Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2

Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2

Pangcheng David Cen Cheng, Marina Indri, Fiorella Sibona, Matteo De Rose
Dipartimento di Elettronica e Telecomunicazioni (DET)
Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129 Torino, Italy
{pangcheng.cencheng, marina.indri, fiorella.sibona}@polito.it
matteo.derose@studenti.polito.it

Gianluca Prato
Fondazione LINKS
Torino, Italy
gianluca.prato@linksfoundation.com

***Abstract*—Mobile robots can highly contribute to achieve the production flexibility envisaged by the Industry 4.0 paradigm, provided that they show an adequate level of autonomy to operate in a typical industrial environment, in which the presence of both static and dynamic obstacles must be managed. Robot Operating System (ROS) is a well known open-source platform for the development of robotic applications, recently updated to the enhanced ROS2 version, including a navigation stack (Nav2) providing most, but not all the capabilities required to a mobile robot operating in an industrial environment. In particular, it does not embed a strategy for dynamic obstacle handling. Aim of this paper is to enhance Nav2 through the development of a Dynamic Obstacle Layer, as a plug and play solution suitable for the integration of the dynamic obstacle information acquired by a generic 2D LiDAR sensor. The effectiveness of the proposed solution is validated through a campaign of simulation tests, carried out in Webots for a TurtleBot3 burger robot, equipped with a RPLIDAR A3 LiDAR sensor.**

***Index Terms*—Autonomous Mobile Robots, Dynamic path planning, ROS2**

I. Introduction

Nowadays, mobile robots are frequently employed in industrial automation, surveillance, transportation, personal and medical applications. In an industrial context, mobile robots are revolutionising flexible manufacturing systems and logistics, where Automated Guided Vehicles (AGVs) have predominated for years. Even though they accomplish localization and navigation based on robust methods, such as wire guidance, ceiling mounted bar codes, or magnetic tape following, they have two main disadvantages [1]: a limited drive-path and a restricted interaction with the workstations. On the other hand, Autonomous Mobile Robots (AMRs) are developed as intelligent agents that can actively interact with the industrial environment and so, they better attain the level of flexibility envisioned by the Industry 4.0 revolution.

To this aim, AMRs perform autonomous navigation, typically achieved by integrating perception data, localization, cognition and motion control. Navigation can be decomposed into the following tasks [2]: (i) modelling the world as a map, (ii) computing collision-free trajectories, and (iii) path following while avoiding collision with obstacles.

The last two tasks are usually referred to as the *motion planning* problem [3], which over the years boosted the development of several algorithms for its solution, based on different mathematical approaches and technologies [4]. Path planning algorithms can be grouped in global and local planners. The former exploit the information of the map to build a feasible obstacle-free path to traverse from one point to another. The latter instead, compute new intermediate waypoints taking into account the local information provided by the sensors. Such waypoints try to avoid the obstacles that were not known a-priori, matching as much as possible the ones provided by the global planner. There are many dynamic obstacle avoidance strategies that recompute the trajectories by generating arcs, segments, clothoid lines, etc, whose outputs are intermediate waypoints that deviate the robot from dynamic obstacles [5].

ROS is a well-known open-source platform for developing robotic applications. In particular, the ROS navigation stack metapackage [6] constitutes a widely established framework for robot autonomous navigation. Local planners made available in ROS are Dynamic Window Approach (DWA), Elastic Band (EBand) and Time Elastic Band (TEB).

The DWA is an online collision avoidance algorithm that takes into account the dynamics of the mobile robot. Considering the velocity and acceleration constraints of the robot, its operating principle includes two main phases: firstly, it generates a valid velocity search space, and then it selects the optimal solution through a cost function evaluating the trajectories scores [7]. EBand deforms the global path when new obstacles are detected, using an artificial force model [8]. An elastic band is created by a contraction force that pulls the robot towards the goal position, while a repulsive force pushes the path away from obstacles. TEB is an improvement of the EBand, in which the time information is added to the path computation. Nevertheless, issues arise when the dynamic obstacle intercepts the recomputed path, so three elastic bands are created as alternative paths and the shortest one compliant with the planning requirements is chosen [9]. The choice of the most suitable planner depends on the navigation requirements, and it is generally a trade-off among precision, speed and performance. For instance, according to [10], [11], DWA planner stands out for its small computing power requirement and repeatability in consecutive tests, while EBand provides more accurate results, and TEB is the fastest to react to the dynamic obstacles, although it requires more computer resources as it tries to optimize multiple trajectories.

ROS currently has two versions: from now on we will refer to the original version as ROS1. Despite ROS1 usage has increased a lot since its first distribution, it has some architectural limitations that prevents it from being competitive with other solutions. In fact, ROS1 requires significant computing power and cannot guarantee fault-tolerance, deadlines or process synchronization and, most importantly, it does not satisfy real-time execution requirements [12]. The first ROS2 distribution was released in 2017 with the following design goals: support teams of multiple robots, small embedded platforms, real-time control, non-ideal networks, and multi-platform support [13]. Alongside ROS2, the navigation stack underwent a substantial architecture evolution as well, giving birth to the Navigation2 (Nav2) stack [14]. However, Nav2 still leaves some open issues about dynamic obstacles avoidance, as raised by the ROS2 community [15]. Indeed, it does not embed a strategy for dynamic obstacles handling: navigation is performed on the basis of a costmap representation of the environment, where static occupied costmap cells are inflated by an exponentially decreasing cost decay rate, irrespective of whether they are occupied by moving or still objects. Thus, the robot plans a more pessimistic trajectory and maintains a larger separation from obstacles than actually required by collision avoidance, but without the awareness of the temporal evolution of the occupancy grid. As a consequence, the AMR navigation performance is affected by unnecessary delay, which in dynamic factory environments and shopfloors results in lower productivity and preventable downtimes.

The main goal of this paper is to propose the Dynamic Obstacle Layer (DOL) approach, as a plug and play solution to integrate the current Nav2 stack with the dynamic obstacles information coming from a generic 2D LiDAR sensor, starting from some preliminary results available in [16]. First, Section II outlines some characteristics of the Nav2 architecture, highlighting how the DOL is integrated. Then, some related works, taken as a starting point for this study, are presented. Section III illustrates the concepts and theory behind the DOL approach, while Section IV presents the main steps followed for the implementation in ROS2. The results are analysed in Section V. Finally, the conclusion and future works are outlined in Section VI.

II. Background

A. Nav2: A navigation system

Nav2 comes with a modular and (run-time) reconfigurable core, consisting in a Behavior Tree (BT) navigator [17] and task-specific asynchronous servers: *Planner*, *Controller* and *Recovery* servers. They are action servers hosting the environmental representation used by the algorithm plugins to compute their outputs, under the orchestration of the BT navigator. In particular:

- The task of *Planner* plugins is the computation of a valid, and potentially optimal, path from the current pose to a goal pose, serving the function of global planning.
- *Controller* plugins replace the *local_planner* of Nav1: they compute a feasible control effort to follow the global plan, based on a local environmental representation, thus performing local planning.
- The *Recovery* behaviours are plugins triggered by the BT when a navigation failure occurs.

The *Planner* and *Controller* servers work on two different costmap representations of the environment, the *global_costmap* and the *local_costmap*, respectively. The former built upon a pre-loaded static map and the latter based on sensor information. The huge potential of the costmap representation in Nav2 lies in the adoption of the costmap layers method [18]: unlike traditional monolithic costmaps, where all the data are stored in a singular grid of values, in the costmap layers approach, each layer tracks one type of obstacle or constraint, and then modifies a master costmap that is used for the path planning. The base layers in Nav2 are essentially three: (i) static layer – stores the costs associated with the static map provided at launch time, (ii) obstacle layer – continuously marks and clears cells according to sensor data, and (iii) inflation layer – propagates cost values out from occupied cells that decrease with distance, in order to provide a safety margin for the robot navigation.

The dynamic obstacle handling is usually faced in motion planning at a local level, meaning that in Nav2 dynamic obstacle avoidance should be addressed by the *Controller Server*. In fact, among the Controller Server plugins for local planning deployed in Nav2 there are the DWB Controller (*nav2_dwb_controller*), and the TEB Controller (*teb_local_planner*). In particular, the former is the successor to the DWA controller in ROS1.

The DOL has been thought as an additional costmap layer plugin to the *local_costmap*, embedding the information of dynamic obstacles velocity and orientation information into the occupancy grid, together with the existing controller plugins.

B. Dynamic path planning

Being the current flexible production plants highly demanding environments, research is brought on with the aim of enhancing the ROS1 NavStack to make it compliant with industrial highly dynamic and ever-changing environments [19], along with ROS1 to ROS2 migration processes to take advantage and improve well-established frameworks, providing valuable capabilities, e.g., task planning [20]. Given its features, ROS2 is suitable for industrial-grade mobile platforms, making it a relevant choice for industrial applications to achieve safe dynamic obstacles avoidance. Several works about laser range finder data processing for dynamic obstacles detection and tracking are available in literature [21]–[23]. Here, the dynamic obstacles detection is performed applying some heuristic algorithms directly on the LiDAR pointclouds, clustering the obstacles based on parameters such as the cluster radius or the distance between points. In this work, dynamic obstacles are detected and tracked taking advantage of the occupancy grid provided by the Nav2 packages, making it a modular solution such as the ROS1 *costmap_converter* package implemented in [24]. In this way, the DOL is less dependent on the type of sensor used. Then, a policy for

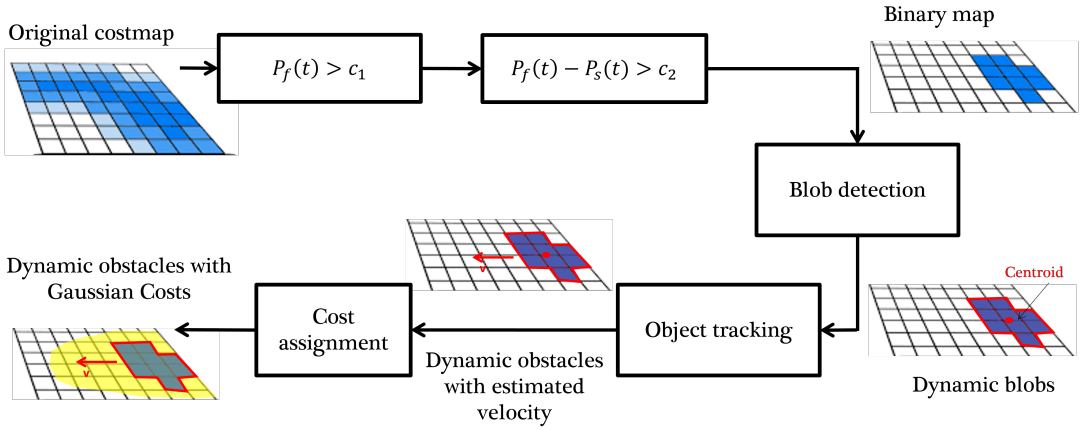


Fig. 1: Dynamic Obstacle Layer method steps.

translating the obstacles velocity and orientation information into the costmap has been defined exploiting the available Inflation Layer structure and applying a risk level set concept similar to the one presented in [25].

III. The DOL approach

The DOL approach developed in this work is articulated in three steps:

Object detection Starting from the costmap representation of the environment, dynamic obstacles are identified and separated from static ones, applying image processing algorithms and running average filters.

Object tracking Detected dynamic obstacles are tracked and their velocity is estimated applying a Kalman Filter.

Cost assignment A developed costmap layer assigns costs around each moving obstacle in the *local_costmap*, according to a 2D Gaussian shape, with variances proportional to the obstacle velocity and oriented along its moving direction.

A. Object detection

Once the robot is ready to navigate, its costmap representation of the environment is handled as an image during the object detection step. So, foreground indicates whatever is moving, while the background is everything that is static. *Background subtraction* is obtained applying two running average filters to each pixel – a "fast" and a "slow" filter:

$$P_f(t+1) = \beta[(1-\alpha_f)P_f(t)+\alpha_f C(t)] + \frac{1-\beta}{8}\sum_{i\in NN} P_{f,i}(t)$$

$$P_s(t+1) = \beta[(1-\alpha_s)P_s(t)+\alpha_s C(t)] + \frac{1-\beta}{8}\sum_{i\in NN} P_{s,i}(t)$$

where $P_f(t)$ and $P_s(t)$ represent the output of the fast and the slow running average filters at time t, respectively. β denotes the ratio between the contribution of the central cell filter and the effect of the neighbouring cells to $P_f(t)$ and $P_s(t)$, so to capture the running average filter of the 8 Nearest Neighbour (NN) cells, since large objects form blocks of cells in the local costmap. The gains α_f and α_s define the effect of the current costmap $C(t)$ on both filters. Therefore, the two filter rates are chosen such that

$$0 \leq \alpha_s < \alpha_f \leq 1$$

Then, two thresholding steps are performed to filter out the high and low frequency noise and identify those pixels occupied by dynamic obstacles:

1) The fast filter classifies a cell as foreground if it exceeds threshold c_1:
$$P_f(t) > c_1$$
2) The difference between the fast and the slow filter has to exceed a threshold c_2 in order to eliminate quasi-static obstacles with low frequency noise:
$$P_f(t) - P_s(t) > c_2$$

The constant values c_1 and c_2 have been set heuristically based on the range of values that cells can assume in a Nav2 costmap (from 0 to 255) and considering the same settings adopted in [24]. The output is a binary map where all the dynamic pixels are marked with "1", as it is shown in the general scheme of the DOL approach reported in Figure 1.

Then, the *SimpleBlobDetector* heuristic algorithm, provided by the OpenCV library [26], clusters the dark pixels in the binary image into blobs representing each dynamic obstacle in terms of contours – a list of the cells which define the blob contours – and centroid – the coordinate of the cell (pixel) in the weighted center of the blob.

B. Object tracking

The centroid of dynamic obstacles progresses with each costmap update and subsequent foreground detection. The assignment of blobs in the current map to obstacle tracks constitutes a data association problem. In order to disambiguate and track multiple objects over time, the current obstacles are matched with the corresponding tracks of previous obstacles. A new track is generated whenever a novel obstacle emerges that is not tracked yet. Tracks that are not assigned to current

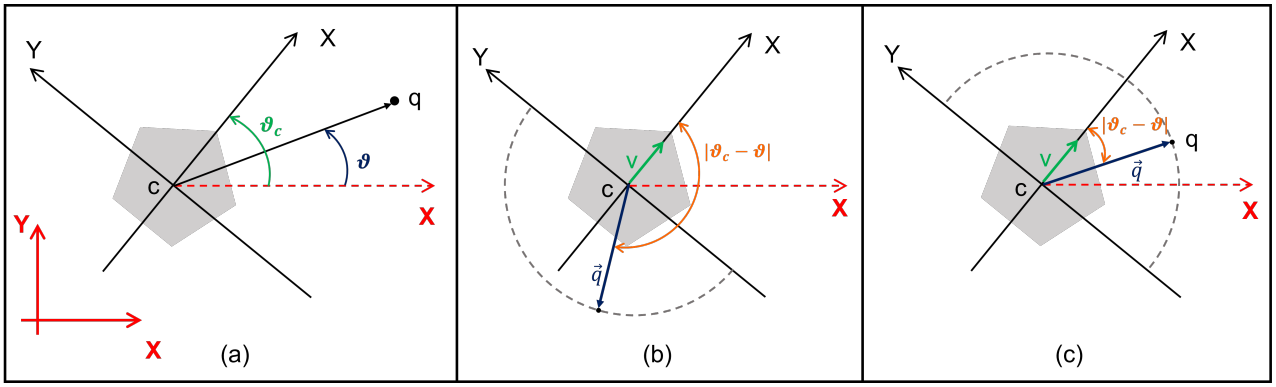


Fig. 2: a) Local obstacle reference frame (black) with respect to the global (map) reference frame (red). b) A point q within the back area. c) A point q in the frontal space.

objects in the foreground frame are temporarily maintained. The track is removed if it is no longer confirmed by object detections over an extended period of time. The assignment problem is solved by the so-called Hungarian algorithm [27], which solves weighted assignment problems by minimizing the total Euclidean distance between the tracks and the current set of obstacle centroids.

Then, a Kalman filter estimates the current velocity of tracked obstacles assuming a first order constant velocity model, since it is sufficient to capture the prevalent motion patterns of humans and robots in indoor environments.

C. Cost assignment

To include the dynamic obstacles information into the costmap, the region around each detected obstacle is inflated with a 2D Gaussian shape. In particular, the magnitude of the obstacle velocity has been associated with the peak of the Gaussian; in this way, faster obstacles are inflated more than slower ones. This has the aim to make the local planning aware of the obstacle with a sufficient heads-up for replanning.

Furthermore, the orientation information is used to inflate more the cells along the moving direction of the obstacles. This is actually obtained blending two 2D Gaussian shapes, one inflating the cells in front of the obstacle and the other inflating the cells on its back region. Given an obstacle O with centroid in position $c(x, y)$ in the map reference frame, we define a local coordinate system with origin in c, X-axis oriented along the velocity vector direction, Z-axis pointing outwards the costmap plane and Y-axis set according to the right-hand rule (Figure 2a). Therefore, the obstacle inflation region is represented by the following function:

$$\Phi_{c,\Sigma_{front},\Sigma_{back}}(q) = \delta(q)\Phi_{c,\Sigma_{front}}(q) + [1 - \delta(q)]\Phi_{c,\Sigma_{back}}(q)$$

where $q = (x_q, y_q)$ collects the coordinates of a point in the map reference frame, $\Phi_{c,\Sigma_{front}}$ and $\Phi_{c,\Sigma_{back}}$ are the Gaussian functions that inflate the frontal and the back area of the obstacle, respectively. $\delta(q)$ selects the correct Gaussian function depending on whether the considered cell is in the frontal or back space of the obstacle (Figs. 2b and 2c), and it is defined as follows:

$$\delta(q) = \begin{cases} 1 & \text{if } \vec{v} \cdot \vec{q} \geq 0 \\ 0 & \text{if } \vec{v} \cdot \vec{q} < 0 \end{cases} \Rightarrow \delta(q) = \begin{cases} 1 & \text{if } cos|\vartheta_c - \vartheta| \geq 0 \\ 0 & \text{if } cos|\vartheta_c - \vartheta| < 0 \end{cases}$$

where $\vec{v}$ is the velocity vector of the obstacle and angles ϑ_c and ϑ are defined as in Figure 2a. Each Gaussian function is computed as:

$$\Phi_{c,\Sigma}(q) = A\, exp\left\{ -\frac{[d\, cos(\vartheta - \vartheta_c)]^2}{2\sigma_x^2} - \frac{[d\, sin(\vartheta - \vartheta_c)]^2}{2\sigma_y^2} \right\}$$

where d is the Euclidean distance between q and $c(x, y)$, A is an amplitude parameter set to the maximum cost possible on the costmap, i.e., 255, and σ_x^2, σ_y^2 are the diagonal entries of the Σ covariance matrix, which determines the shape of the inflation region. In particular, the two covariance matrices are defined as follows:

$$\Sigma_{front} = \begin{pmatrix} \sigma^2_{x\_front} & 0 \\ 0 & \sigma^2_{y\_front} \end{pmatrix}$$

$$\Sigma_{back} = \begin{pmatrix} \sigma^2_{x\_back} & 0 \\ 0 & \sigma^2_{y\_back} \end{pmatrix}$$

Therefore, σ_x and σ_y can be tuned to model a generic shape at will. Here, in order to take into account the obstacle velocity magnitude, a maximum obstacle speed max_speed has been set. Then, in order to inflate more the front region, we defined the *speed ratio* $r = \frac{vel}{max\_speed}$, where vel is the estimated obstacle speed. Finally, the variances are modified according to the following heuristics:

$$\begin{cases} \sigma^2_{x\_front} = & (1 + r)\sigma^2_{x\_front} \\ \sigma^2_{y\_front} = & (1 - \frac{r}{2})\sigma^2_{y\_front} \\ \sigma^2_{x\_back} = & (1 - r)\sigma^2_{x\_back} \\ \sigma^2_{y\_back} = & (1 - \frac{r}{4})\sigma^2_{y\_back} \end{cases} \tag{1}$$

In this way, the Gaussian shape is lengthened in the direction of the obstacle motion and narrowed in the lateral area.

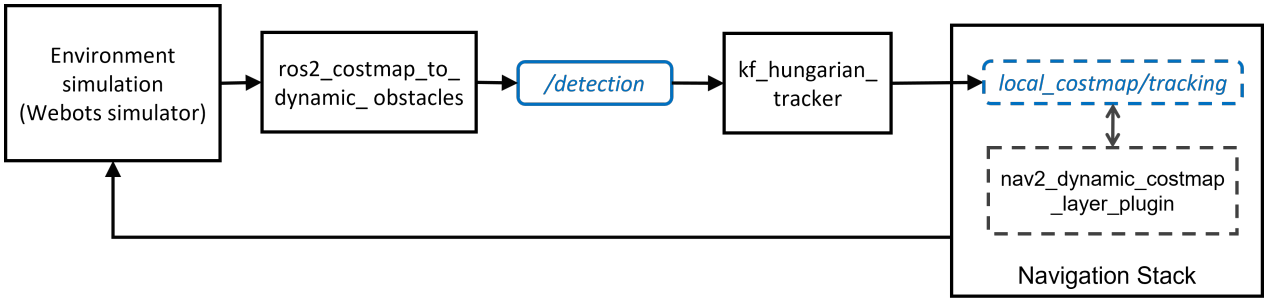


Fig. 3: Interaction scheme between internal SW functional blocks and the simulated external environment.

IV. Implementation in ROS2 framework

The presented approach has been implemented within ROS2 Foxy version on an Intel NUC8 with a Linux Ubuntu 20.04 environment. For testing purposes, during the development phase, a virtual environment has been created using the Webots simulator, while Rviz has been used for outputs visualisation and debugging. The simulated robot is the TurtleBot3, for which both Webots and Nav2 already provide a physical model and interface packages. Nevertheless, the DOL still remains independent of the robot which is considered. Concerning Nav2, due to its modular architecture, different configurations are possible, according to the specific plugins that are activated. Here, the default Nav2 configuration has been employed, according to the dedicated TurtleBot3 navigation packages, including the *nav2_dwb_controller* (DWB) as plugin for the Controller Server.

As it is shown in Figure 3, the *ros2_costmap_to_dynamic_obstacles* package provides the nodes implementing the obstacle detection functions and publishing the blobs corresponding to detected obstacles through a specific custom ROS2 message type on the */detection* topic. The *kf_hungarian_tracker* package provides a subscription to this topic, so that it can perform object tracking. Then, it publishes the dynamic obstacles and their estimated velocities on the *local_costmap/tracking* topic, which is created when Nav2 is launched. Finally, the costmap layer, implemented through *nav2_dynamic_costmap_layer_plugin*, processes this information to compute the Gaussian costs and updates the master costmap used for the robot navigation.

V. Simulation tests and analysis

In this section the path planning approach available in ROS2 (DWB Controller) and the DWB integrated with the proposed Dynamic Obstacle Layer method (DWB + DOL) are compared through simulations for a DOL preliminary evaluation.

The simulations have been run on Webots employing a TurtleBot3 burger robot, a two-wheeled robot equipped with a RPLIDAR A3 LiDAR sensor. In particular, the sensor provides a maximum distance range of $25\,m$, an angular resolution of $0.225°$ and a scan rate of $15\,Hz$.

A. Experimental setup

Webots allows to create realistic 3D virtual worlds including the physical properties of each object (Figure 4). In particular, it is possible to specify the dynamic behaviour of robotic objects (*Robot Nodes*) through a Webots Controller.

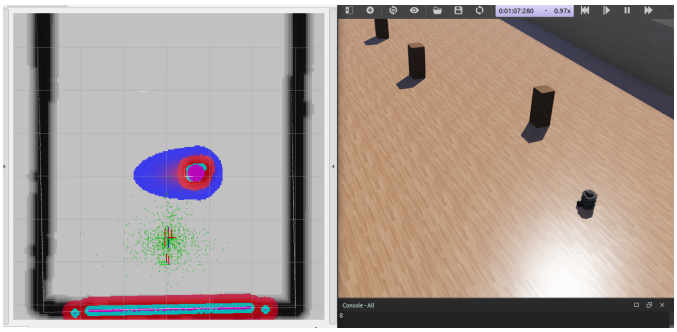


Fig. 4: *dyn_env_1.wbt* virtual world on the right and Rviz output on the left with DOL plugin.

The world created for testing is an empty rectangular arena $10\,m \times 6\,m$ where dynamic obstacles are simulated as wooden boxes of $20\,cm \times 20\,cm$ base and $50\,cm$ tall, configured as *Robot Nodes*. Starting from an assigned initial position, each box moves with a constant speed back and forth traversing the entire arena along the short side direction. A Webots Controller has been coded for each obstacle and the speed can be commanded when the world is launched, so that simulations scenarios can be easily modified. Note that, before launching the navigation tests, SLAM was performed, exploiting the *turtlebot3_cartographer* package, to obtain and upload the static map of the virtual environment without boxes.

In order to compare the performances of the baseline DWB with the DWB + DOL configuration, the TurtleBot3 is commanded to navigate in the *dyn_env_1.wbt* world N times, from one side of the rectangular arena to the opposite side, covering a total distance of $8\,m$, as shown in Figure 5. The Planner plugin used for computing the global path is *nav2_navfn_planner/NavPlanner*. A brief demo video showing the experimental setup and behaviour using DWB alone and DWB + DOL can be found at [28].

The parameters recorded for the evaluation purpose are: the travel time, the number of *wait* recovery behaviours triggered during travel and if any collision occurred. Such data have been collected from experiments conducted in two different conditions: obstacles (boxes) moving at constant speed set to $0.6\,m/s$ (Test set 1) and $0.8\,m/s$ (Test set 2).

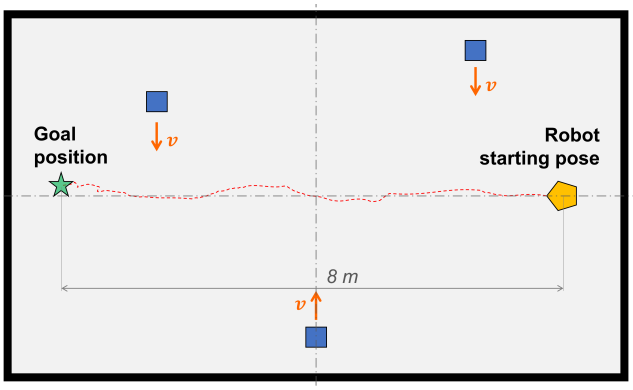


Fig. 5: *dyn_env_1.wbt* test scheme.

The performance indices considered for each set are described hereafter.

Smooth navigations It indicates how many times the TurtleBot3 smoothly navigated to the goal position. It includes also the cases in which the robot stopped briefly to avoid collisions with a moving box in its proximity.

Wait recoveries It is the number of times the *wait* recovery behaviour has been triggered, still successfully reaching the goal. A *wait* recovery is usually triggered when obstacles come suddenly too close and the Controller Server cannot find an affordable path by a given timeout interval.

Collisions It corresponds to the percentage of unsuccessful navigation due to a collision of the TurtleBot3 with a moving box.

Successful navigations It is the total amount of times in which the robot successfully reached the goal position, either performing a smooth navigation or after triggering the recovery behaviour.

B. Results and discussion

For the first test set, $N_1 = 50$ simulations have been launched for both DWB and DWB + DOL configurations. Table I sums up the global navigation results.

TABLE I: Navigation results at $0.6\,m/s$ obstacles speed.

	DWB + DOL	**DWB**
Smooth navigations	86,0%	82,0%
Wait recoveries	10,0%	0,0%
Collisions	4,0%	18,0%
Total successful navigations	**96,0%**	**82,0%**

As can be seen, the proposed method combined with DWB reported a greater successful navigation rate than the simple DWB: 96% against 82%, respectively. In particular, the 'smooth navigations' percentage is quite similar with the two approaches, but the DWB + DOL solution reported fewer collisions, because the *wait* recovery was triggered more times. This means that the dynamic obstacle layer provides a safer navigation if it is combined with the actual DWB planner. Indeed, the Gaussian costs forewarn the TurtleBot about an approaching obstacle and the Recovery Server is triggered on time if moving on would result in a collision. On the other hand, with the chosen obstacle speed ($0.6\,m/s$) and the same Nav2 parameters settings, DWB does not react on time if an obstacle suddenly approaches.

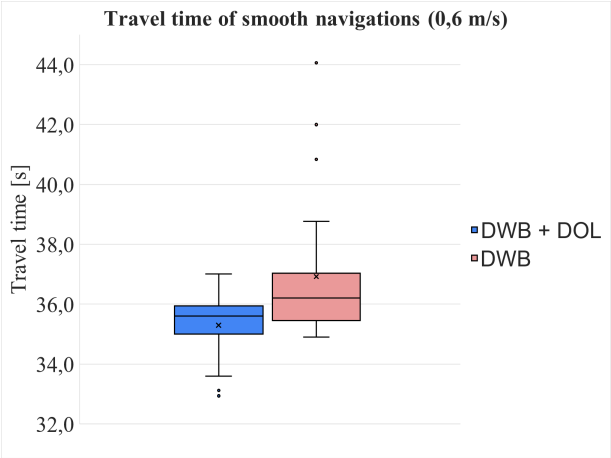
It can be noted that the number of 'smooth navigations' for DWB + DOL and DWB over the total number of tests is similar (86% and 82%, respectively), but the travel time performances are different. Figure 6a shows the box plots representing the travel time data of all the 'smooth navigations' achieved by the robot with the two approaches. First of all, one can notice that the DWB distribution is more asymmetric and for sure non-Gaussian. Secondly, the mean travel time reported in DWB + DOL is lower than that of DWB, even if only of $1\,s$. However, the most important result is that the interquartile range (IQR) for DWB + DOL is smaller than DWB box. Thus, it seems that the proposed method ensures to estimate a more confident travel time for a given environment and settings. Finally, it has to be noticed that outliers lay all below the box plot for the DWB + DOL (shorter travel times), while they are all longer travel times for the DWB plot. This suggests that carrying out more tests might produce less overlapped box plots.

For the second set of data, the obstacle speeds have been set to $0.8\,m/s$. This value has been chosen in order to push the available DWB Controller to its limits. Indeed, in this case, only $N_2 = 30$ simulations are sufficient to clearly prove the poor performance of the DWB compared to the DWB + DOL approach. Actually, the only difference in the settings and parameters of the whole environment – and the Nav2 parameters – with respect to the previous test set is the obstacle speed. The same performance indices have been taken into consideration and the results are shown in Table II.

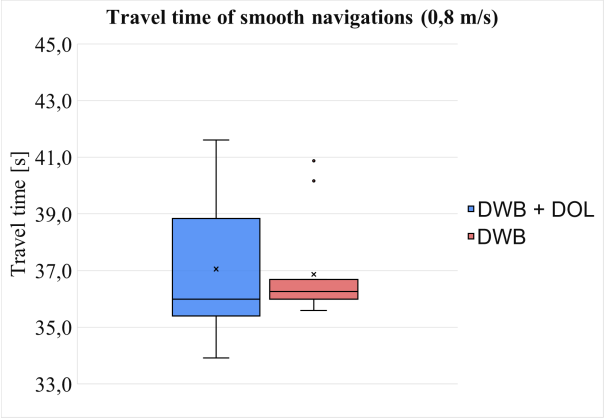
TABLE II: Navigation results at $0.8\,m/s$ obstacles speed.

	DWB + DOL	**DWB**
Smooth navigations	50,0%	43,3%
Wait recoveries	36,7%	0,0%
Collisions	13,3%	56,7%
Total successful navigations	**86,7%**	**43,3%**

It is worth noting that the number of collisions during the simulations launched with only DWB have remarkably increased (from 18,0% at $0.6\,m/s$ to 56,7%); likewise, for the DWB + DOL, collisions increased from 4,0% at $0.6\,m/s$ to 13,3%. Nevertheless, for the DWB + DOL case, the percentage of triggered recoveries has increased as well, ensuring 86,7% of successful navigations, while the percentage of success in case of DWB has halved with respect to Test set 1. Despite this, for the second test set, travel time data have been collected over $N_2 = 30$ data points (Figure 6b). Given that the 'smooth navigations' indices are equal to 50% (DWB + DOL) and 43,3% (DWB), it is worth pointing out that no robust consideration can be made.



(a) Test set 1



(b) Test set 2

Fig. 6: Box plot of the travel times during 'smooth navigations' for both test sets.

It can be noticed that the median values of both test sets are very similar. The first remarkable difference is that in Test set 2, due to the small amount of data, the DWB + DOL box plot has a greater variance. This is also because the faster are the obstacles, the more corrective actions are performed by the robot, making the travel time more unpredictable.

Concerning the data for DWB, the variance has been considerably reduced with respect to the previous test set. However, the data sample is too small, since the robot performed successfully a smooth navigation 13 times out of 30.

As it is shown by the simulation results, the DOL approach definitely reports some performance improvements in terms of collisions rate, but still collisions occur, even at the lower obstacle speed of $0.6\,m/s$. One reason behind this could be the not optimal communication between the Webots virtual environment and the Navigation Stack, that may cause delays in the obstacle detection and costs computation. However, this is strictly related to the computational performances of the whole ROS2 and Linux environment installed on the Intel NUC platform.

Despite this, some Nav2 parameters could be tuned to reduce the collisions rate with the adopted HW/SW setup. Referring to the *Configuration Guide* section of DWB Controller in the Nav2 documentation page, they are:

- *controller_frequency* (default $20\ Hz$): it corresponds to the controller server update rate. Higher values may lead to a faster reaction to obstacles since the local trajectory is replanned more frequently by the DWB Controller plugin.
- *update_frequency* (default $5\,Hz$): updating more often the `local_costmap` allows the robot to read more recent Gaussian cost values. In this way, the local path planning is computed using more reliable data, improving obstacle avoidance.
- *<dwb plugin>.sim_time* (default $1.7\ s$): it is the time in which the DWB plugin simulates looking ahead to generate affordable local trajectories before scoring them and choosing the best one. Slightly increasing this parameter, the DWB Controller should better discriminate colliding trajectories from non-colliding ones. Nevertheless, since the DWB is a local planner, higher values may prevent the correct performance of the Planner Server.
- *<dwb plugin>.BaseObstacle.scale* (default 0.02): it is the scale used by the DWB plugin to score a trajectory and it depends on the location of the path in the costmap. As the value raises, it is more likely that the robot will avoid passing through inflated cells. As a result, the navigation should be smoother but longer.

Modifying the above listed parameters, especially the first three, may be a valid option to improve the overall performance. However, higher computing resources may be needed, so the parameter values should be finely tuned according to the actual target hardware and application requirements.

For what concerns the Gaussian cost assignment, the variances (Gaussian shape parameters), have been set empirically and based on an hypothetical obstacle maximum speed as in (1). As alternative, an additional function could be introduced to compute the costs combining the obstacle speeds with the robot speed. For instance, if the robot maximum speed is too small with respect to the obstacle one, it should not try to pass over it, so Gaussian costs should be modulated accordingly. On the other hand, the obstacle speed information could be directly included into the local planner, but at the price of achieving a much more complex approach, no more as modular as the proposed DOL.

Therefore, the proposed Dynamic Obstacle Layer approach seems to provide a safer navigation in presence of dynamic obstacles. At the same time, in the majority of the test cases, the DOL allows to plan a smoother trajectory than the DWB Controller alone, resulting in reduced travel times starting from the same conditions. Indeed, despite the TurtleBot has a maximum speed lower than the set obstacle speeds, it manages to adjust the trajectory when an obstacle is reported by the DOL, dodging it or passing behind it, even if there is still much room for improvements of the travel time performance.

VI. Conclusions and future works

The proposed Dynamic Obstacle Layer approach implements a strategy for dynamic obstacle handling that can be easily integrated with the current ROS2 Navigation Stack, thus being a flexible and modular solution for the problem of navigation in dynamic environments. The simulation tests carried out in a virtual environment have shown a relevant performance improvement in terms of collisions rate and travel times, integrating the DOL with the available DWB Controller. The two carried out simulation tests, highlighted how the combination of DWB and DOL can improve navigation safety as the dynamic obstacles' speed increases.

Nevertheless, the DOL code involves many parameters (e.g., filters parameters, blob detection parameters, Gaussian costs scaling factors, etc.), some of which have been set based on intuition and manual tuning in this initial implementation.

Therefore, this paper represents the starting point of various possible future works in different directions. First of all, the DOL has been tested only in a virtual environment. Even though Webots has been set to provide as realistic simulations as possible, only tests in real world can validate the proposed approach, i.e., using a physical TurtleBot with real sensors and checking the actual performance of the robot under particular scenarios. Beyond that, object detection is based on the thresholds set for the running average filter: using the values based on reference examples, however, does not allow to achieve the desired filtering accuracy for low obstacle velocities. Thus, a set of tests should be carried out to finely tune these parameters according to the hypothetical future application scenarios.

Finally, the incorporation of camera information can be investigated to provide more detailed information about dynamic obstacles, e.g., discriminating a walking person from another robot, so that the robot could react accordingly with different planning strategies. Semantic information can be easily integrated in a multi-layer costmap on the base of the current DOL, and Planner and Controller plugins can be easily foreseen to implement different behaviours.

References

[1] R. Arkin and R. Murphy, "Autonomous navigation in a manufacturing environment," *IEEE transactions on robotics and automation*, vol. 6, no. 4, pp. 445–454, 1990.

[2] F. Rubio, F. Valero, and C. Llopis-Albert, "A review of mobile robots: Concepts, methods, theoretical framework, and applications," *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, p. 172988141983959, 2019.

[3] S. M. LaValle, *Planning Algorithms*. USA: Cambridge University Press, 2006, ch. Introductory Material.

[4] L. Claussmann, M. Revilloud, D. Gruyer, and S. Glaser, "A review of motion planning for highway autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 5, pp. 1826–1848, May 2020.

[5] P. Marin-Plaza, A. Hussein, D. Martin, and A. d. l. Escalera, "Global and local path planning study in a ROS-based research platform for autonomous vehicles," *Journal of Advanced Transportation*, vol. 2018, 2018.

[6] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 300–307.

[7] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23–33, 1997.

[8] S. Quinlan and O. Khatib, "Elastic bands: connecting path planning and control," in *[1993] Proceedings IEEE International Conference on Robotics and Automation*. IEEE Comput. Soc. Press, 1993, pp. 802–807 vol.2.

[9] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram, "Trajectory modification considering dynamic constraints of autonomous robots," in *ROBOTIK 2012; 7th German Conference on Robotics*. VDE, 2012, pp. 1–6.

[10] B. Cybulski, A. Wegierska, and G. Granosik, "Accuracy comparison of navigation local planners on ROS-based mobile robot," in *2019 12th International Workshop on Robot Motion and Control (RoMoCo)*. IEEE, 2019, pp. 104–111.

[11] I. Naotunna and T. Wongratanaphisan, "Comparison of ROS Local Planners with Differential Drive Heavy Robotic System," in *2020 International Conference on Advanced Mechatronic Systems (ICAMechS)*. IEEE, 2020, pp. 1–6.

[12] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10.

[13] Why ROS 2? Accessed: April 2022. [Online]. Available: https://design.ros2.org/articles/why_ros2.html

[14] S. Macenski, F. Martin, R. White, and J. G. Clavero, "The Marathon 2: A Navigation System," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 2718–2725.

[15] Navigation Dynamic Obstacle Integration. Student Program challenge launched in 2021 by the Nav2 community. Accessed: April 2022. [Online]. Available: https://navigation.ros.org/2021summerOfCode/projects/dynamic.html

[16] M. De Rose, "LiDAR-based Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2," Master's thesis, Politecnico di Torino, 2021.

[17] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An introduction*. CRC Press, Jul 2018.

[18] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 709–715.

[19] I. H. Savci, A. Yilmaz, S. Karaman, H. Ocakli, and H. Temeltas, "Improving Navigation Stack of a ROS-Enabled Industrial Autonomous Mobile Robot (AMR) to be Incorporated in a Large-Scale Automotive Production," *The International Journal of Advanced Manufacturing Technology*, pp. 1–22, 2022.

[20] F. Martín, J. G. Clavero, V. Matellán, and F. J. Rodríguez, "Plansys2: A planning system framework for ROS2," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 9742–9749.

[21] H. Dong, C.-Y. Weng, C. Guo, H. Yu, and I.-M. Chen, "Real-Time Avoidance Strategy of Dynamic Obstacles via Half Model-Free Detection and Tracking With 2D Lidar for Mobile Robots," *IEEE/ASME transactions on mechatronics*, vol. 26, no. 4, pp. 2215–2225, 2021.

[22] M. Przybyła, "Detection and tracking of 2D geometric obstacles from LRF data," in *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, 2017, pp. 135–141.

[23] T. Mori, T. Sato, H. Noguchi, M. Shimosaka, R. Fukui, and T. Sato, "Moving objects detection and classification based on trajectories of LRF scan data on a grid map," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010, pp. 2606–2611.

[24] F. Albers, C. Rösmann, F. Hoffmann, and T. Bertram, *Online Trajectory Optimization and Navigation in Dynamic Environments in ROS*. Springer, 01 2019, pp. 241–274.

[25] A. Pierson, W. Schwarting, S. Karaman, and D. Rus, "Navigating congested environments with risk level sets," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 5712–5719.

[26] Blob Detection Using OpenCV. Accessed: April 2022. [Online]. Available: https://learnopencv.com/blob-detection-using-opencv-python-c/

[27] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[28] DOL: A costmap-based Dynamic Path Planning approach in ROS2. Accessed: April 2022. [Online]. Available: https://youtu.be/CrZ6VolJpwk