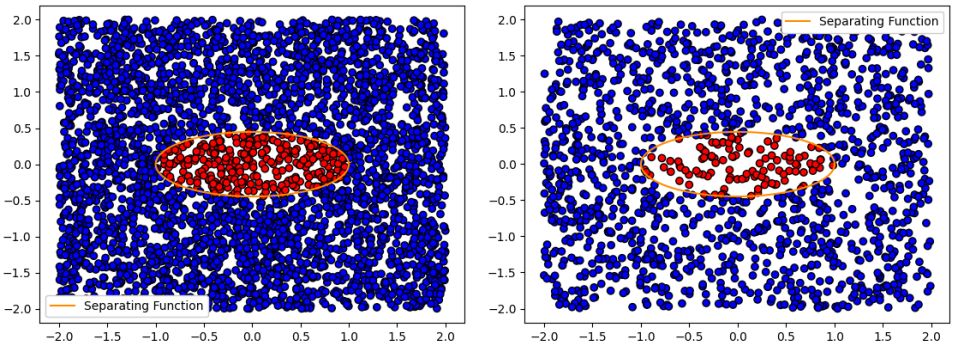


Figure 1.6: $\alpha = 5e^{-2}$, min_grad_norm=$2e^{-2}$.

1.2.3 Simulations

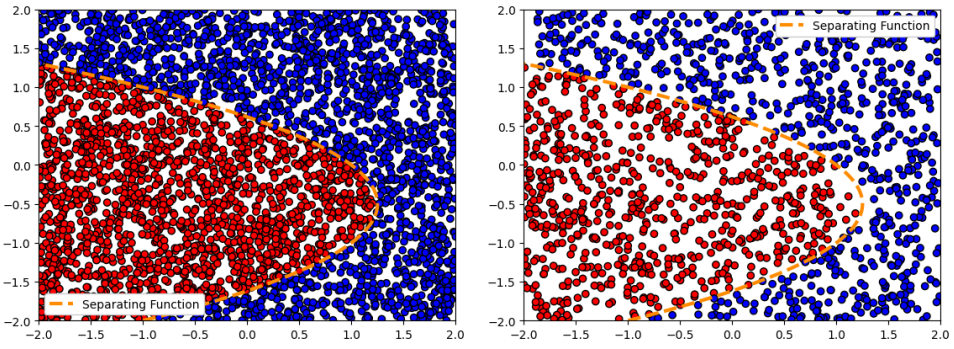
A threshold of 0.5 is applied to classify each point as either 0 or 1: if an element in Z is greater than 0.5, it becomes 1; otherwise, it becomes 0. Classification is done on both the training set and the testing set to evaluate performance metrics. Classification results are shown in Figures, 1.7a, 1.7b, 1.8a, 1.8b, with parameters α=5e-2, min_grad_norm=2e-2.



(a) Training set (b) Test set

Figure 1.7: Classification on horizontal Ellipse

The obtained results are highly satisfactory, with a classification error of 0% on the training set over multiple runs. The error rate on the test set also dropped to 0.0% in several setups, indicating high classification accuracy. Detailed results are shown in Table 1.1. These results were obtained from multiple runs on different shapes for the classification function over 5000 points. The step-size was kept at 0.6 for all the reported experiments.



(a) Training set (b) Test set

Figure 1.8: Classification on Parabola

Metric	Hor. ellipse		Vert. ellipse		Parabola	
	Train Set	**Test Set**	**Train Set**	**Test Set**	**Train Set**	**Test Set**
Classification Error	0.0%	0.2%	0.0%	0.0%	0.0%	0.3%
Accuracy	1.0	0.998	1.0	1.0	1.0	0.997
Precision	1.0	0.999	1.0	1.0	1.0	1.0
Recall	1.0	0.999	1.0	1.0	1.0	0.994
F1 Score	1.0	0.999	1.0	1.0	1.0	0.997

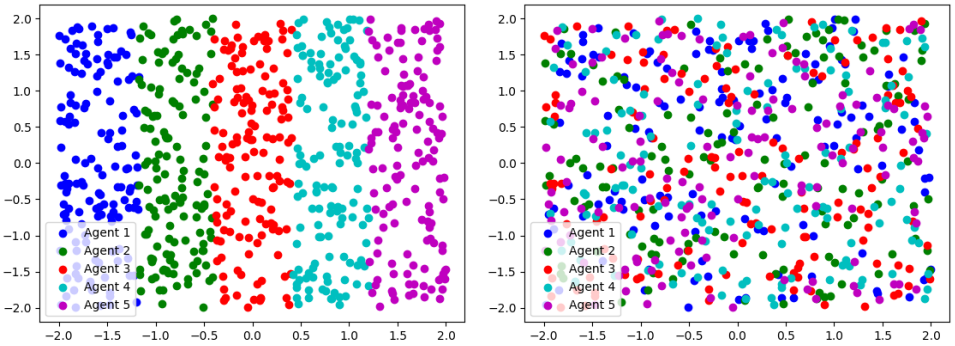
Table 1.1: Metrics for different shapes with 5000 points

1.3 Distributed Classification

The final task in this chapter is Distributed Classification, which combines elements from the previously discussed tasks in chapters 1.1 and 1.2.

1.3.1 Distributed Dataset Generation

The dataset generated during Task 1.2 is split into N subsets, one for each agent $i \in \{1, \ldots, N\}$. The conic separating function $w^\top \varphi(\mathcal{D}^m) + b$ used to label data points in this task directly comes from task 1.2, since the two methods share the same learning goals. Similar to Task 1.1, agents are represented by nodes in a graph generated using the same strategies and functions. An auxiliary function, called *distribute_data()* was defined to distribute the dataset among all the agents, allowing selection among different splitting patterns, as shown in Figures 1.9a and 1.9b. The distribution can be 'sequential', where data points are sorted by the first feature and evenly divided, or 'random', where data points are shuffled and distributed in a round-robin fashion. This setup helps in evaluating the classification model's performance under varied data conditions and distribution patterns.



(a) Dataset Sequential Split (b) Dataset Random Split

Figure 1.9: Data Distribution

1.3.2 Gradient Tracking with Logistic Regression Loss

To perform classification on the distributed dataset, a modified version of the Gradient Tracking (GT) algorithm has been used, referred to as Distributed Gradient Tracking (DGT). This algorithm, explained in detail in Section 1.1, optimizes two variables, w and b, as discussed in Section 1.2. The DGT algorithm leverages the logistic regression loss function, which has been previously outlined. The key distinction between DGT and the original GT algorithm lies in the tracker's update mechanism, which is executed twice at each step to separately account for the different decision variables. The DGT algorithm was implemented as detailed in Section 2.5.5. Initially, the data is distributed among multiple agents using either a sequential or random pattern. The agents then collaboratively update their local variables and gradient estimates using a weighted sum of their own values and those of their neighbors, as determined by the adjacency matrix. The progress of the DGT execution is illustrated in Figures 1.10 and 1.18, showcasing the convergence of each agent's gradient estimation towards the total gradient until it nearly reaches zero, demonstrating the algorithm's effectiveness in distributed optimization. A stopping criterion ϵ has been implemented for this task, to allow the algorithm running until a certain desired threshold has been met. At the end of the algorithm, optimal decision variables are used to classify the points, and results are compared with the true decision variables. Some examples will be shown in chapter 1.3.5.

1.3.3 Classification Noise

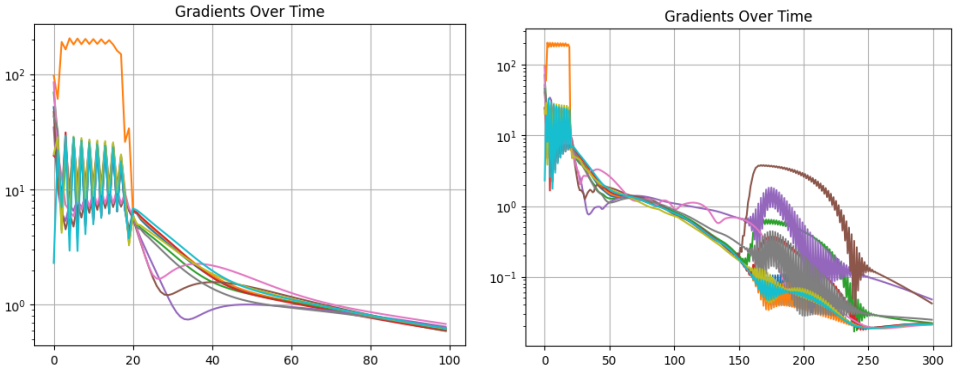
In this task, noise can be added to the training dataset (D_train) to simulate real-world conditions and test model robustness if the noise flag is set to True. The add_noise() function introduces noise by randomly inverting the labels of data points based on a specified noise_level. Introducing noise can reduce the classifier's accuracy, particularly at high noise levels, by making it harder to learn correct decision boundaries. Some experiments with are going presented in this chapter.

1.3.4 Model Evaluation

Various evaluation metrics are used to assess the performance of the classification model. The *evaluation_metrics()* function computes key metrics such as Classification Error, Accuracy, Precision, Recall, and F1 Score for both the training and test sets. Predictions for the training (y_train_pred) and test (y_test_pred) sets are generated using the optimized model parameters (w_opt, b_opt). These predictions are compared against the actual labels (y_train, y_test) to calculate the metrics, which are displayed through the *print_evaluation_metrics()* function.

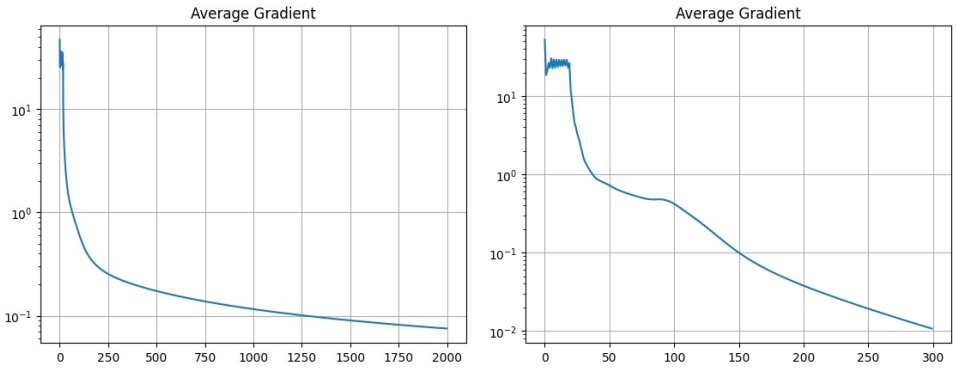
1.3.5 Simulations

The simulation results, similar to Task 1.2, showed a classification error of 0% on the training set and negligible values on the test set in multiple setups, indicating high accuracy. These outcomes were achieved across various classification function shapes, with datasets containing 5000 points and five agents. Figures 1.10, 1.11, 1.12, 1.13, and 1.14 illustrate the impact of α parameter. A higher step-size, despite causing instability during training, yielded higher classification accuracy (see Table 1.2). Conversely, a lower step-size provided more stable training but resulted in a slightly higher classification error, though still excellent.



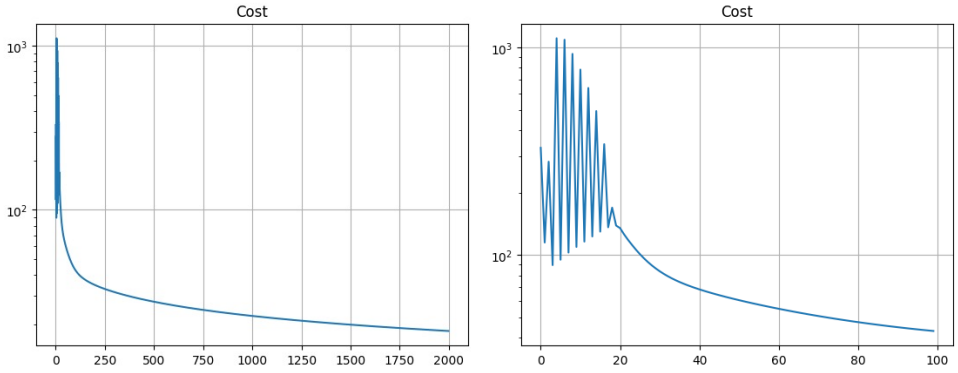
(a) Gradient Estimates, α =5e-2 (b) Gradient Estimates, α =8e-1

Figure 1.10: Local Gradients comparison for different α, random split



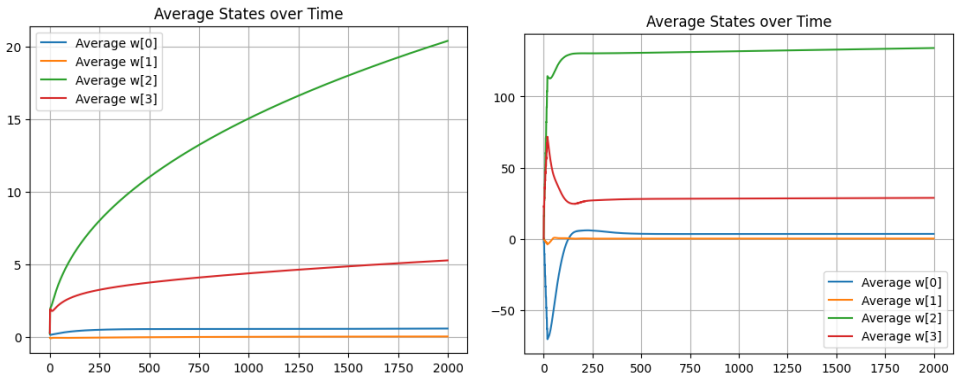
(a) Overall Gradient Norm, α =5e-2 (b) Overall Gradient Norm, α =8e-1

Figure 1.11: Global Gradients comparison for different α, random split



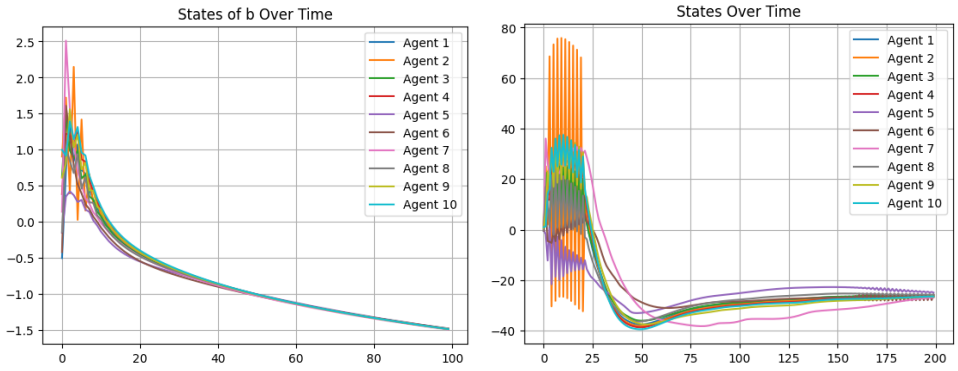
(a) Evolution of Cost, α =5e-2 (b) Evolution of Cost, α =8e-1

Figure 1.12: Costs comparison for different α, random split



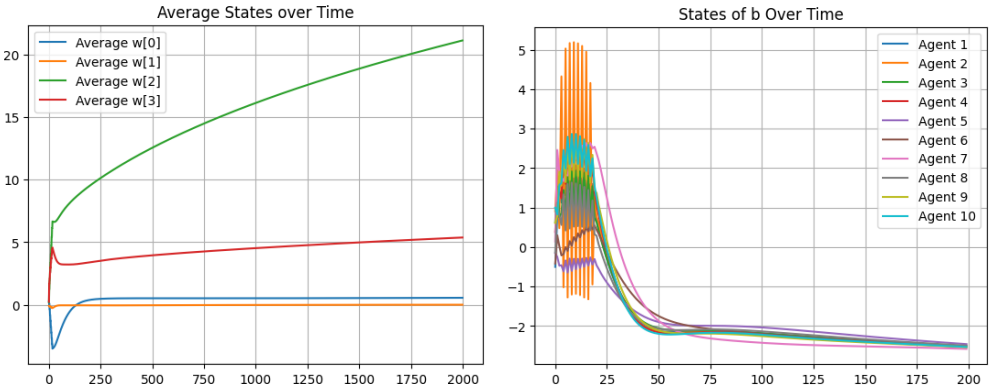
(a) W learning, α =5e-2 (b) W learning, α =8e-1

Figure 1.13: W states comparison for different α, random split



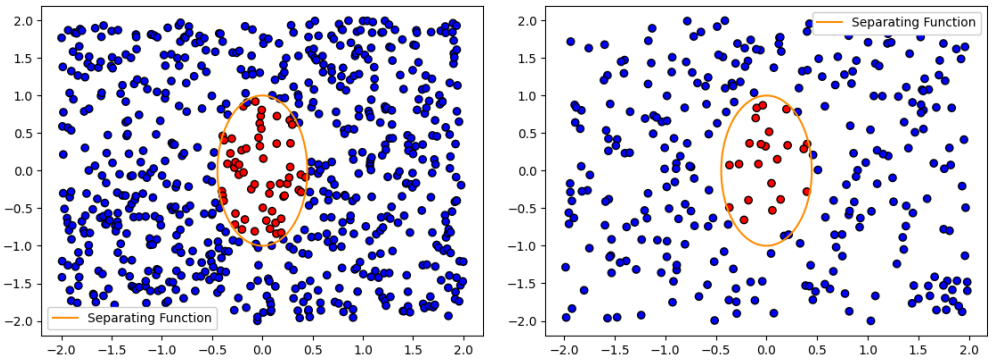
(a) b learning, α =5e-2 (b) W learning, α =8e-1

Figure 1.14: b state comparison for different α, random split



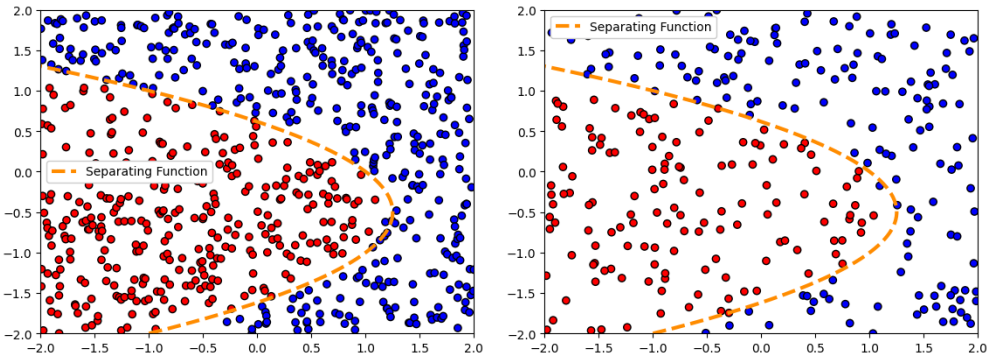
(a) W state evolution (b) b state evolution

Figure 1.15: States for sequential split



(a) Classification on Training set (b) Classification on Test set

Figure 1.16: Vertical Ellipse



(a) Classification on Training set (b) Classification on Test set

Figure 1.17: Parabola

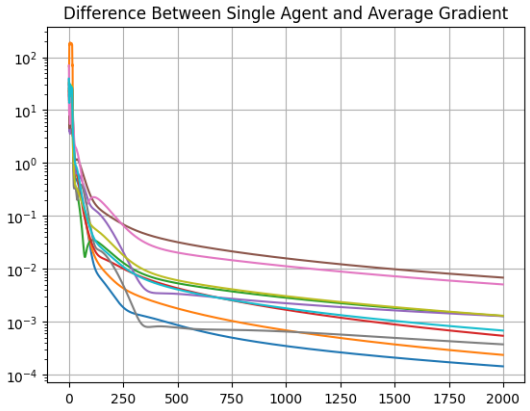


Figure 1.18: Gradient Tracking Error, $\alpha = 5e^{-2}$

Results with sequential (Figure 1.9a, $\alpha = 5e^{-2}$) and random splits (Figures 1.13 and 1.14) showed minor differences in parameter evolution, but similar performance metrics. Both training and testing sets were used to assess performance, as shown in Figures 1.16 and 1.17. The distributed classification achieved comparable results to the centralized approach, though it required more computational resources and time, especially with larger datasets.

Metric	**Class. Err.**	**Accuracy**	**Precision**	**Recall**	**F1**
Train_Set ($\alpha = 8e^{-1}$)	0.0%	1.0	1.0	1.0	1.0
Test_Set ($\alpha = 8e^{-1}$)	0.0%	1.0	1.0	1.0	1.0
Train_Set ($\alpha = 5e^{-2}$)	0.3%	0.997	0.996	1.0	0.998
Test_Set ($\alpha = 5e^{-2}$)	0.7%	0.993	0.992	1.0	0.996

Table 1.2: Comparison between different Learning Rates

To evaluate the algorithm's robustness, some tests have been conducted by adding noise to the testing data using a normal distribution with varying standard deviation values. The results are summarized in Table 1.3.

Metric	**Class. Err.**	**Accuracy**	**Precision**	**Recall**	**F1**
$\sigma = 0.2$	4%	0.96	0.97	0.98	0.97
$\sigma = 0.4$	12%	0.88	0.93	0.93	0.93
$\sigma = 0.6$	10%	0.90	0.94	0.95	0.94
$\sigma = 0.8$	16%	0.84	0.91	0.91	0.91

Table 1.3: Comparison between different Noise standard deviation values

Chapter 2

Task2 - Aggregative Optimization for Multi-Robot Systems

2.1 Theoretical Framework

Aggregative optimization is crucial in distributed systems, enabling agents to collectively optimize global objectives while managing local targets. Consider robots in the plane aiming to optimize their positions $z_i \in \mathbb{R}^2$, $i = 1, \dots, N$, via minimization of a local cost function $\ell_i(z_i, \sigma(z))$. Let $b \in \mathbb{R}^2$ be a global target to protect, $r_i \in \mathbb{R}^2$ be the local target for robot i, and $\sigma(z) = \frac{1}{N}\sum_{i=1}^{N} z_i$ be the robots' barycenter. The optimization problem is:

$$\min_{z_1,\dots,z_N} \sum_{i=1}^{N} \ell_i(z_i, \sigma(z)), \quad \text{with } \sigma(z) = \frac{1}{N}\sum_{i=1}^{N} \phi_i(z_i) \tag{2.1}$$

There are two global terms: $\sum_{j=1}^{N} \frac{\partial}{\partial \sigma}\ell_j(z_j, \sigma)$ and $\sigma = \frac{1}{N}\sum_{j=1}^{N}\phi_j\left(z_j^k\right)$. Since agents lack complete information about each other, computing these terms requires consensus over time. Each agent i only knows ϕ_i and ℓ_i, and maintains estimates z_i^k of $z_i^\star$, s_i^k of $\sigma\left(z^k\right)$, and v_i^k of $\sum_{j=1}^{N}\nabla_2\ell_j\left(z_j^k, \sigma\left(z^k\right)\right)$. The gradient tracking algorithm applied to aggregative optimization is:

$$\begin{aligned} z_i^{k+1} &= z_i^k - \alpha\left(\nabla_1\ell_i\left(z_i^k, s_i^k\right) + \nabla\phi_i\left(z_i^k\right)v_i^k\right) \\ s_i^{k+1} &= \sum_{j\in\mathcal{N}_i} a_{ij}s_j^k + \phi_i\left(z_i^{k+1}\right) - \phi_i\left(z_i^k\right) \\ v_i^{k+1} &= \sum_{j\in\mathcal{N}_i} a_{ij}v_j^k + \nabla_2\ell_i\left(z_i^{k+1}, s_i^{k+1}\right) - \nabla_2\ell_i\left(z_i^k, s_i^k\right) \end{aligned} \tag{2.2}$$

with initial conditions given as: $z_i^0 \in \mathbb{R}^{n_i}$, $s_i^0 = \phi_i\left(z_i^0\right)$ and $v_i^0 = \nabla_2\ell_i\left(z_i^0, s_i^0\right)$.

2.2 Problem Set-Up

2.2.1 Graph Representation

The network among robots is represented using a connected graph, with nodes for robot and edges for communication links. The adjacency matrix `Adj` specifies direct connections between robots, and Metropolis-Hastings weights `WW` are assigned to edges based on the network structure, determining the influence of neighboring robots during optimization.

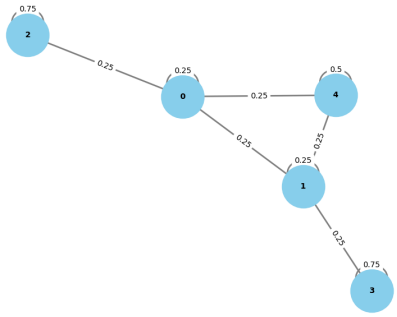


Figure 2.1: Graph Connectivity, 5 agents

2.2.2 Cost function

The method's effectiveness hinges on the cost function's structure, which dictates agent behavior. It reads:

$$\ell_i\left(z_i, \sigma(z)\right) = \gamma_{r_{lt}} \left\|z_i - r_i\right\|^2 + \gamma_{bar} \left\|\sigma(z) - b\right\|^2 + \gamma_{agg/rep} \left\|\left|z_i - \sigma(z)\right\|^2 + obst\_pot \tag{2.3}$$

with the following components:

- **Local Target Attraction Term** ($\gamma_{r_{lt}} \times d_{robot\_target}$), based on the distance of each robot to its local target and weighted by $\gamma_{r_{lt}}$;
- **Barycenter Goal Term** ($\gamma_{\text{bar}} \times d_{\text{robot\_barycenter}}$), based on the distance between the barycenter and global target, weighted by γ_{agg};
- **Barycenter Attraction/Repulsion Term** ($\gamma_{\text{agg/rep}} \times d_{\text{barycenter\_target}}$), based on the distance between the robot and the barycenter. Barycenter's attraction is weighted by γ_{agg}, whereas barycenter's repulsion can be weighted by γ_{rep} with a reversed sign in the cost;
- **Potential Function** (for task 2.3), based on the distance between the robot and each obstacle point, weighted by K;

The cost function value (`FF`) is stored and plotted to monitor the algorithm's performance, together with cost's gradients with respect to agent (ZZ) and barycenter (tracker SS)'s positions. This latter gradient is kept into account by the tracker VV, computed by the algorithm over the iterations.

2.2.3 Obstacle Avoidance

The potential fields method enables robots to dynamically adjust their paths in high-dimensional environments when obstacles are detected. The potential function is defined as $U : \mathbb{R}^m \to \mathbb{R}$ and the gradient of U with respect to q is given by:

$$\nabla U(q) = \begin{pmatrix} \frac{\partial U(q)}{\partial q_1} \\ \vdots \\ \frac{\partial U(q)}{\partial q_n} \end{pmatrix}$$

In subsequent chapters, this method will be presented for obstacle avoidance in corridor walls. These walls are discretized into finite coordinate points, each associated with a repulsive potential field, marking unsafe areas for the robots' paths. This approach ensures robust and adaptive obstacle handling, crucial for maintaining safe and efficient navigation.The repulsive force at each obstacle increases as the robot approaches, starting from a certain safety distance from the target. U_{rep} is defined as :

$$U_{\text{rep},i}(q) = \begin{cases} \frac{1}{2} K_{r_i} \left(\frac{1}{d_i(q)} - \frac{1}{q^*} \right)^2 & \text{if } d_i(q) < q^* \\ 0 & \text{otherwise} \end{cases}$$

The associated gradient is:

$$\nabla U_{\text{rep},i}(q) = \begin{cases} K_{r_i} \left(\frac{1}{q^*} - \frac{1}{d_i(q)} \right) \frac{\nabla d_i(q)}{d_i(q)^2} & \text{if } d_i(q) \leq q^* \\ 0 & \text{if } d_i(q) > q^* \end{cases}$$

This method effectively guides robots toward their goals while avoiding collisions. However, a significant issue with potential fields is the occurrence of local minima, where a robot can become trapped in a position that is neither at the goal nor moving towards it. Future developments aim to overcome these limitations by engineering more advanced algorithms.

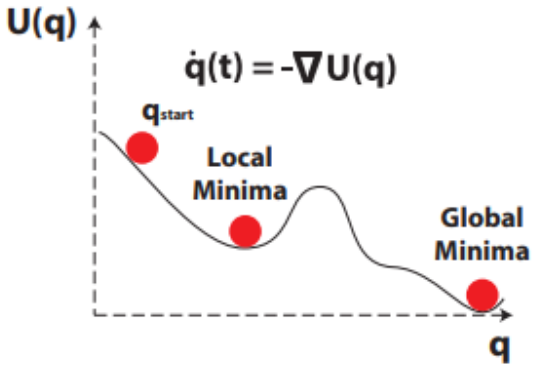


Figure 2.2: Potential Functions, An Overview

2.3 Task 2.1

Task 2.1 is designed to define and achieve both global and local goals for a team of robots, simulating a surveillance scenario. Each robot, starting from different initial conditions, aims to fulfill two primary objectives: staying close to its assigned local target (RR) and protecting a designated global target (b), all while maintaining cohesion within the formation. This dual-objective approach ensures that individual robots are effective in their local tasks while contributing to the collective mission. Aggregative optimization is employed to achieve such objectives through an iterative process, where each robot updates its position based on both local and global considerations. The cost function, which the algorithm aims to minimize, incorporates weighted terms for each objective, with specific gains assigned to local target ($\gamma_{r_{\text{lt}}}$), formation cohesion (γ_{agg}), and global target protection (γ_{bar}).

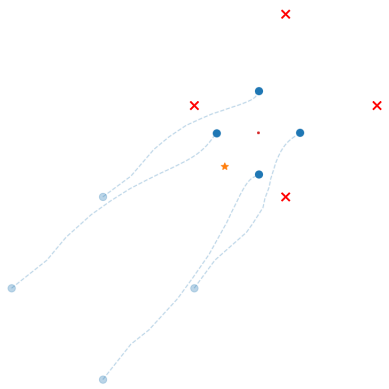


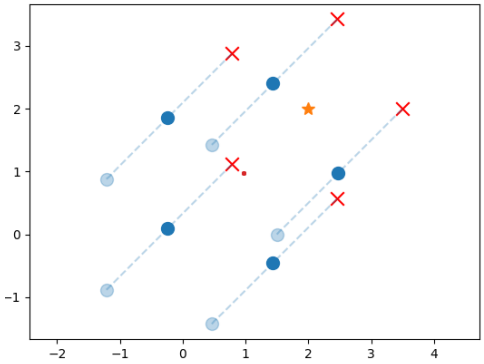
Figure 2.3: Task 2.1, Example Animation

2.3.1 Update local targets

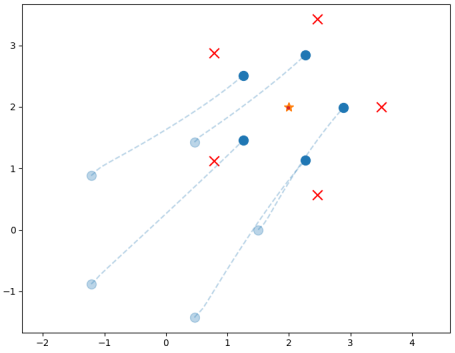
In this optional part of the task, local targets are dynamic, changing over time according to user-specified behaviors. The local targets are updated at fixed intervals, effectively altering the optimization problem at each update. These updates can follow various patterns, such as horizontal, vertical, and diagonal movements, determined by parameters such as *targets_motion_type*, *targets_shift_x*, and *targets_shift_y*. The algorithm is capable of handling such a dynamic environment, as it re-evaluates and minimizes the cost function at each iteration, taking into account the new positions of the local targets. This causes the cost and gradient norms to exhibit periodic increases and decreases, reflecting the continuous adjustment process as the targets move. The dynamic updating of local targets is controlled by *targets_rate_change*, which determines how frequently the targets are updated.

2.3.2 Simulations

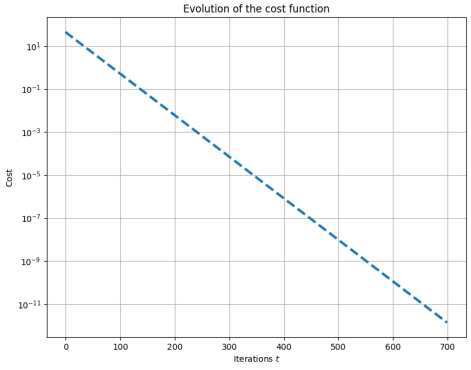
The following figures demonstrate the effectiveness of the aggregative optimization algorithm in a robot surveillance task. Fixed parameters include dt=10, NN=5, n_z=2, max_iters=700, initial_avg=(0,0), radius=3.0, gain=0.5, target_avg=(2,2), b=(2,2), stepsize=1e-2, $\gamma_{r_{\text{lt}}}$=0.9, and γ_{bar}=0.2, with γ_{agg} varying between the plots. In the static target scenario, the optimization algorithm demonstrates actual minimization of the cost function over iterations, ensuring that agents remain close to their local targets while maintaining formation and protecting the global target.



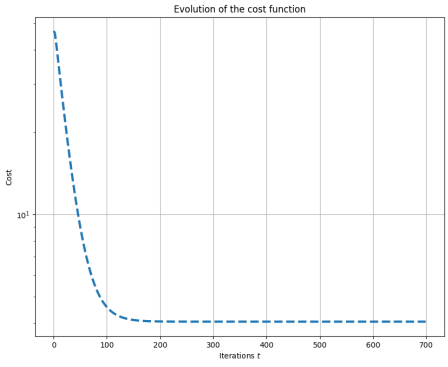
(a) 2D Animation, γ_{agg}=0



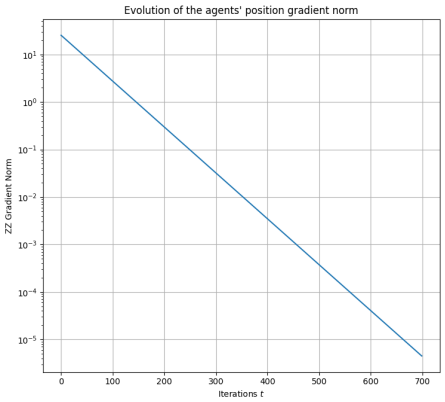
(b) 2D Animation, γ_{agg}=0.6



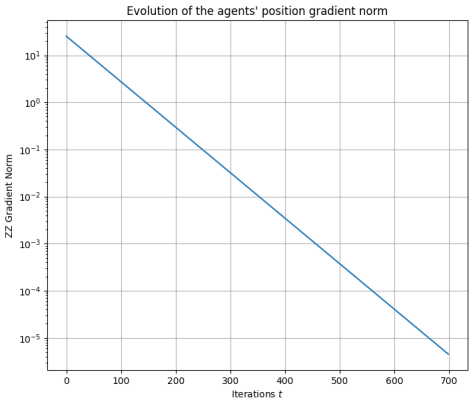
(a) Evolution of Cost, γ_{agg}=0



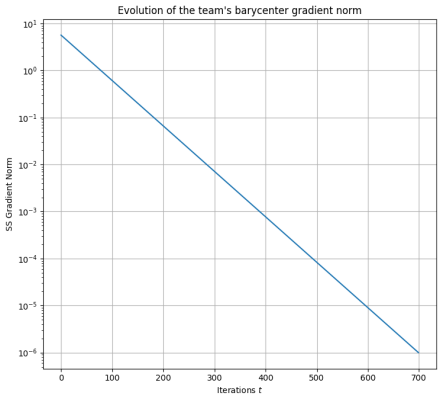
(b) Evolution of Cost, γ_{agg}=0.6



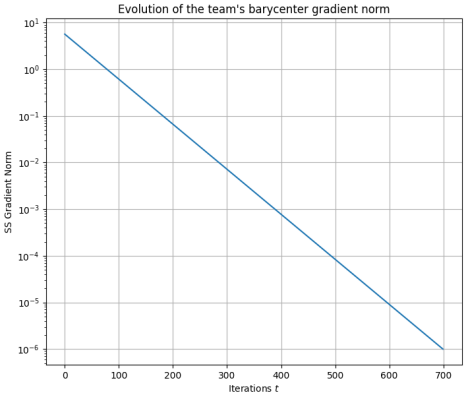
(a) Agents Gradient Norm, γ_{agg}=0



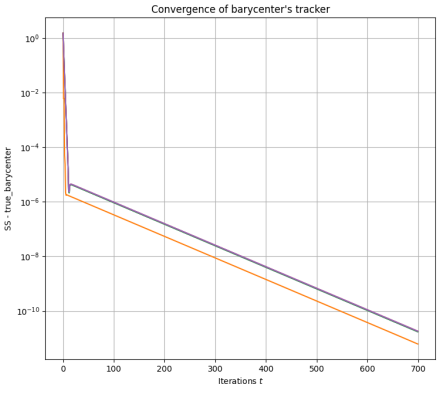
(b) Agents Gradient Norm, γ_{agg}=0.6



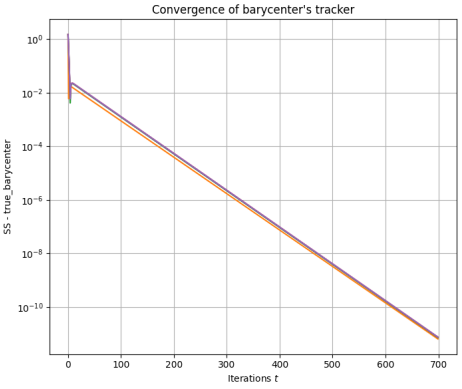
(a) Barycenter Gradient Norm, γ_{agg}=0



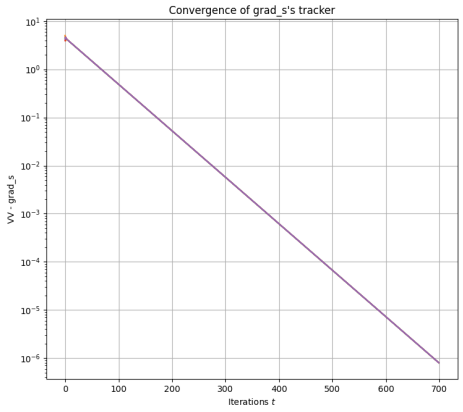
(b) Barycenter Gradient Norm, γ_{agg}=0.6



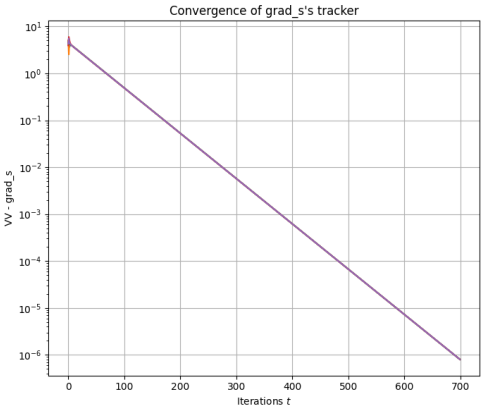
(a) SS Convergence, γ_{agg}=0



(b) SS Convergence, γ_{agg}=0.6

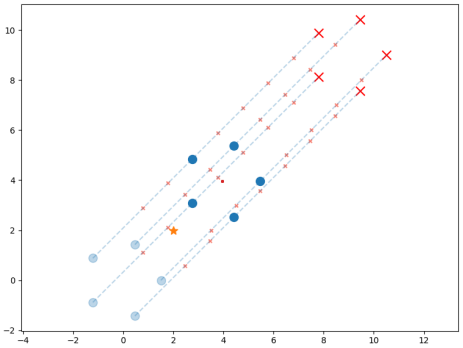


(a) VV Convergence, γ_{agg}=0

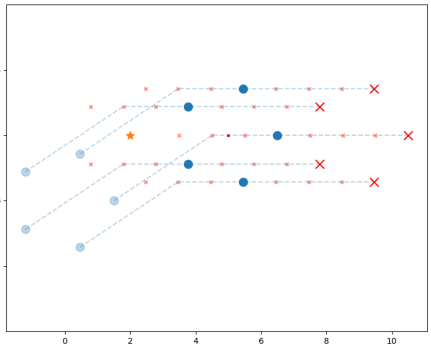


(b) VV Convergence, γ_{agg}=0.6

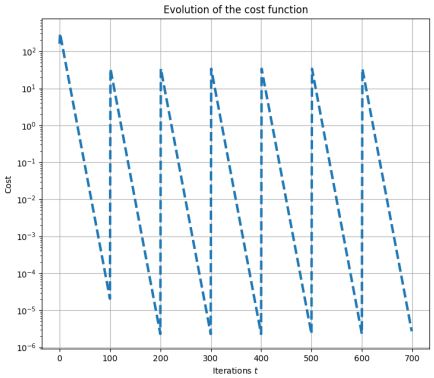
In the subsequent simulations, local targets move in time, updated at each iteration. Parameters don't change with respect to the previous set of simulations, with the only difference in γ_{agg}, which is always equal to zero. The targets move based on the parameters *targets_rate_change*=100, *targets_shift_x*=1, and *targets_shift_y*=1, with *targets_motion_type* being either *vert*, *hor* or *diag*. For simplicity, only horizontal and vertical movements are shown in the following plots:

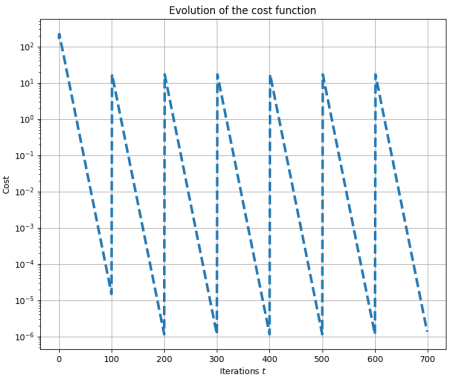


(a) 2D Animation - Diagonal Update

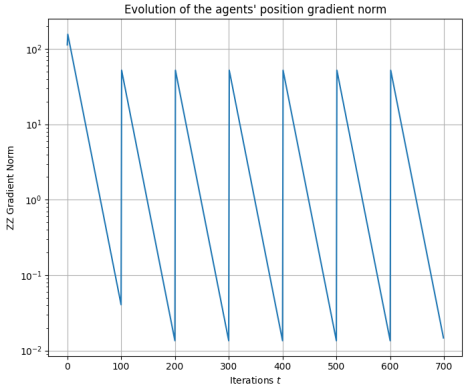


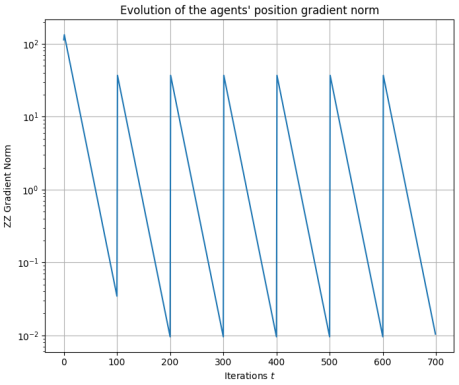
(b) 2D Animation - Horizontal Update



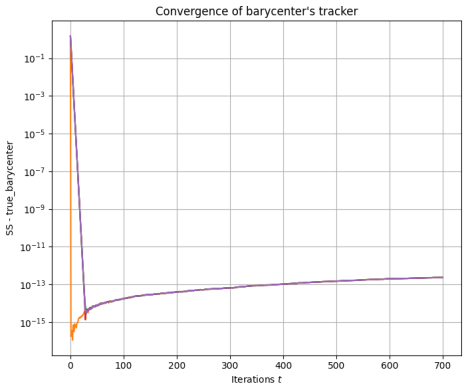


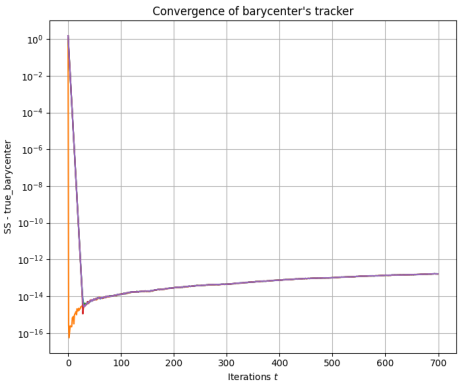
(a) Evolution of Cost, Diagonal Update (b) Evolution of Cost, Horizontal Update





(a) Agents Gradient Norm, Diagonal Update (b) Agents Gradient Norm, Horizontal Update





(a) SS Tracker Convergence, Diagonal Update (b) SS Tracker Convergence, Horizontal Update

2.4 Task 2.3

The objective of Task 2.3 is for agents to navigate through a corridor from the left to the right side of the environment, ultimately reaching their final goals. A primary challenge in this task is to prevent agents from colliding with the corridor walls. Two main approaches are employed to address this:

- **Potential Functions**: Each agent is assigned a potential function that increases as the agent approaches the corridor walls. The agents are then guided to move in a direction that decreases this potential function, thereby avoiding collisions.
- **Projected Aggregative Tracking**: This approach modifies the solution of the optimization problem if infeasible (i.e., if it would result in agents colliding with the corridor walls), by projecting it into the nearest feasible set, i.e., within the corridor's bounds.

2.4.1 Potential Functions for Corridor Navigation

The potential fields method, as detailed in Subsection 2.2.3, guides robots through the corridor, avoiding wall collisions using potential functions as repulsive forces. These functions are defined along points on the corridor walls and a line perpendicular to the entrance to ensure robots navigate through the corridor. Distances between each robot and obstacle points are evaluated at each iteration, and as robots approach these points and breach a safe distance, the potential function's effect intensifies, influencing the cost and its gradient. The repulsive forces keep robots away from the walls, while the local targets' attractive effects steer them towards their goals, ensuring coordinated and collision-free corridor's traversal.

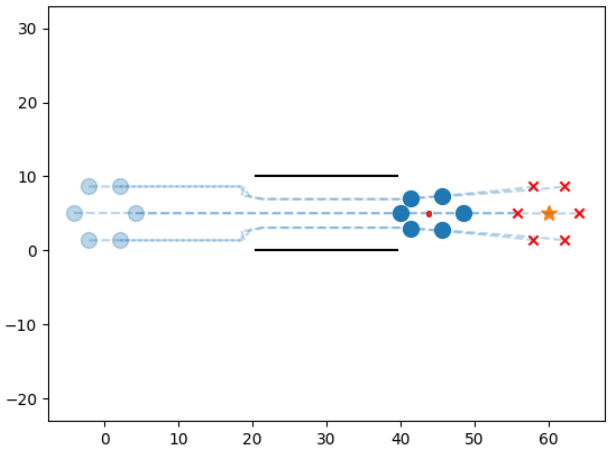


Figure 2.14: Task 2.1, Example Animation

2.4.2 Projected Aggregative Tracking (online)

The Projected Aggregative Tracking (PAT) optimization algorithm handles online scenarios where agents may be constrained to remain within a feasible set. The goal is to generate a sequence $\left\{z_i^k\right\}_{k\in\mathbb{N}}$ that tracks the solution $z^{k,\star} = \left(z_1^{k,\star}, \ldots, z_N^{k,\star}\right)$ of the optimization problem:

$$\begin{aligned} \min_{z=(z_1,\ldots,z_N)} & \sum_{i=1}^{N} \ell_i^k\left(z_i, \sigma^k(z)\right) \\ \text{subj. to } & z_i \in Z_i^k, \quad \forall i = 1, \ldots, N \end{aligned} \tag{2.4}$$

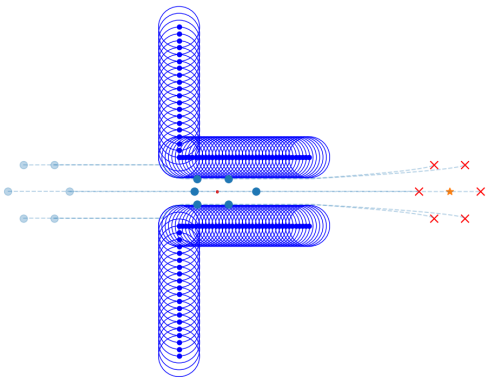
The algorithm updates each agent's position as follows:

$$\begin{aligned} \tilde{z}_i^k &= P_{Z_i^k}\left[z_i^k - \alpha\left(\nabla_1 \ell_i^k\left(z_i^k, s_i^k\right) + \nabla\phi^k\left(z_i^k\right) v_i^k\right)\right] \\ z_i^{k+1} &= z_i^k + \delta\left(\tilde{z}_i^k - z_i^k\right) \\ s_i^{k+1} &= \sum_{j\in\mathcal{N}_i} a_{ij} s_j^k + \phi_i^{k+1}\left(z_i^{k+1}\right) - \phi_i^k\left(z_i^k\right) \\ v_i^{k+1} &= \sum_{j\in\mathcal{N}_i} a_{ij} v_j^k + \nabla_2 \ell_i^{k+1}\left(z_i^{k+1}, s_i^{k+1}\right) - \nabla_2 \ell_i^k\left(z_i^k, s_i^k\right) \end{aligned} \tag{2.5}$$

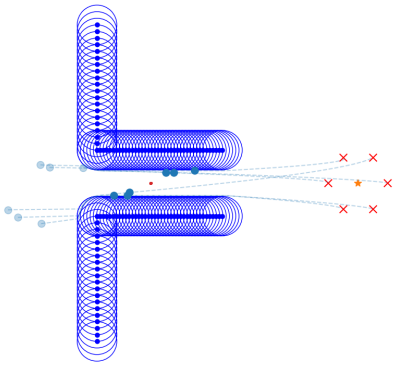
where $P_{Z_i^k}$ is the projection onto the feasible set Z_i^k. The PAT method distinguishes itself from conventional potential functions by dynamically adjusting robot trajectories through optimization and projection techniques, rather than solely relying on repulsive forces near obstacles. This approach prioritizes real-time feasibility adjustments and adaptive responsiveness to varying conditions. While its implementation closely resembles the classical aggregative tracking approach, it includes an additional step of projecting robots onto the feasible set to confine them within corridor boundaries. PAT's implementation involves three distinct optimization phases tailored to different path segments, each with specific parameter configurations. Initially, robots are guided from their starting positions to the corridor entrance using local target goals, ensuring a smooth initial movement. Subsequently, the second optimization problem takes over, moving robots through the corridor toward local targets while continuously applying online projections to prevent collisions and maintain safe navigation. Finally, robots are directed from the corridor exit to their ultimate target positions, ensuring a seamless transition. At the end of each optimization phase, local information on costs, gradients, and robot trajectories is integrated into overall variables, allowing agents to adjust their behaviors based on the specific optimization segment being addressed. This decoupled structure ensures precise and efficient task handling, distinct from traditional potential function approaches.

2.4.3 Simulations

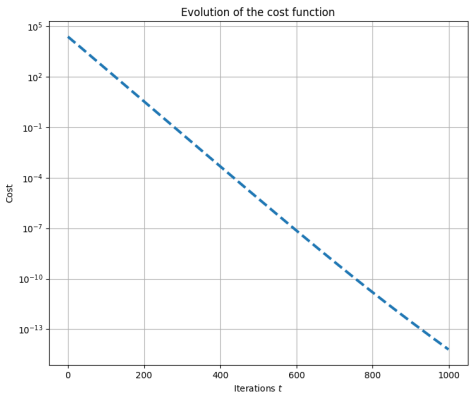
The following simulations demonstrate the effectiveness of potential functions for corridor navigation, in a unique optimization process. By employing repulsive potential fields along corridor walls and entrances, robots safely navigate towards their targets while minimizing the overall cost and gradients. Parameters utilized for simulation generation include: dt=10, NN=6, n_z=2, MAXITERS=1000, corr_length=19, corr_width=10, corr_pos=(30,5), initial_avg=(0,5), radius=5, gain=0.7, target_avg=(60,5), $\gamma_{r_{\text{lt}}}$=0.9, γ_{agg}=0, γ_{bar}=0.2, γ_{rep}=0, q_star=5, K=1000, α=10^{-2}.



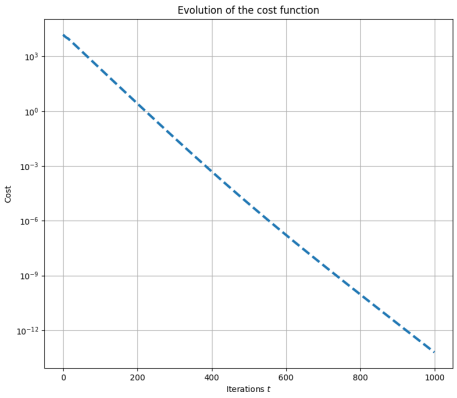
(a) 2D Animation, *circular_init*



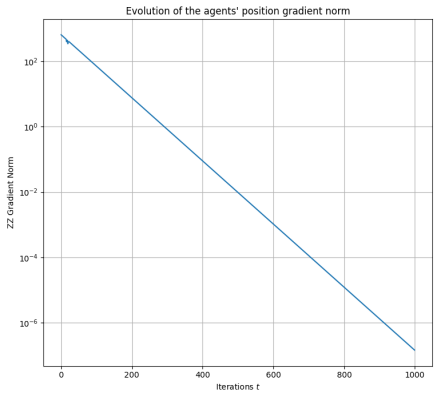
(b) 2D Animation, *random_init*



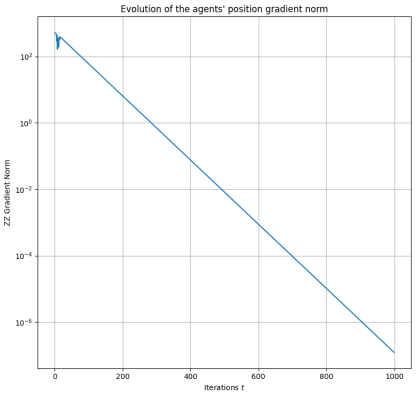
(a) Evolution of Cost, *circular_init*



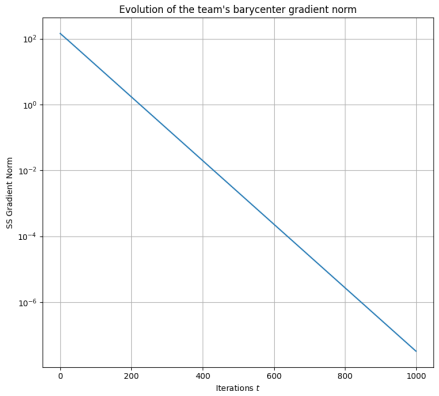
(b) Evolution of Cost, *random_init*



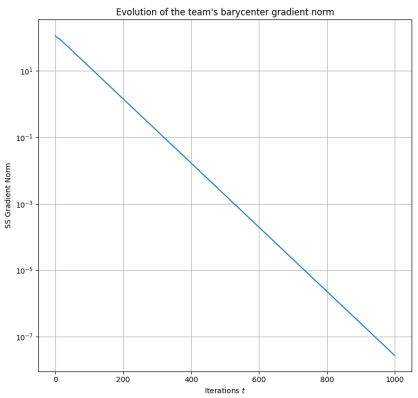
(a) Agents Gradient Norm, *circular_init*



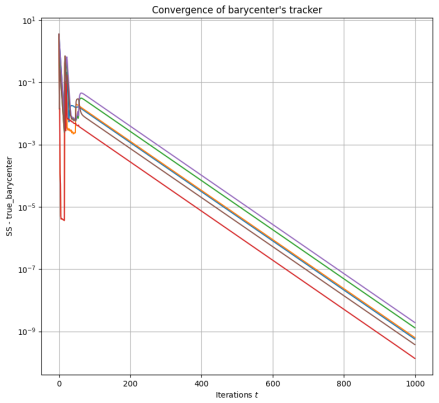
(b) Agents Gradient Norm, *random_init*



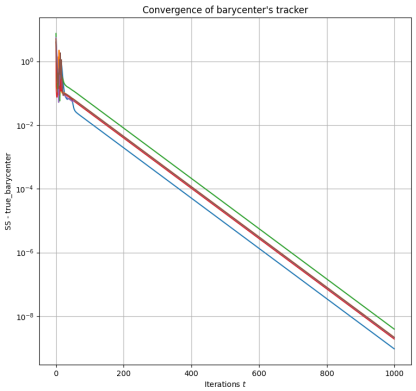
(a) Barycenter Gradient Norm, *circular_init*



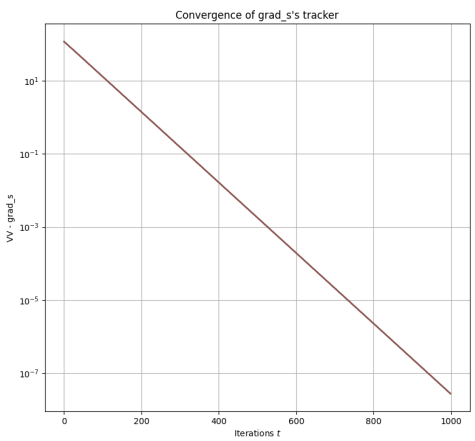
(b) Barycenter Gradient Norm, *random_init*



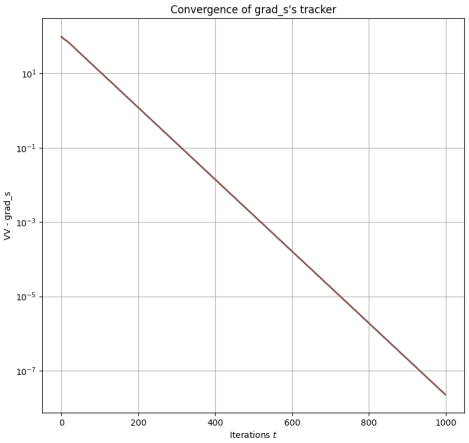
(a) SS Convergence, *circular_init*



(b) SS Convergence, *random_init*

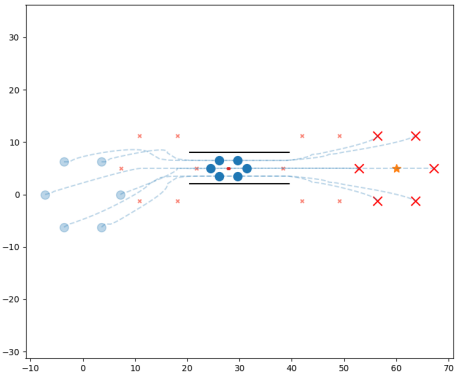


(a) VV Convergence, *circular_init*

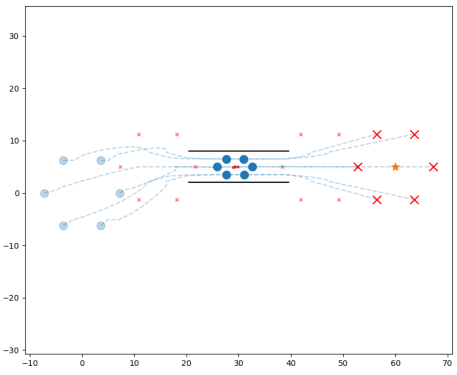


(b) VV Convergence, *random_init*

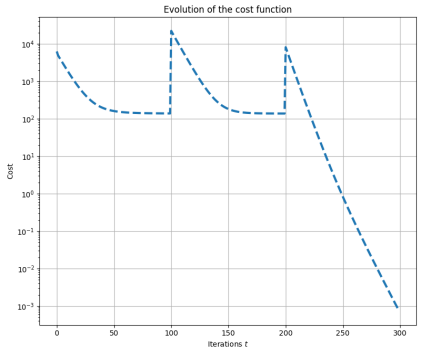
The following simulations illustrate the effectiveness of PAT in managing different optimization problems, leading to effective outcomes. To generate the subsequent plots, the following parameters were utilized: delta=0.8, dt=1, walls_margin=1.5, NN=6, n_z=2, MAXITERS=300, corr_length=19, corr_width=6, corr_pos=(30,5), initial_avg=(0,0), radius=9, target_avg=(60,5), with changing step-size between the simulations.



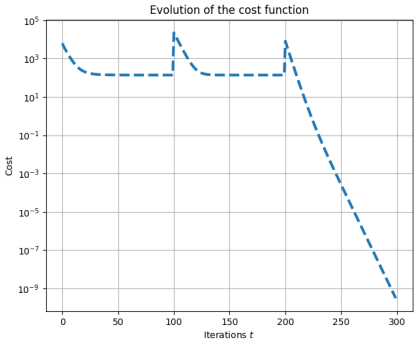
(a) PAT 2D Animation, α=$1e^{-2}$



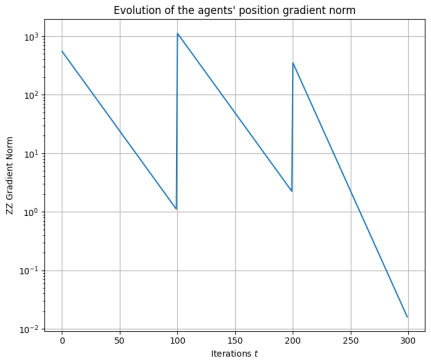
(b) PAT 2D Animation, α=$2e^{-2}$



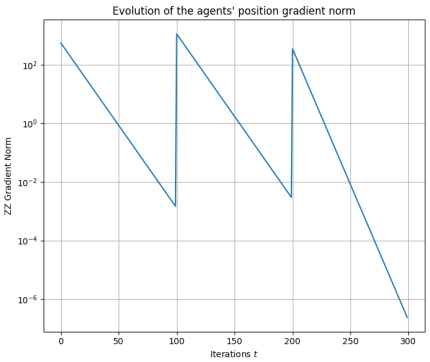
(a) Evolution of Cost, α=$1e^{-2}$



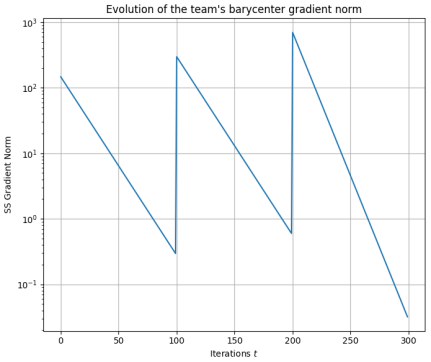
(b) Evolution of Cost, α=$2e^{-2}$



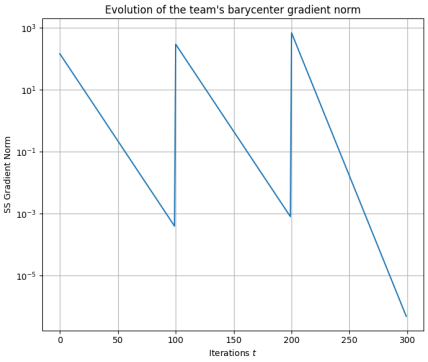
(a) Agents Gradient Norm, α=$1e^{-2}$



(b) Agents Gradient Norm, α=$2e^{-2}$



(a) Barycenter Gradient Norm, α=$1e^{-2}$



(b) Barycenter Gradient Norm, α=$2e^{-2}$

2.5 ROS2 Implementation

This section outlines the ROS2 implementation for the multi-robot tasks discussed in task 2.1 (robot surveillance) and task 2.3 (corridor navigation), within a real-time distributed environment.

2.5.1 Environment Setup

The system architecture relies on ROS2, where each robot functions as a node. These nodes interact within the environment, exchanging information and executing optimization algorithms to achieve collective objectives. Object-oriented programming principles are employed, encapsulating functionality within classes and methods. Each robot is an instance of the Agent class, featuring attributes like `agent_id`, `ZZ_init`, `neigh`, and methods such as `listener_callback`, `publish_message`, and `timer_callback`. Project parameters are configured using yaml files (`parameters_task_2_1.yaml` and `parameters_task_2_3.yaml`), offering structured parameter management. The launch file orchestrates environment setup, handling parameter extraction, agent initialization, corridor point creation, and node launching. It allows the generation of a connected graph to ensure agent communication via an adjacency matrix with Metropolis-Hastings weights.

2.5.2 Tasks Implementation

The `the_agent.py` file implements a distributed aggregative optimization algorithm for the multi-robot system. Each agent subscribes to messages from neighbors, ensuring synchronization and collaborative optimization. Agents communicate via message passing, exchanging essential information like ID, iteration count, decision variables, and cumulative gradients. This setup enables agents to coordinate their actions and optimize the cost function collectively. They also write their data to individual CSV files for centralized visualization and analysis. In task 2.1, robots perform surveillance using the aggregative optimization algorithm, detailed in chapter 2.3. Each robot aims to stay close to local targets while maintaining formation around a global target. For task 2.3, detailed in chapter 2.4 robots navigate the corridor using potential functions exclusively, dynamically adjusting paths to avoid obstacles and ensure smooth navigation. The ROS2 implementation integrates potential functions for task optimization and collision avoidance, showcasing its versatility and efficiency in multi-robot coordination as a real-time operating system for distributed algorithms.

2.5.3 RVIZ Visualization

The `Visualizer.py` node provides a useful visualization of ROS2 project's tasks, by monitoring and analyzing the algorithm's performance in real-time. It subscribes to agents' topics `/topic_<agent_id>` to receive messages about their positions and goals via the `listener_callback()` function. Afterward, `publish_data()` routine properly updates real-time information. This latter routine constructs visualization markers using the `Marker` message type, to dynamically update RVIZ to reflect the latest agent positions (balls) and goals (cubes), leading to an effective real-time visualization of the algorithm.

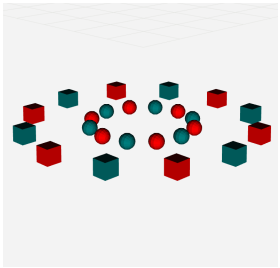


Figure 2.25: Example RVIZ Animation, 12 Agents

2.5.4 Centralized Visualization

The `centralized_animation.py` script visualizes the multi-robot system behavior over time once all the nodes have finished their computations. It reads gradient, cost and position of each agent over the iterations from CSV files, through Pandas library. Afterward, it generates gradients and costs plots, as well as 2D and 3D animations, to illustrate real robot trajectories and interactions within the corridor. A 3D animation example is shown in figure 2.26, where agents' trajectories (thick curves) and barycenter estimates' trajectories (thin curves) are plotted over iterations, the latter reaching consensus over the true barycenter at the last iteration.

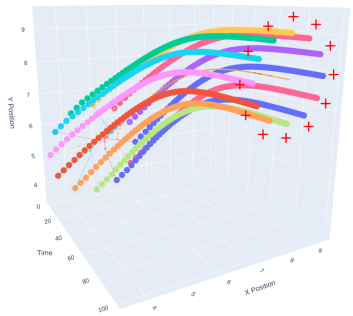
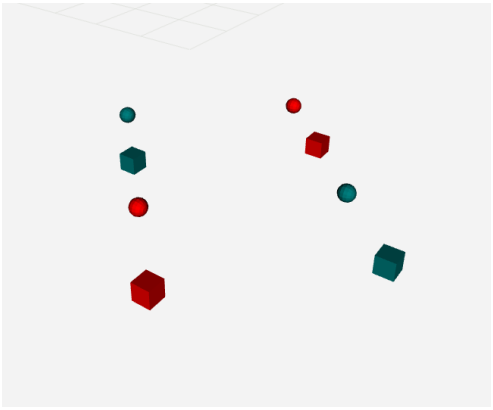


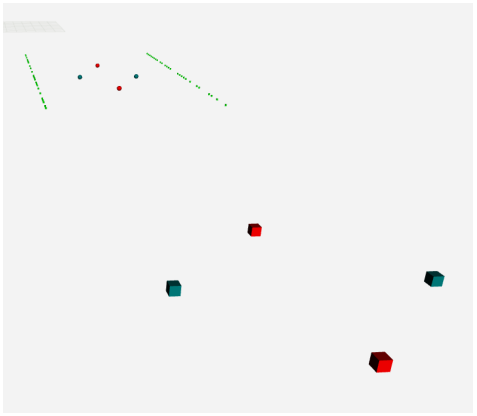
Figure 2.26: Example Centralized 3D Animation, 12 Agents

2.5.5 Simulations

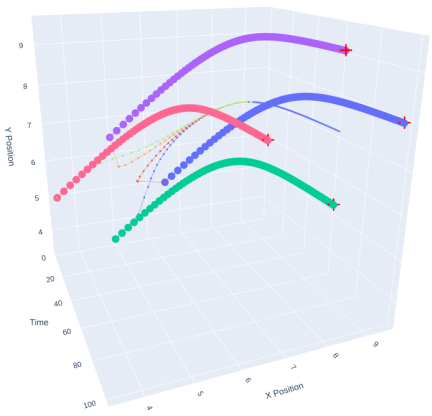
The following figures show the aggregative optimization algorithm applied to ROS2 framework. In particular, robot surveillance (task 2.2.1) and corridor navigation (task 2.2.3) tasks are performed by the MAS, according to the rules previously detailed in chapters 2.3 and 2.4, respectively. Fixed parameters include dt=3, NN=4, n_z=2, visu_frequency=200, gain=0.5, timer_period= 0.5, p_ER=0.3, $\alpha = 1e^{-2}$,$\gamma_{r_{\text{lt}}}$=0.9, γ_{bar}=0.5 and γ_{agg}=0. For multi-robot surveillance task, the following parameters have been properly fixed: max_iters=100, init_avg=(5,5), radius=3.0, target_avg=(8,8), b=(8,8). For corridor navigation's task, the following parameters have been decided: max_iters=250, initial_avg=(0,5), radius=7.0, target_avg=(60,5), b=(60,5), K=1000, q_star=5, corr_length=19, corr_width=8, corr_pos=(30,5), corr_bound_offset=5, walls_margin=0.7 and num_corr_points=100.



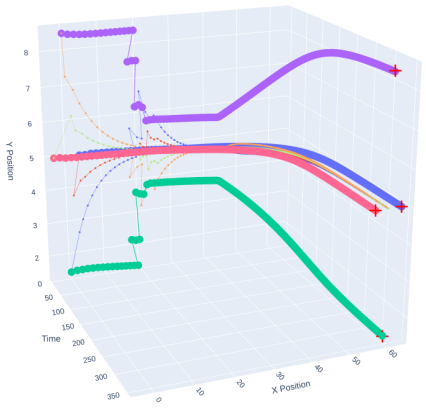
(a) Task 2.2.1, RVIZ Animation



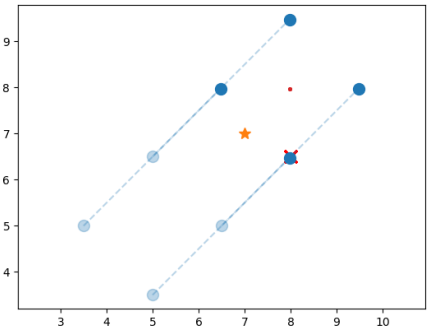
(b) Task 2.2.3, RVIZ Animation



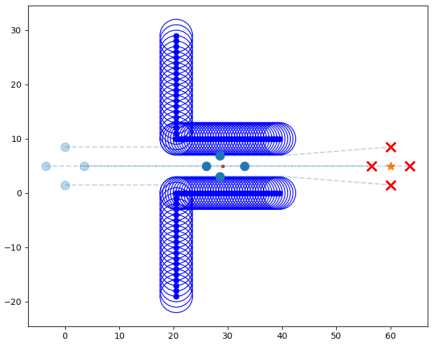
(a) Task 2.2.1, 3D Animation



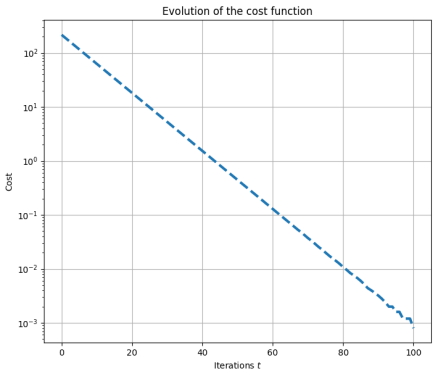
(b) Task 2.2.3, 3D Animation



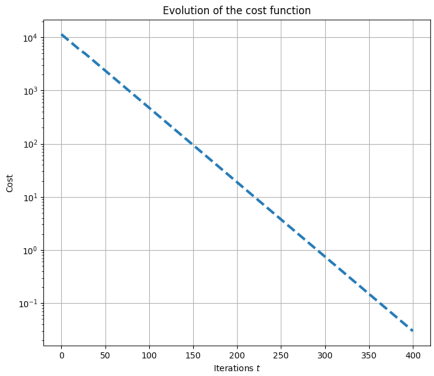
(a) Task 2.2.1, 2D Animation



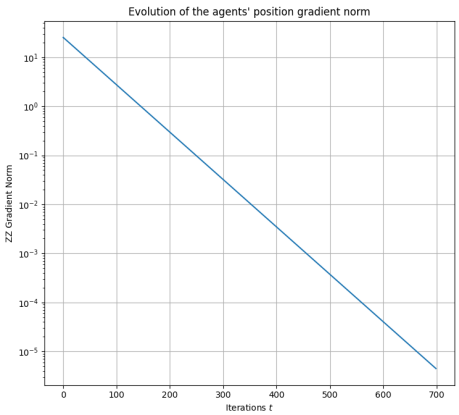
(b) Task 2.2.3, 2D Animation



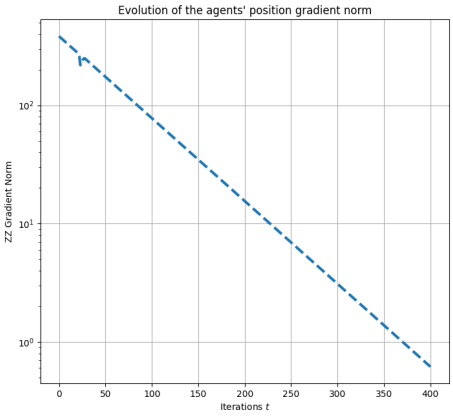
(a) Task 2.2.1, Evolution of Cost



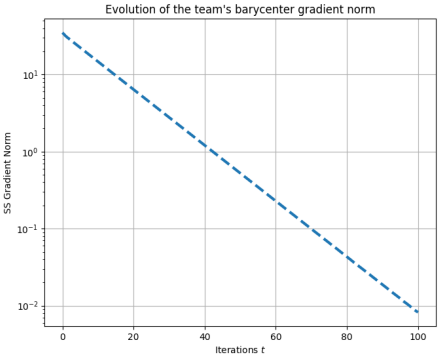
(b) Task 2.2.3, Evolution of Cost



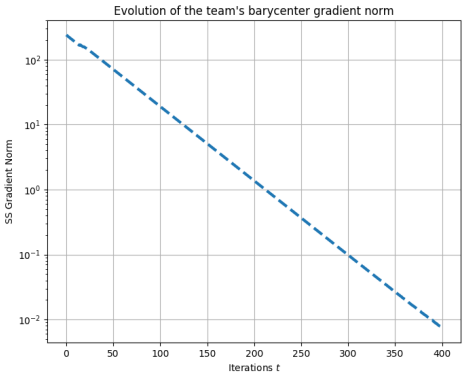
(a) Task 2.2.1, Agents Gradient Norm



(b) Task 2.2.3, Agents Gradient Norm



(a) Task 2.2.1, Barycenter Gradient Norm



(b) Task 2.2.3, Barycenter Gradient Norm

Conclusions

In conclusion, this report has outlined the successful implementation of two primary tasks in distributed autonomous systems, with distinct goals and applications. The first task focused on distributed classification and optimization, utilizing the gradient tracking algorithm to achieve efficient data classification and optimization across a distributed network. This application holds promise for real-world scenarios, such as data analytics and learning applications for companies, where distributed classification is crucial for processing real data. Meanwhile, the second task addressed the control of multi-robot systems in ROS2, employing the Aggregative Tracking framework to maintain formation and navigate towards targets collaboratively. This task could become a foundational step towards the development of a multi-robot system capable of performing house exploration, particularly in challenging scenarios like navigating through narrow or broken passages. Such scenarios are relevant in search and rescue applications, where robots must collaborate to traverse tight spaces and access areas inaccessible to humans. By successfully executing these tasks within a Python environment, this report highlights the efficacy and versatility of consensus-based distributed algorithms in addressing complex problems within distributed autonomous systems.

Algorithms Appendix

1: **procedure** GT($iterations, N, A, Adj, alpha, a, b$)
2: Initialize $Z, S, cost$
3: **for** $i \leftarrow 0$ to $N - 1$ **do**
4: Compute $S[0, i] \leftarrow$ grad_quadratic($Z[0, i], a[i], b[i]$)
5: **end for**
6: **for** $k \leftarrow 0$ to $iterations - 2$ **do**
7: **for** $i \leftarrow 0$ to $N - 1$ **do**
8: Compute $neighbors \leftarrow$ nonzero elements of $Adj[i]$
9: Update $Z[k + 1, i], S[k + 1, i]$ with their own and neighbors' values
10: Update $Z[k + 1, i]$ with gradient descent step
11: Compute $nabla\_new \leftarrow$ grad_quadratic($Z[k + 1, i], a[i], b[i]$)
12: Compute $nabla\_old \leftarrow$ grad_quadratic($Z[k, i], a[i], b[i]$)
13: Update $S[k+1, i]$ with the difference of new and old gradients
14: Update $cost[k]$ with the quadratic cost at $Z[k, i], a[i], b[i]$
15: **end for**
16: **end for**
17: **return** $Z, cost$
18: **end procedure**

```
1: procedure CGM(D, p, step_size, max_iterations, min_grad_norm)
2:     Initialize q ← len(φ(D[0]))
3:     Initialize w ← random values of size q
4:     Initialize b ← random value
5:     Initialize costs ← empty list
6:     Initialize grad_norms ← empty list
7:     for k ← 0 to max_iterations − 1 do
8:         Compute loss, grad_w, grad_b ← logistic_regression_loss(w, b, D, p)
9:         Compute grad_norm ← √(∑ grad_w² + grad_b²)
10:        Update w ← w − step_size × grad_w
11:        Update b ← b − step_size × grad_b
12:        Append loss to costs
13:        Append grad_norm to grad_norms
14:        if k%200 == 0 then
15:            Print progress
16:        end if
17:        if grad_norm < min_grad_norm then
18:            Print stopping message
19:            Break
20:        end if
21:    end for
22:    return w, b, costs, grad_norms
23: end procedure
```

1: **procedure** DGT($iterations, N, A, Adj, alpha, agent\_datasets$)
2: Initialize $Z, S, cost$
3: Initialize D, p
4: **for** $i \leftarrow 0$ to $N - 1$ **do**
5: Extract attributes and labels from $agent\_datasets[i]$ to $D[i], p[i]$
6: Initialize $Z[0, i, : 4], Z[0, i, 4]$ with random values
7: Compute $loss, S[0, i, : 4], S[0, i, 4] \leftarrow$ logistic_regression_loss($Z[0, i, : 4], Z[0, i, 4], D[i], p[i]$)
8: **end for**
9: **for** $k \leftarrow 0$ to $iterations - 2$ **do**
10: **for** $i \leftarrow 0$ to $N - 1$ **do**
11: Compute $neighbors \leftarrow$ nonzero elements of $Adj[i]$
12: Update $Z[k + 1, i, : 4], Z[k + 1, i, 4], S[k + 1, i, : 4], S[k + 1, i, 4]$ with their own and neighbors' values
13: Update $Z[k + 1, i, : 4], Z[k + 1, i, 4]$ with gradient descent step
14: Compute $grad\_new\_w, grad\_new\_b \leftarrow$ gradient of logistic regression loss at $Z[k+1, i, : 4], Z[k + 1, i, 4]$
15: Compute $grad\_old\_w, grad\_old\_b \leftarrow$ gradient of logistic regression loss at $Z[k, i, : 4], Z[k, i, 4]$
16: Update $S[k + 1, i, : 4], S[k + 1, i, 4]$ with the difference of new and old gradients
17: Update $cost[k]$ with the logistic regression loss at $Z[k, i, : 4], Z[k, i, 4]$
18: **end for**
19: **end for**
20: **return** $Z, cost$
21: **end procedure**

Bibliography

[1] W.K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[2] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087–1092, 1953.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.