Path Planning with Dynamic Obstacles using Velocity Obstacles and Nonlinear Model Predictive Control

Planning and Reasoning 2021/22

Leandro Maglianella - 1792507

Abstract

The overall goal of this project was the implementation and comparison of algorithms for path planning with dynamic obstacles. In particular, it was decided to focus on two of the major classical approaches: *Velocity Obstacles* (*VO*) [1] and *Nonlinear Model Predictive Control* (*NMPC*) [3]. The proposed algorithms have been tested over a series of experiments in six different scenarios, based on different configurations of obstacles. For each algorithm and each scenario, a video of the solution has been produced. For each experiment involving the *random* obstacles' configuration, a statistical analysis has been performed to extract some more in-depth insights. Possible improvements and other existing approaches have been considered during the implementation phase of the project and some multi-agent path planning experiments have been performed.

1 Introduction

This project deals with the path planning task of a robot agent navigating in an observable, dynamic and planar environment populated by moving obstacles. The work was simplified by considering both the robot and the obstacles as simple omni-directional planar disks: doing so, the kinematics and dynamics of all the agents become unconstrained and, namely, they can instantaneously modify their velocity to any value at any moment.

Figure 1 shows a typical scenario: the robot (blue circle) starts from an initial configuration $p_0 = (x_0, y_0)$, in this case $(0.5, 0.5)$, and needs to plan its velocities so to avoid collisions with the dynamic obstacles (red circles) to reach a goal configuration $p_g = (x_g, y_g)$, in this case $(9.5, 9.5)$.

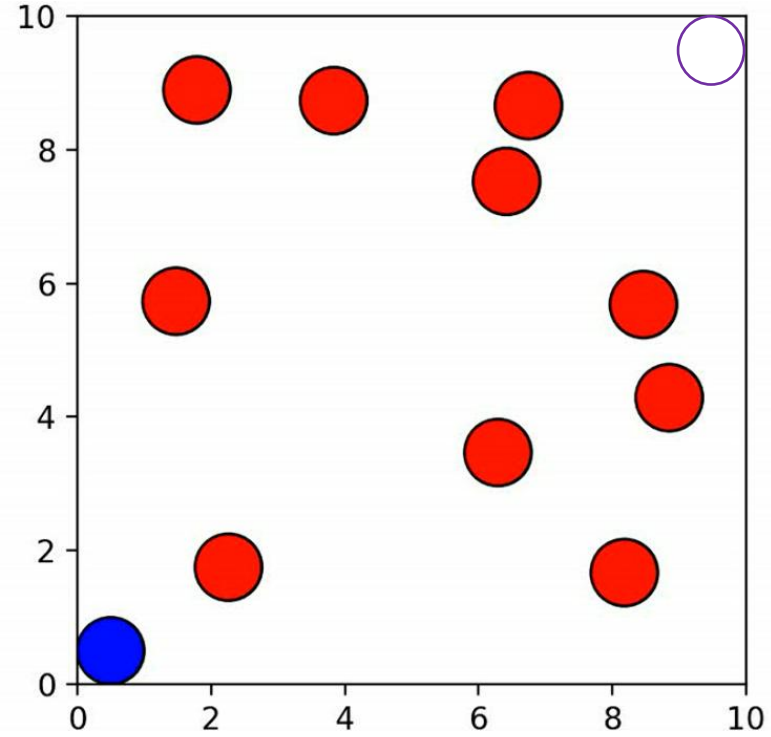

Figure 1: a typical scenario (the goal position is shown in purple).

This first section will explore the selected algorithms to solve the task, explaining them both qualitatively and in a formal and mathematical way. The following sections of this report will discuss the implementation choices to simulate and visualize the interactions between robot and obstacles. Then, the created scenarios, their parameters and the obtained results will be shown together with some in-depth statistical studies regarding *Velocity Obstacles*' success rate and *Nonlinear Model Predictive Control*'s cost functions. Finally, we will consider some possible algorithmic and environmental improvements viable to extend the project: in particular, some multi-agent path planning experiments have been performed by simply switching the dynamic obstacles with some other robot agents.

Let us begin the analysis on how robots can avoid collisions with dynamic obstacles using *Velocity Obstacles* and *Nonlinear Model Predictive Control* algorithms.

1.1 Velocity Obstacle

Velocity Obstacles (*VO*) [1] [2] is a classic path planning algorithm used for local and reactive navigation, which works with the concept of *Collision Cones*. Let us explore it more in details.

Let us consider a disk robot A with radius r_A and a dynamic disk obstacle B with radius r_B moving in our defined environment. Let their position and velocity at a particular time t be, respectively, p_A and v_A and p_B and v_B. Figure 2 (a) shows the described scenario.

The Velocity Obstacle $VO_{A|B}$ of the robot A generated by the presence of the dynamic obstacle B at time t is defined assuming B will keep on moving with a constant velocity v_B and is the set containing all the velocities of A that would cause a collision at some time $\hat{t} > t$. Formally, for our situation, it is defined as:

$$VO_{A|B} = \{v \mid \exists \hat{t} > t : \hat{t}(v - v_b) \in D(p_B - p_A, r_A + r_B)\} \quad (1)$$

where $D(p, r)$ is a disk of radius r centred at p. Figure 2 (b) shows a clear geometrical representation of $VO_{A|B}$.

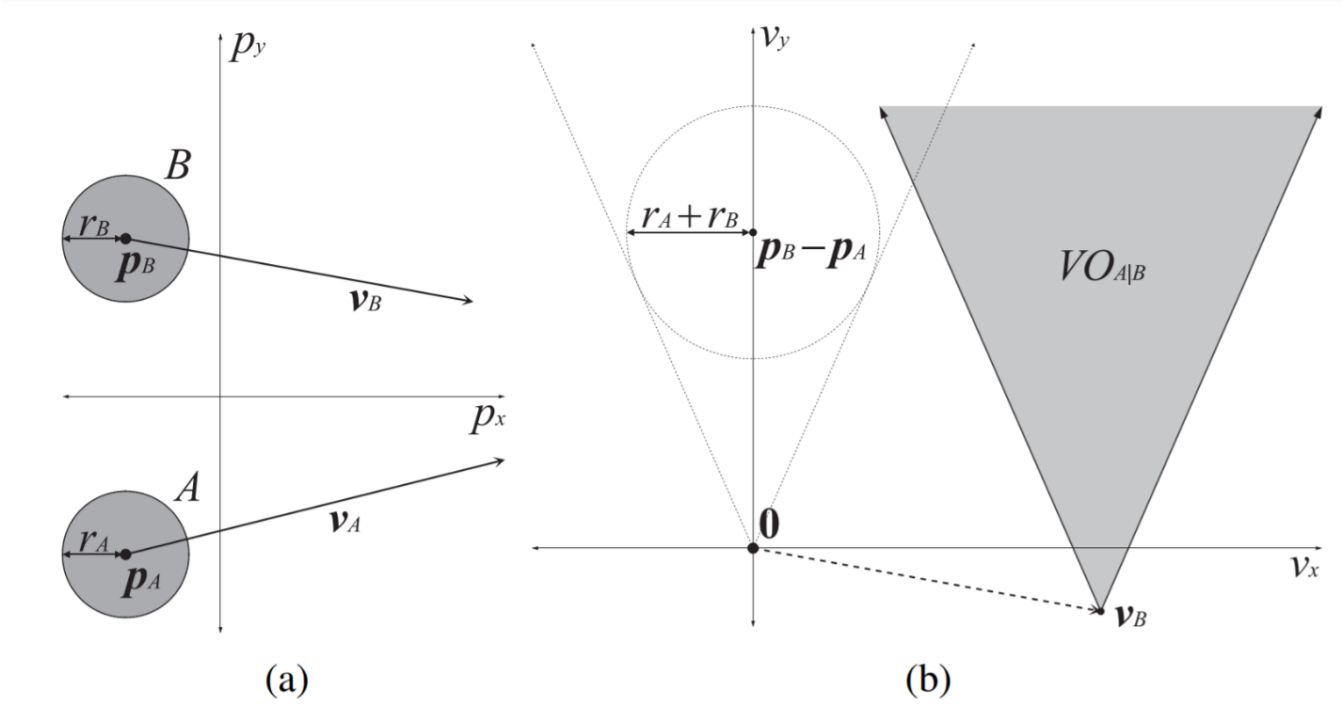

Figure 2: scenario including a robot A and a dynamic obstacle B (a) and the geometrical representation of the Velocity Obstacle induced by B on A (b).

Given the task of reaching from an initial position p_0 a goal position p_g, a generic iteration at time t of the algorithm adopted in the project is structured as follow:

1. The velocity desired by the robot A is computed ignoring the presence of obstacles as:

$$v_A^{des} = v_A^{MAX} \frac{p_A - p_g}{|| p_A - p_g ||} \tag{2}$$

where v_A^{MAX} is the max speed feasible by the robot A.

2. Construct, for each obstacle B_i, the set of constraints representing its Velocity Obstacle $VO_{A|B_i}$. In particular, I constructed each VO as two lines constraining the planar velocity-space. Figure 3 shows how I computed the constraints. Let v_B be the origin of the plane, $r = r_A + r_B$ and $d = || p_B - p_A ||$. The two constraining lines are constructed so to form $\theta + \varphi$ and $\theta - \varphi$ angles with the x-axis where:

$$\theta = \text{atan2}(d_y, d_x) \tag{3}$$

$$\varphi = \text{asin}(r/d) \tag{4}$$

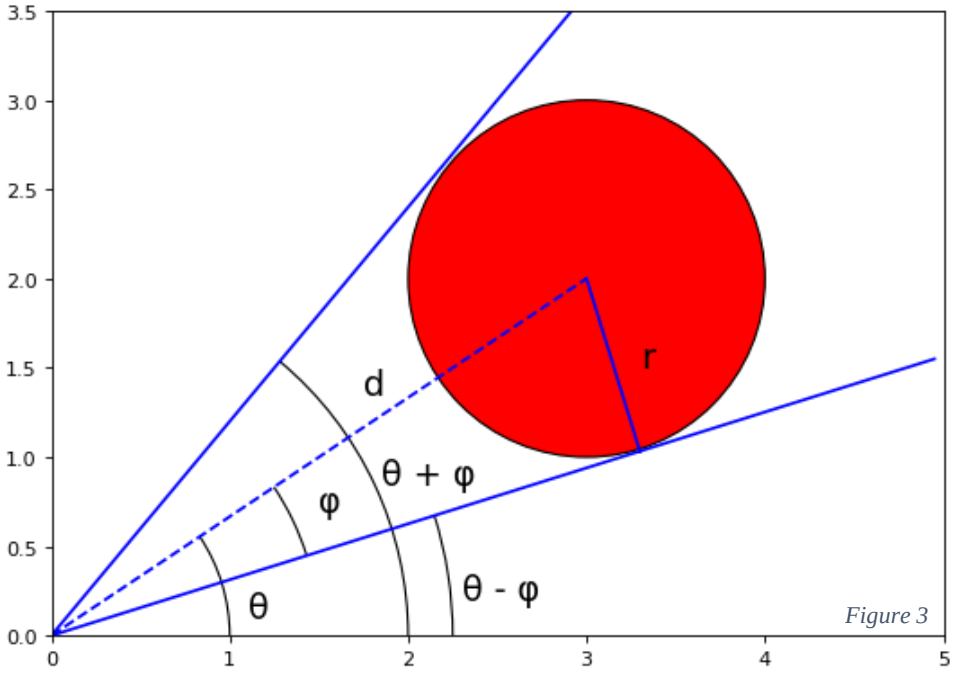

Figure 3

3. Let $V_A^{feasible}$ be the set of all the velocities feasible by the robot A, the set of safe velocities V_A^{safe} is now computed as:

$$V_A^{safe} = V_A^{feasible} - \bigcup_{B_i} VO_{A|B_i} \tag{5}$$

4. v_A is finally chosen as the safe velocity which is the most similar to v_A^{des}, computed as:

$$v_A = \underset{v \in V_A^{safe}}{arg\,min} || v - v_A^{des} || \tag{6}$$

Sometimes it can happen that $V_A^{safe} = \emptyset$: in this case I decided to set $v_A = \left(v_{A_x}, v_{A_y}\right) = (0,\ 0)$. In my work, this is considered as an "insecurity" and can cause a robot-obstacle collision: the algorithm is considered successful only if the simulation gets less than 6 insecurities.

1.2 Nonlinear Model Predictive Control

The second used algorithm is based on *Nonlinear Model Predictive Control* (NMPC) [3] [4], a classic method used to control a process while satisfying a set of constraints: in our case we are using it to solve the same path planning problem (for local and reactive navigation).

The general objective of this method aims at optimizing a desired task at time t_i considering a finite future time-horizon $\{t_{i+1}, t_{i+2}, t_{i+3}, \dots, t_j\}$, predicting in these future moments the states of relevant variables. Afterwards, when the next iteration is performed to solve the task at time t_{i+1}, the prediction is completely re-computed discarding the old predictions.

Given the task of reaching from an initial position p_0 a goal position p_g, a generic iteration at time t_i of the algorithm adopted in the project is structured as follow:

1. Defining the velocity desired by the robot v_A^{des} as in the *VO* algorithm, the trajectory τ_A^{des} desired by the robot in its future time-horizon is computed as:

$$\tau_A^{des} = \begin{pmatrix} p_A^{des}(t_{i+1}) \\ p_A^{des}(t_{i+2}) \\ .. \\ p_A^{des}(t_j) \end{pmatrix} = \begin{pmatrix} p_A + \Delta t v_A^{des} \\ p_A^{des}(t_{i+1}) + \Delta t v_A^{des} \\ .. \\ p_A^{des}(t_{j-1}) + \Delta t v_A^{des} \end{pmatrix} = \begin{pmatrix} p_A + \Delta t v_A^{des} \\ p_A + 2\Delta t v_A^{des} \\ .. \\ p_A + (j - i + 1)\Delta t v_A^{des} \end{pmatrix} \quad (7)$$

2. Predict for each obstacle B_i its future positions' trajectory $\tau_{B_i}^{pred}$ knowing its current position p_{B_i} and velocity v_{B_i}, it is simply accomplished in the following way:

$$\tau_{B_i}^{pred} = \begin{pmatrix} p_{B_i}^{pred}(t_{i+1}) \\ p_{B_i}^{pred}(t_{i+2}) \\ .. \\ p_{B_i}^{pred}(t_j) \end{pmatrix} = \begin{pmatrix} p_{B_i} + \Delta t v_{B_i} \\ p_{B_i}^{pred}(t_{i+1}) + \Delta t v_{B_i} \\ .. \\ p_{B_i}^{pred}(t_{j-1}) + \Delta t v_{B_i} \end{pmatrix} = \begin{pmatrix} p_{B_i} + \Delta t v_{B_i} \\ p_{B_i} + 2\Delta t v_{B_i} \\ .. \\ p_{B_i} + (j - i + 1)\Delta t v_{B_i} \end{pmatrix} \quad (8)$$

3. v_A is finally chosen as the velocity which minimizes my cost function, defined as:

$$v_A = \min\ cost(u) \quad (9)$$

with $u = (u_x, u_y)$ bounded so that both u_x and $u_y \in \left[-\frac{\sqrt{2}}{2} v_A^{MAX}, \frac{\sqrt{2}}{2} v_A^{MAX}\right]$.
The cost function I am using is defined as:

$$cost(u) = || \tau_A^u - \tau_A^{des} || + \sum_{t \in horizon} \sum_{B_i} \frac{Q_c}{1 + e^{k\left(|| p_A^u(t) - p_{B_i}^{pred}(t) || - r\right)}} \quad (10)$$

where $Q_c > 0$ is a tuning parameter; $k > 0$ is a parameter that defines the smoothness of the cost function (which is based on a logistic function); $r = r_A + r_B$; τ_A^u is the trajectory of the robot if it would move with a velocity u:

$$\tau_A^u = \begin{pmatrix} p_A^u(t_{i+1}) \\ p_A^u(t_{i+2}) \\ .. \\ p_A^u(t_j) \end{pmatrix} = \begin{pmatrix} p_A + \Delta t u \\ p_A + 2\Delta t u \\ .. \\ p_A + (j - i + 1)\Delta t u \end{pmatrix} \quad (11)$$

$cost(u)$ is conceptually composed by the sum of two parts. The first addend is the trajectory cost representing the difference between the trajectory traveled with a velocity u and the desired trajectory. The second addend is the collision cost representing the safety of the robot if it proceeded with a velocity u, quantified using the distance of the robot to the predicted position of each obstacle.

The minimization problem has been solved using a Sequential Least SQuares Programming (SLSQP) optimizer [5]. The described trajectories are graphically summarized in Figure 4 using a time-horizon of four timesteps.

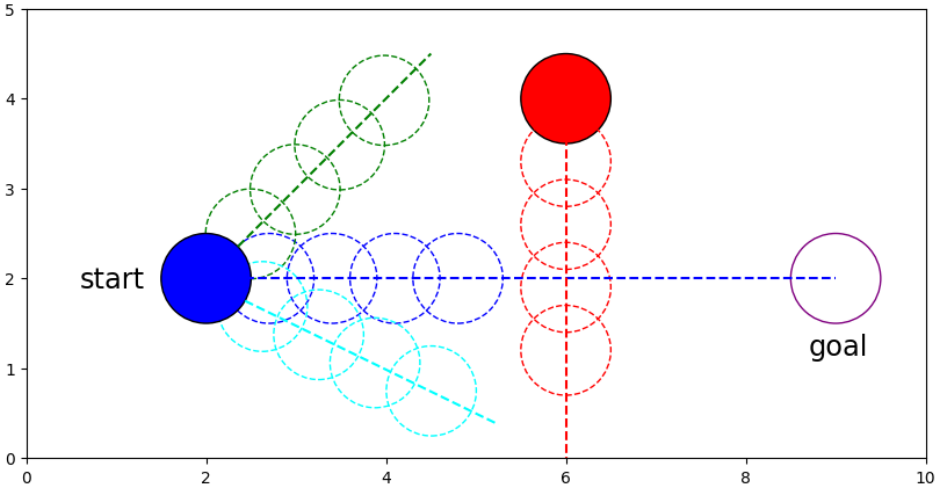

Figure 4: graphical representation of trajectories, τ_A^{des} in blue, τ_B^{pred} in red and two possible $\tau_A^{u_1}$ and $\tau_A^{u_2}$ in green and aqua.

2 Implementation

Let us now understand the development structure of my project.

2.1 User's experience and parameters

This project has been entirely built using the *Python* programming language inside of the environment offered by *Google Colab* (and it is supposed to be executed in that framework): indeed, one of the main goals I tried to achieve with this implementation was to offer to the users an easy benchmark for experimenting and simulating path planning with dynamic obstacles tasks with no need of supplementary installation of complex tools and without assuming that users are experts with a significant programming knowledge.

Running a simulation is extremely simple: after the execution of the "*Definitions*" cells, it is just needed to specify a set of desired experimental parameters to be ready to start an experiment. Figure 5 shows the available graphical user interface: all the variables' names are very self-explanatory and are here set to the values I personally chose for my collection of experiments.

There are five available obstacle configurations: *random*, *synchronized*, *synchronized_big*, *different* and *narrow*; *random* can be also distinguished in *random_reach* and *random_still*. Their connotations will be specified later inside the "*Experimental scenarios*" subsection.

For each experiment, the entire simulation time T is divided and discretised in timesteps of a chosen duration Δt. Therefore, a simulation will be clearly composed by $T/\Delta t$ timesteps. For instance, if a simulation is $T = 10\ s$ long and it is chosen to be discretized in timesteps of $\Delta t = 0.1\ s$, the path planning routine will have to execute $T/\Delta t = 100$ iterations.

The parameters Q_c and k required by *NMPC* algorithm have been set to $Q_c = 5$ and $k = 4$.

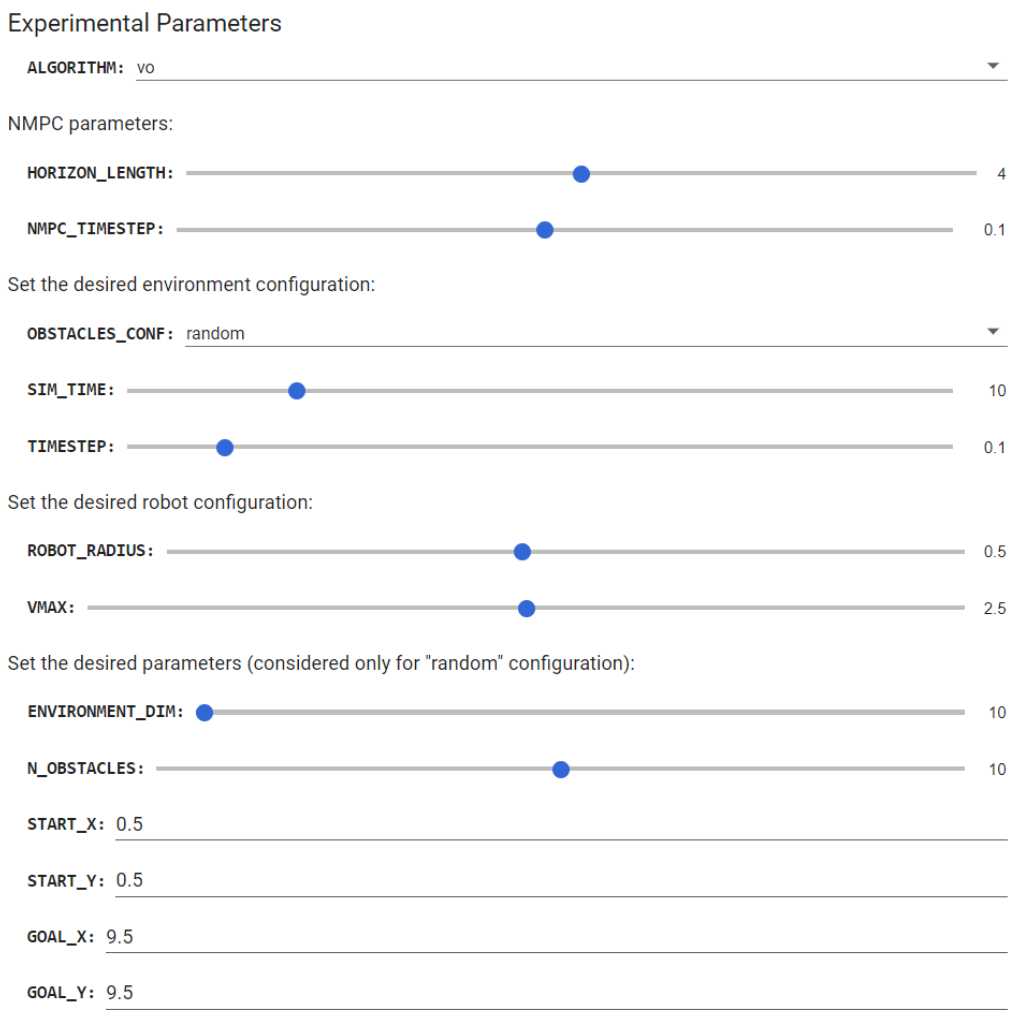

Figure 5: GUI the users can interact with to set their desired experimental parameters.

2.2 Robot and obstacles

The robot and the dynamic obstacles populating the environment are coded as simple classes. The main attributes for both of them are the lists $P = [p_0, p_1 \ldots]$ and $V = [v_0, v_1 \ldots]$ containing, for each timestep, their positions and velocities. The robot's motion between different timesteps is assumed to be uniform and linear: thus, every planning iteration computes v_i and then the robot's position is updated as $p_i = p_{i-1} + \Delta t v_i$.

Differently, the dynamic obstacles do not plan their velocities. They are spawned in a p_0 position with a v_0 velocity and they keep on moving with this velocity in a uniform linear motion for the entire simulation. If an obstacle collides with the environment's limits, its velocity's direction is programmed to reverse to keep each obstacle inside the mentioned limits. Obstacle-obstacle collisions are allowed.

Respecting our interpretation, Obstacle objects do not have any methods (i.e., they cannot interact with the surrounding environment), Robot objects comes with some methods fully implementing the two planning algorithms.

A last remark needs to be reported: in VO, $V_A^{feasible}$ is created as a finite set of 185 sampled feasible velocities of the robot; in *NMPC* instead v_A can be any solution of the minimization problem (obviously inside the problem's bounds). That's why it may happen that $V_A^{safe} = \emptyset$.

2.3 Simulate and video-generating functions

The simulate function contains all the main steps of an experiment. First of all, a list of Obstacle objects is created accordingly to the obstacle configuration selected by the user. The robot is also created in the chosen initial position and the goal is set. At this point, all that remains is simply to use the selected algorithm for each one of the timesteps and updating the relative positions $p_A(t_i)$ and velocities $v_A(t_i)$ of the robot with the planned values.

Once a simulation is completed, the user is given the opportunity to create a video using the robot and the list of obstacles. In particular, a classical routine to generate a video from *matplotlib* frames is used: for each timestep, an image representing it is created; later this set of images is merged together into a video using *OpenCV*.

3 Experiments

3.1 Experimental scenarios

Five different obstacle configurations are available for experiments:

- *Random* configuration: This scenario corresponds to the one already shown in Figure 1. The environment is a 10×10 square where n obstacles are spawned in random positions with random velocity directions but fixed magnitudes: the user is free to choose this n value. Obstacles are programmed so to avoid initial arrangements in which they are already colliding with the robot. The simulation time $T = 10\ s$. This configuration was experimented in two different settings:
    - *Random_reach*: The robot is created in $p_0 = (0.5, 0.5)$ and has the goal of reaching $p_g = (9.5, 9.5)$.
    - *Random_still*: The robot is created in $p_0 = (5, 5)$ and has the goal of staying still in that positions, i.e. $p_g = (5, 5)$.

- *Synchronized* configuration: The environment is a 10×10 square where 8 obstacles are spawned in a "grid" configuration moving horizontally and vertically. The simulation time $T = 10\ s$. The robot is created in $p_0 = (0.5, 0.5)$ and has the goal of reaching $p_g = (9.5, 9.5)$.

- *Synchronized_big* configuration: The environment is a 50×50 square where a lot of obstacles are spawned in a "grid" configuration moving horizontally and vertically. The simulation time $T = 30\ s$. The robot is created in $p_0 = (0.5, 0.5)$ and has the goal of reaching $p_g = (49.5, 49.5)$.

- *Different* configuration: The environment is a 20×20 square where 13 obstacles are spawned. Five of these obstacles are fixed, while the others move horizontally and vertically. While developing this configuration, I started testing new elements: in facts, the original papers only used dynamic obstacles (no fixed ones) with radii equal to the robot's one. This scenario's peculiarity is the presence of obstacles with different radii lengths (0.5 for the dynamic obstacles, 1 and 4 for the fixed ones). The simulation time $T = 30\ s$. The robot is created in $p_0 = (0.5, 0.5)$ and has the goal of reaching $p_g = (19.5, 19.5)$.

- *Narrow* configuration: The environment is a 20×20 square where 2 dynamic obstacles are spawned moving vertically. This scenario's peculiarity is the presence of a lot of fixed obstacles simulating some walls: the robot must navigate in a narrow passage to reach the goal. The simulation time $T = 15\ s$. The robot is created in $p_0 = (2, 2)$ and has the goal of reaching $p_g = (19.5, 19.5)$.

In Figure 6 below all the defined scenarios are shown with their obstacle configurations. Obstacles' paths are displayed as red dashed lines and goal positions as purple circles.

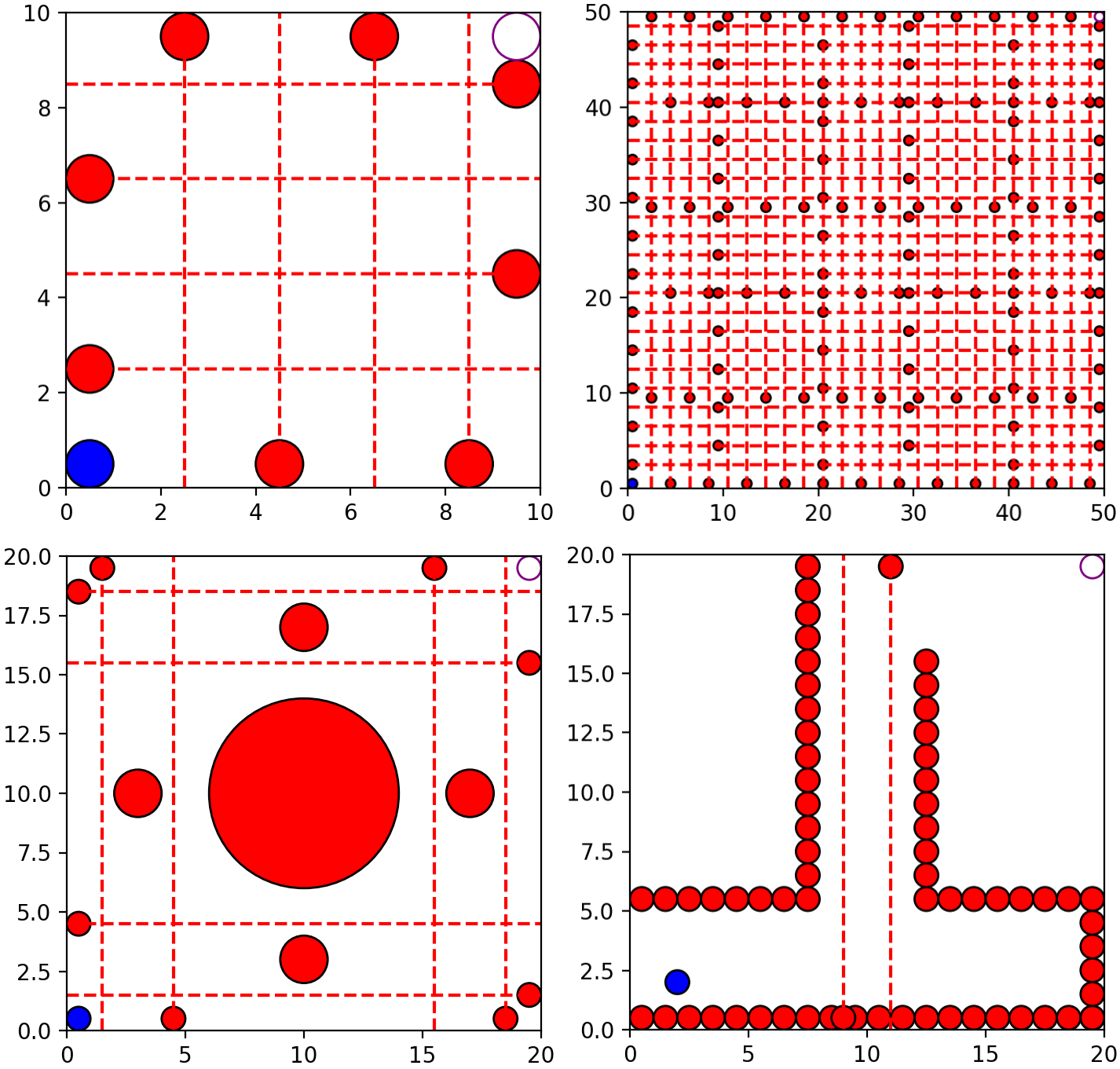

Figure 6: representation of scenarios. Synchronized (top left), synchronized_big (top right), different (bottom left) and narrow (bottom right).

3.2 Experimental results

All the created environments have been tested different times in multiple simulations. Excluding the *random* configurations, if a simulation is repeated several times the result is always almost the same. Let us now analyse the results obtained to better understand the behaviour of *VO* and *NMPC*.

3.2.1 Velocity Obstacles

Figure 7 shows for each experiment the path planned and travelled by the robot (as a dotted line) in the respective T of the scenario.

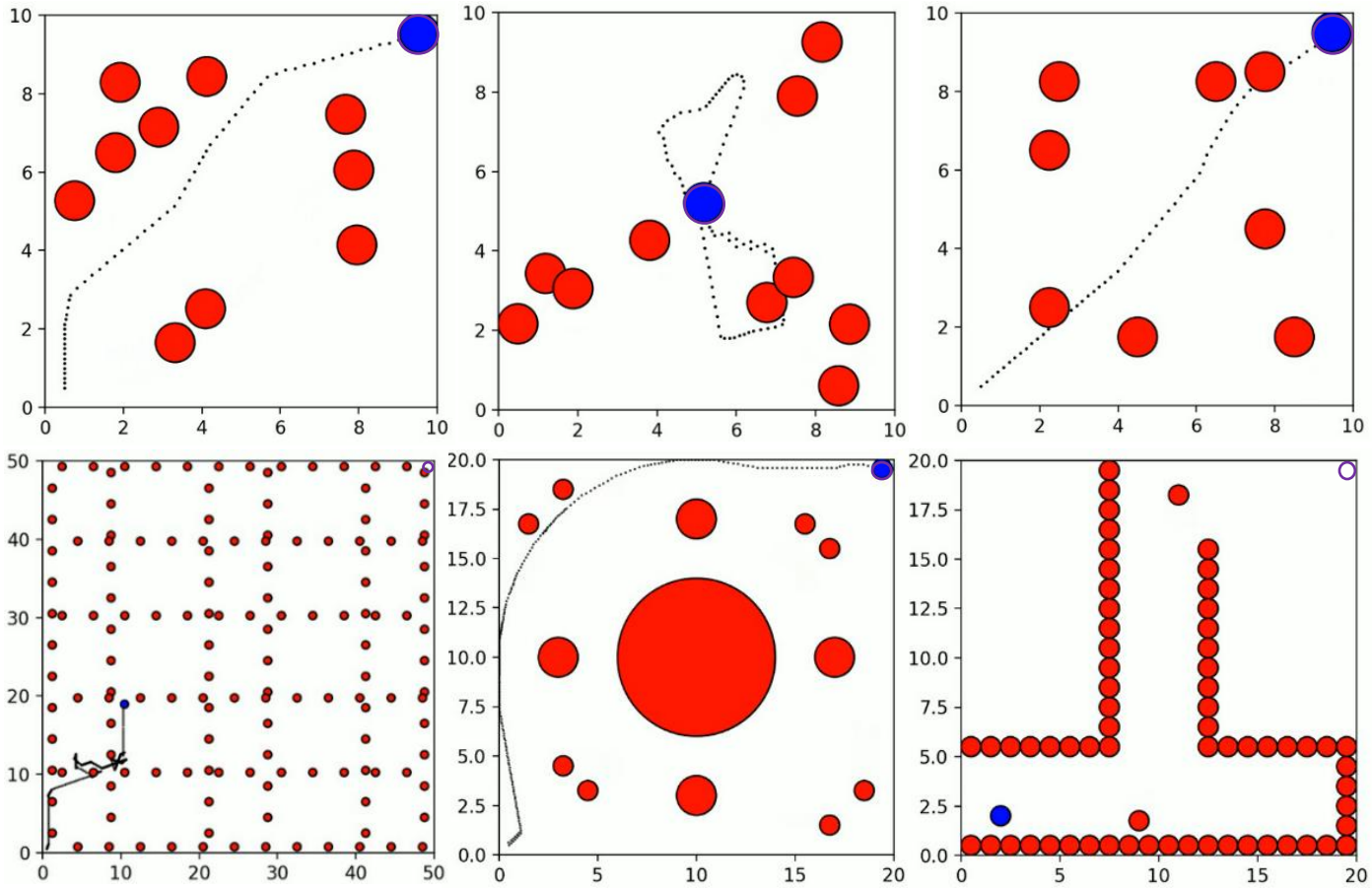

Figure 7: VO results.

VO's unsuccessful cases are the *synchronized_big* and the *narrow* configurations. Clearly this is caused by the fact that in both of these scenarios there are a large number of obstacles that greatly limit the V_A^{safe} set. However, while in *synchronized_big* the robot manages to begin a difficult journey towards its goal (and perhaps with a much higher T it would be able to reach it), instead in *narrow* the robot is completely blocked by the presence of fixed obstacles surrounding it and remains immobile (which in any case is a position that guarantees it to not collide with any obstacle).

For the *random* obstacles' configuration, a statistical analysis has been performed to extract some more in-depth insights: it is shown in Figure 8. In particular, for every $n \in [1, 20]$ I simulated both *random_reach* and *random_still* 300 times with n obstacles and extracted the success rate depending on n. As mentioned in section 1.1, a simulation success is determined based on the number of its "insecurities".

The worst performance of *random_still* is explained by the initial and final position of the configuration: the robot having to remain in the center of the environment is forced to stay surrounded by obstacles and interact with them much more.

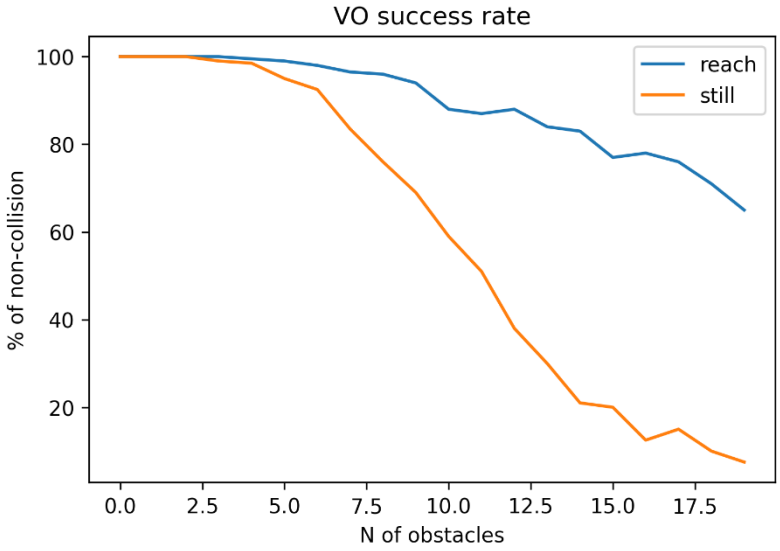

Figure 8: VO's success rate in the random configuration depends from the n of obstacles used.

3.2.2 NMPC

Figure 9 shows for each experiment the path planned and travelled by the robot (as a dotted line) in the respective T of the scenario.

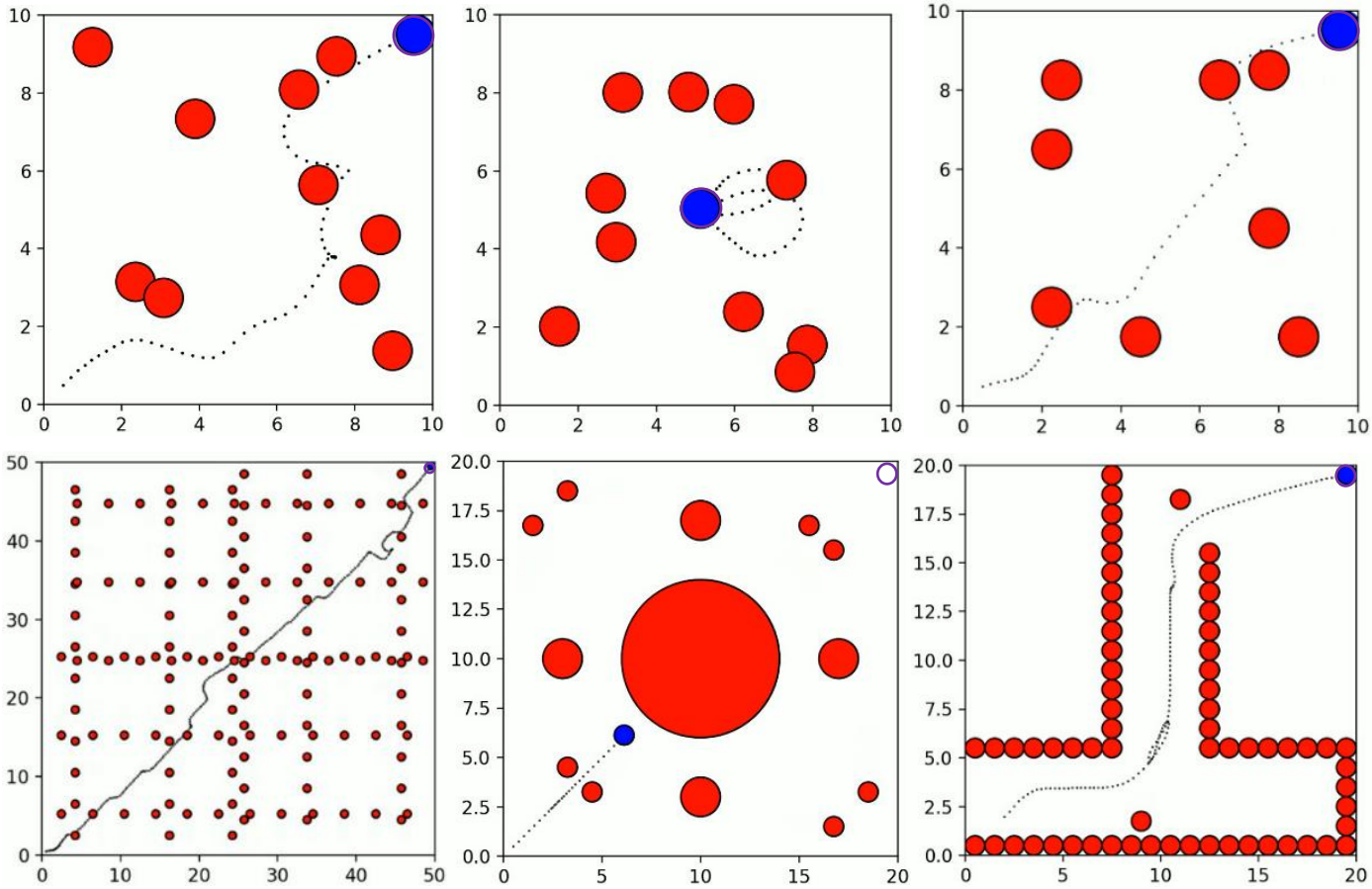

Figure 9: NMPC results.

NMPC's unsuccessful case is only the *different* configuration where the robot gets stuck in front of the biggest fixed obstacle. I suppose that the algorithm computes a larger cost for a motion around the big obstacle because the cost would increase both because the robot would get closer to the smaller fixed obstacles and both because it would step away from the desired trajectory (which clearly is a 45° line from p_0 to p_g): therefore, it remains stationary there. Anyway, excluding this failing case, it must be noted how well this algorithm behaves in the other configurations.

For the *random* obstacles' configuration, a statistical analysis has been performed to extract some more in-depth insights: it is shown in Figure 10. In particular, for every $n \in [1, 20]$ I simulated both *random_reach* and *random_still* 300 times with n obstacles and extracted the experiments' mean cost depending on n. Again, it is obvious that *random_still* is going to have higher mean costs because the robot is forced to stay surrounded by obstacles and interact with them much more. Moreover, the selected costs for each timestep of the other configurations have been plotted in Figure 11.

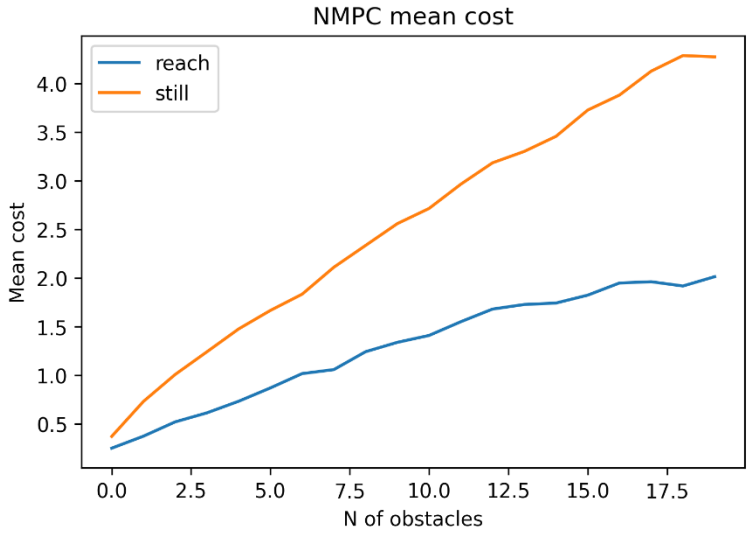

Figure 10: NMPC's mean cost in the random configuration depends from the n of obstacles used.

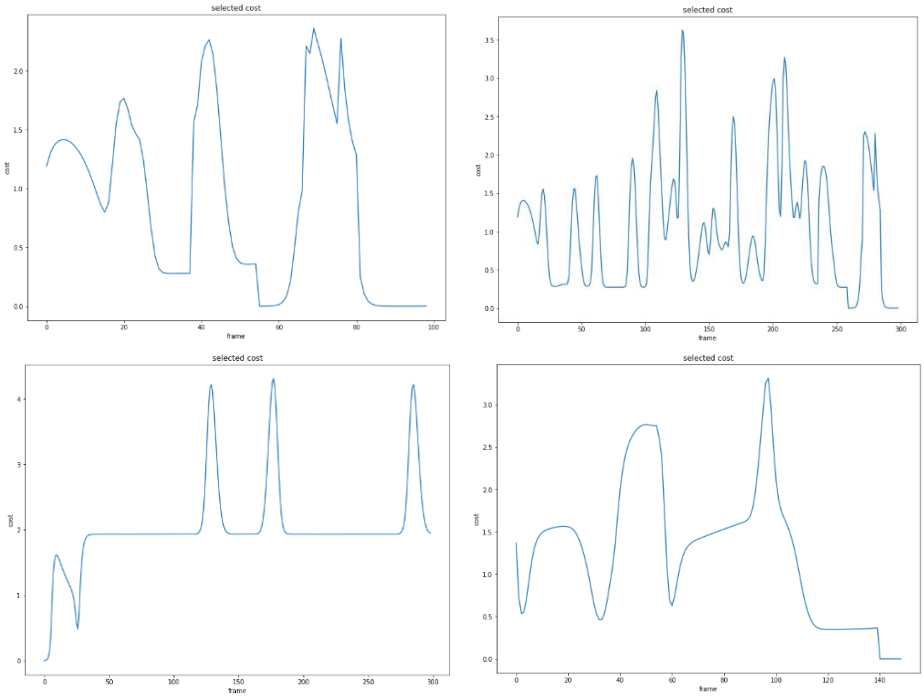

Figure 11: NMPC's costs for timestep. Synchronized (top left), synchronized_big (top right), different (bottom left), narrow (bottom right).

3.3 Extensions

At this point, the overall goal of this project is achieved. As it is easily understandable, there may be infinite possible extensions to my simple benchmark: the framework could be improved, there may be ways to solve the failure cases for *VO* and *NMPC* (perhaps even by just tweaking the experimental parameters some more), new scenarios could be created and new algorithms could be considered. Expert users are free to implement any extension they want.

So, I decided to implement just a small extra point to conclude the project by switching the task from "Path planning with dynamic obstacles" to "Multi-agent path planning". This was easily done by removing the basic dynamic obstacles and using various n robot agents in the same environment: each one of them considers the rest of the robots as dynamic obstacles. In each timesteps, one robot at a time plans its own path; therefore, the selected algorithm is used n times in a timestep.

To experiment this task, I created a simple scenario where four robots are spawned in its corners and, for each one of them, their goal is to reach the opposite corner. Figure 12 shows this scenario and the solutions produced, respectively, by *VO* and *NMPC*. *VO*'s result is optimal, each robot moves just a bit to the side to avoid collisions. In *NMPC*'s solution instead the robots initially move on the desired trajectories but, when they get close enough to each other and the collision cost overcomes the trajectory cost, they begin moving to the side with an oscillating motion; finally, they reach the goal again using the desired trajectory.

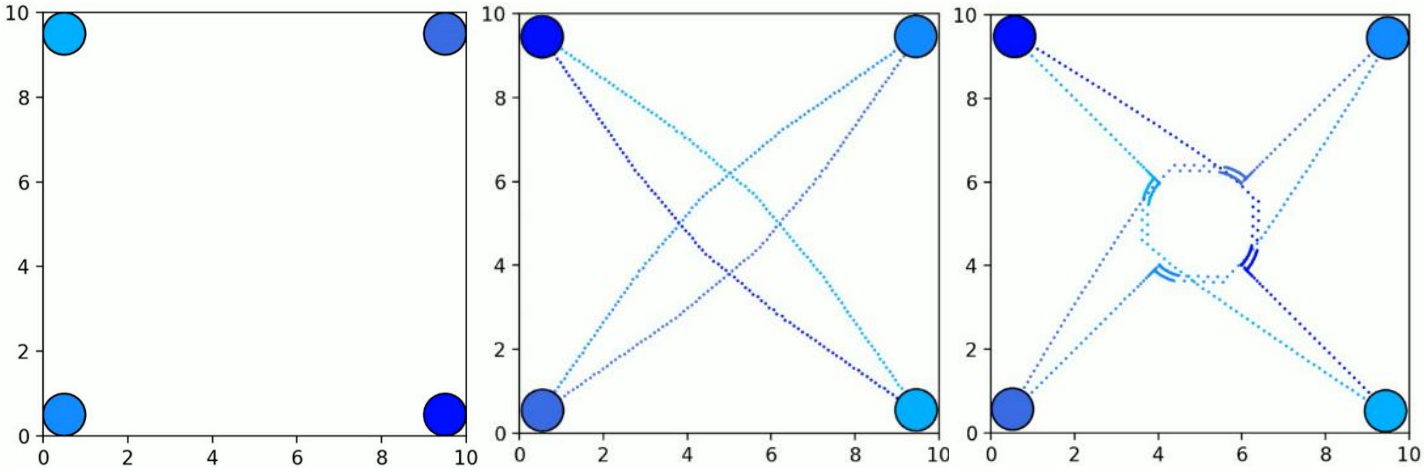

Figure 12: multi-agent scenario (left), VO solution (middle), NMPC (right).

The costs selected by each robot in the *NMPC*'s iterations have been plotted in Figure 13 and, as it can be seen, they are almost identical because the dynamics of the robots are almost the same for each one. This figure therefore clearly demonstrates that there exists a relationship between a robot's behavior and its selected cost.

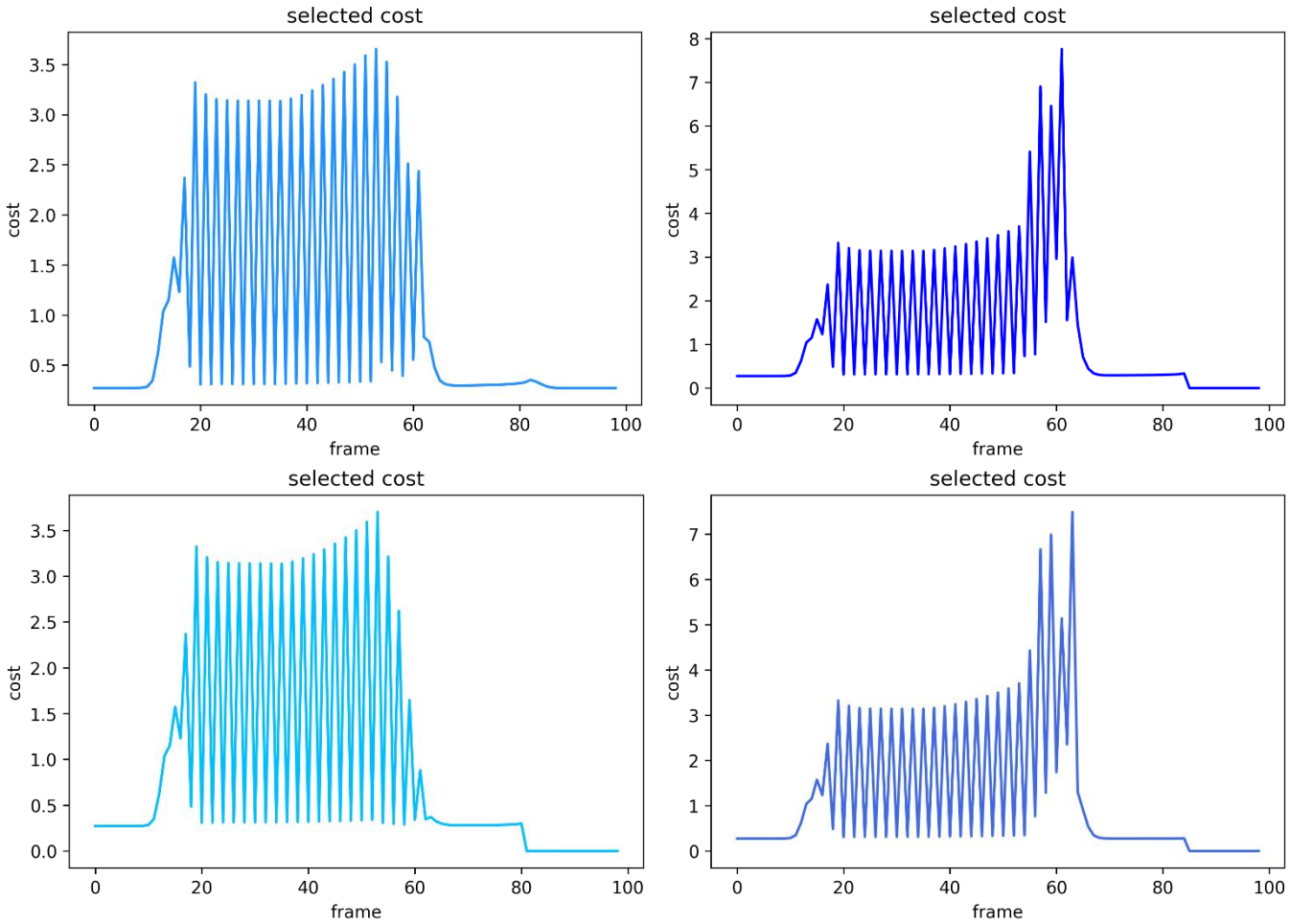

Figure 13: NMPC's costs for timestep for each one of the four robots.

4 Conclusions

The main goal of the project was the implementation and comparison of algorithms for path planning with dynamic obstacles. Two of the major classical approaches were used to achieve this goal: *Velocity Obstacles* (*VO*) and *Nonlinear Model Predictive Control* (*NMPC*). They have been formally defined and explained. The proposed algorithms have been tested over a wide set of experiments in six different scenarios I created, based on different configurations of obstacles. The implementation choices to simulate and visualize the interactions between robot and obstacles were analysed, together with the experimental parameters. The obtained results and some statistical studies were shown: I explored both the successes and the failure cases of the algorithms. Finally, coherent and possible project's extensions were considered: in particular, some multi-agent path planning experiments have been performed to demonstrate the adaptability of my benchmark to new tasks.

5 References

[1] Paolo Fiorini and Zvi Shiller (1998) *Motion planning in dynamic environments using velocity obstacles.*

[2] Jamie Snape, Jur van den Berg, Stephen J. Guy, and Dinesh Manocha (2011) *The Hybrid Reciprocal Velocity Obstacle.*

[3] Mina Kamel, Javier Alonso-Mora, Roland Siegwart, and Juan Nieto (2017) *Nonlinear Model Predictive Control for Multi-Micro Aerial Vehicle Robust Collision Avoidance.*

[4] Mukhtar Sani, Bogdan Robu, and Ahmad Hably (2021) *Dynamic Obstacles Avoidance Using Nonlinear Model Predictive Control.* IECON 2021.

[5] Kraft, D. (1988) *A software package for sequential quadratic programming.* DLR German Aerospace Center - Institute for Flight Mechanics, Koln, Germany.