Path Planning– Matteo Villani – 12343046

1.1 - Lateral Control

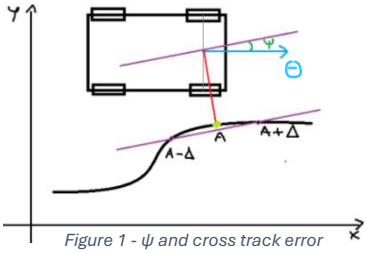

Figure 1 - ψ and cross track error

The lateral control law aims to minimize the **heading error (*ψ*)**, defined as the difference between the tangent angle and the **actual direction of the robot (Θ)**; and the cross-track error, **e(t)**, (red in the Fig.1): the distance between the middle point of the front axle of the robot to the **closest pose ('A')** among the ones in the `global_plan_`. In the Fig.1 the point A represents the closest pose, from which a **displacement** (**Δ** in the Fig.1) of 15 has been used to retrieve the poses 'A- Δ' and 'A+ Δ' that allow to compute the tangent angle (purple line) to the path. The goal is to write the computed driving commands (**linear velocity and steering angle $\delta(t)$**) into `cmd_vel`. To compute them the following control law is used [1]:

$$\delta(t) = \begin{cases} \psi(t) + \arctan\frac{k\,e(t)}{v(t)} & \text{if } |\psi(t) + \arctan\frac{k\,e(t)}{v(t)}| < \delta_{max} \\ \delta_{max} & \text{if } \psi(t) + \arctan\frac{k\,e(t)}{v(t)} \geq \delta_{max} \\ -\delta_{max} & \text{if } \psi(t) + \arctan\frac{k\,e(t)}{v(t)} \leq -\delta_{max} \end{cases}$$

Where **v(t)** is the **current** velocity and **K** represents the control gain: k determines the influence of the cross-track error on the steering angle. A higher value makes the robot more responsive to cross-track errors but may cause oscillations. If the robot oscillates, reduce k. If the robot is too sluggish in correcting its path, increase it.

At this point, the robot was not computing the right steering angle since the control law was not minimizing the errors, thus, it was diverging form the trajectory. I spent some time on this to find the errors, and there were many: firstly, the **current** velocity has to be used in the formula, not a constant value that I was using both for the `cmd_vel.linear.x` command (that is correct) and for the formula (wrong). So, the magnitude of the linear velocity vector, retrieved from the 'twist' object in the odometry at each iteration, is now **v(t)** in the formula.

Now, the approach seems to work better, but only when the robot has to reach a goal on the **left** side of its current position. This problem is now related to **e(t)**, since I was computing it as a positive distance between 2 points: in this way the robot consider itself always on the right with the respect to the path, so I had to compute the **signed cross-track error** to keep track the relative position of the robot with the respect to the planned path:

```
double cross_track_error_signed = crossT_error * (dx_robot * dy - dy_robot * dx >= 0 ? 1.0 : -1.0);
```

After this implementation, the robot was correctly following the path, successfully reaching the goal. A simple improvement has been made [2]: to cope with problems at low speed and noisy velocity measurements a *softening constant* (**Ks = 0.5**) has been implemented in the formula letting the robot perform better performance at low speed:

$$\delta(t) = \psi(t) + \arctan\left(\frac{ke(t)}{k_s + v(t)}\right)$$

To fine-tune the parameter **K** I started low values like 0.01, that lead to the robot to be very slow to response to the track error, then 0.1 has been used leading to better performance but still slowly diverging. Finally with the value 1 the robot was on the right way to follow the trajectory. With higher

values like 10, the robot was oscillating and diverging from the path. A final satisfying value of **1.25** is used (the final implementation is robust to higher values also).

1.2 – Longitudinal Control

The primary goal of longitudinal control is to minimize the error between the actual and reference linear velocity. The **reference velocity** $\boldsymbol{v_d}$ is set to the maximum allowable speed of the robot, found by printing the 'twist' object while driving the robot on a straight line with the keyboard control. The **velocity error** $\boldsymbol{e_v}$ is defined as: $e_v = v_d - v$, where **v** is the **current velocity** previously computed. A Proportional-Integral (PI) controller is used to determine the acceleration command ***a*** based on the velocity error. The control law is given by:

$$\boldsymbol{a} = K_p * e_v + K_i \int e_v dt$$

Where $\boldsymbol{K_p}$ and $\boldsymbol{K_i}$ are the proportional and integral gains, respectively. The integral term accumulates the velocity error over time to eliminate steady-state error.

1.3 – Compute Command Velocity

The previously computed acceleration is clamped within constrained boundaries (-0.3, 0.3):

```
acceleration = max(ACC_MIN, min(ACC_MAX, acceleration));
```

Using the formula: $\boldsymbol{v} = v(t) + a * \Delta t$ it's possible to compute the velocity command that minimizes the error between the desired velocity and the actual one. Δt represents the time elapsed between one iteration and the other. Since the frequency for the controller is set to **20Hz** (parameter found in 'move_base.launch') the delta time between one iteration and the next one should be ~0.05s. The following solution has been implemented to let the integral error to be computed and never be neglected (even if the delta_t will always be ~0.05), clamping it between the range [0.01, 1] (for the max 1s because the initialization is done in the *initialize* function and from that point till a new goal is assigned to the robot the delta_t keeps counting, e.g. if the goal is given after 20s after launching the node, delta_t will have the value of ~20s for the first iteration):

```
delta_t = max(0.01, min(delta_t, 1.));
```

The parameters $\boldsymbol{K_p}$and $\boldsymbol{K_i}$ are essential components of the PI: the **proportional gain** determines the reaction to the current error. It applies a correction based on the present value of the error: a higher value, like >~5 in my case, increases the system's responsiveness, causing it to react more aggressively to the error. This can help the system reach the desired state more quickly, but the system was unstable overshooting the target. With lower values, like ~0.01 the system was responding slowly and taking longer to reach the goal. A final, satisfying value of **1.3** is set.

The **integral gain** accumulates the error over time and applies a correction based on the accumulated error. It helps to eliminate the steady-state error (the persistent difference between the desired and actual values). Like the other gains, a high value of the integral gain increases the influence of the accumulated error, which helps to reduce the steady-state error more quickly, but if too high (>~4) it can

lead to excessive overshooting and instability. If K_i is too low (<~0.1) the systems take long time to correct the any error. A final, satisfying value of **0.5** is set.

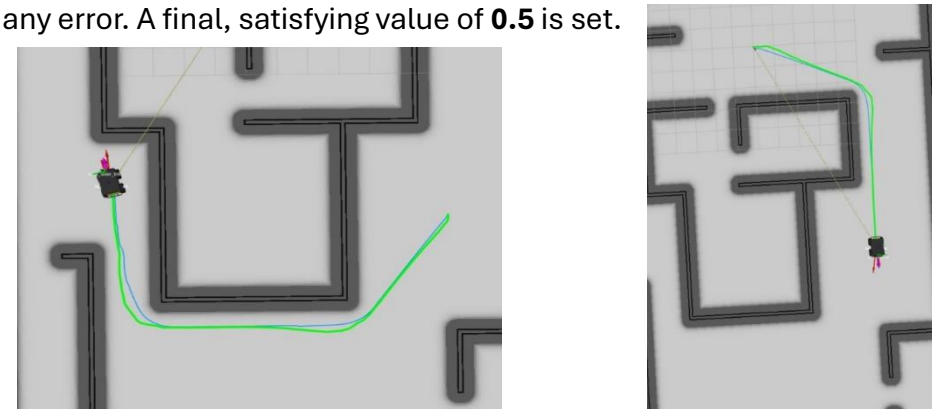

Figure 2,3 - Example of Trajectory following

In Fig. 2,3 we can notice the planned path and the actual trajectory performed by the robot.

1.4- Check for collisions

This task requires keeping the robot from colliding with obstacles in the environment. To achieve this an inspection to the map information stored in the `costmap_2d::Costmap2DROS costmap` variable. The idea is to check at each iteration, if the **predicted pose for the next iteration** will get in touch with some forbidden cells of the *costmap*.

To retrieve the predicted pose the following transformation is applied:

```
// Get the predicted pose after applying the velocity command
geometry_msgs::Pose predicted_pose = odom_.pose.pose;
predicted_pose.position.x += cmd_vel.linear.x * cos(tf2::getYaw(predicted_pose.orientation)) * delta_t;
predicted_pose.position.y += cmd_vel.linear.x * sin(tf2::getYaw(predicted_pose.orientation)) * delta_t;
```

Then, in the function *isCollisionFree* it's checked that the robot doesn't fall into some occupied cells. To perform this check, an inspection to the robot's **footprint** (the dimension of the robot) is made retrieving that information from the *costmap* as well. For each point of the footprint, the transformation to the robot's predicted pose is applied. Then I convert the coordinates of the transformed points into cells coordinates with the *worldToMap* function. If the cells coordinates are being returned means that the conversion is successful, otherwise the robot would end up out of the map boundaries. Retrieved the cell, for each footprint-point, it's time to retrieve its cost, thus the value that defines whether that cell is FREE_SPACE, LETHAL_OBSTACLE or INFLATED_OBSTACLE. Performing the check with an if-statement for each cell retrieved we can infer if the robot is going to collide with an obstacle or not. If the predicted pose encounters an obstacle or an inflated obstacle cell, the robot stops by assigning 0 to linear/angular velocity commands, thus achieving the task.

1.5 – Consider new obstacles

Having achieved all the task above the robot Is able to reach a given goal in the map by planning and correctly/smoothly follow the planned trajectory. But the robot considers only the obstacles already present in the map: if we introduce new obstacles (by the simulation in Gazeboo) the robot is not able to update the cost map and thus plan the trajectory accordingly.

To let it be robust in terms of dynamic obstacle I needed to improve the *move base* navigation architecture. Firstly, I decided to implement the dynamic window approach by creating another file (*dwa_local_planner.yaml*) and trying to bend it to the actual *move_base.launch*, but this approach resulted too much complicated without proving good results: the robot just crush into the dynamic added obstacle.

Then, I decided to dive into the concept of **plugin** seen into the launch file. After some research [3]-[4], I realized that I was overengineering the solution. I implemented the following plug in into the **Global** cost map:

```
            - {name: obstacle_layer,     type: "costmap_2d::ObstacleLayer"}
```

I realized also that the order between *obstacle_layer* and *inflation* layer matter a lot: the obstacle plugin must be defined **before** the inflation layer plugin.

I then defined the *obstacle_layer* as:

```
obstacle_layer:
    observation_sources: laser
    laser: {sensor_frame: front_laser_link, data_type: LaserScan, topic: /front_laser/scan,
marking: true, clearing: true}
```

By looking at what happens in the simulation when new obstacles are added I notice that the Planner notices the obstacle thanks to the laser but doesn't generate a new plan. This problem was related to both frequencies of the local/global planner. So I decided to increase the publish frequency up to a value of **5Hz** (I didn't want to increase this too much in order to avoid interference with the controller's frequency) for the local planner, since it's the one in charge of making new plans according to the costmap.

At this point I noticed that the robot was able to detect new obstacles and generate a new plan accordingly. But the planned trajectory is too close to the inflation area of the obstacles, this led the robot to stop for the task 1.4, since the footprint was touching the inflation of new obstacles. To avoid this, I had to edit the **inflation** parameters:

```
inflation:
       inflation_radius: 2.5
       cost_scaling_factor: 9.0
```

I applied this modification also to the **local** cost map and I obtained a fully dynamic obstacle-responsive robot that follows a trajectory to a given goal avoiding obstacles.

References

[1] – Thorpe, C., Thrun, S., Montemerlo, M., & Stentz, A. (2006). The Stanley controller: Model predictive control for high-precision trajectory tracking. In *Proceedings of the 2006 IEEE Intelligent Vehicles Symposium* (pp. 161-166). IEEE.

[2] – Slide Path Planning II

[3] - https://answers.ros.org/question/321578/unable-to-clear-traces-of-obstacles-on-costmap/

[4] - https://answers.ros.org/question/257286/obstacles-are-not-cleared-completely-in-costmap/