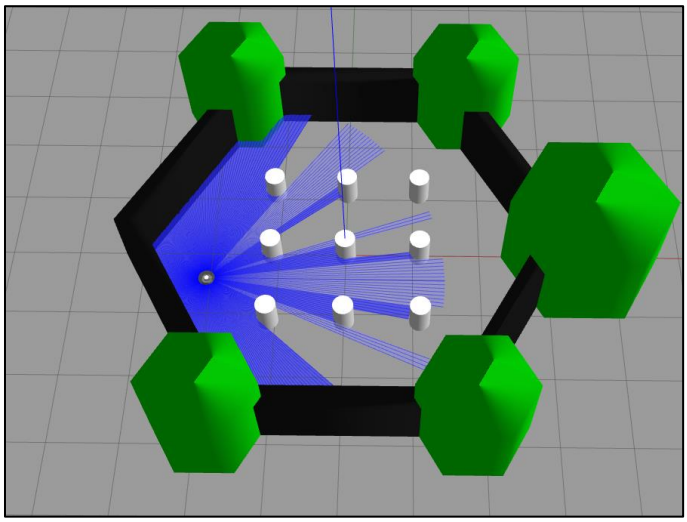
Project Manas Final Task (AI and Automation)

Auto-Nav

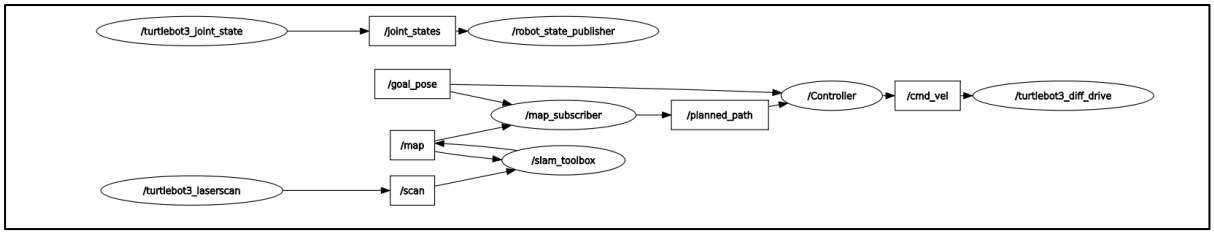
Documentation



Deliverables:

- Velocity Commands - The program should output velocity commands that allows the bot to navigate while avoiding obstacles
- Motion Constraints - The bot's movements must be smooth and not jerky
- Tech Stack - The module must be written in cpp and ros2 wrapped.
- Demonstration - The bot's movements to a provided goal must be visualized along with the path it has chosen to follow.
- Publishing Rate - The velocity commands must update at least at a rate of 15 Hz

rqt_graph:



Node /map_subscriber

Subscriptions:

1. /map : for occupancy grid.
2. /goal_pose: for goal information.

Publishing:

1. /planned_path

Working:

The path planner node uses A* algorithm to determine the best path to the goal. The goal is set using /goal_pose. The A* is upgraded to consider the robot position as start effectively making it D*. This helps the robot to traverse its dynamic environment (dynamic with respect to the robot as the map updates with Lidar sensor).

A* algorithm (Theta*):

Cushioning function: Used to inflate the obstacles to account for the robot dimensions. The blocked coordinates are kept in a list and later, this function encodes two units of pixels in all directions as obstacles.

Find Path function: Uses A* algorithm to find path. Key improvements – if the goal is set on an obstacle, it returns way point with robot coordinates, handling invalid goal conditions.

Post processing:

The way points generated by A* is put through Collinearity based smoothing algorithm where the points are dropped if it lies along the same line joining the previous and next point. The slopes are compared and if the difference between the slopes are under defined tolerance the point is dropped. This reduces redundant points which will help later during curve fitting.

Curve Fitting:

As the path is visible zig zag, there was an attempt to smooth it out. For this Bezier curve fitting is used. However, this soon brought other problems:

1. Path contains too many points: How is this a problem? To answer this, we must look into the controller part of the robot. The Robot uses Pure pursuit damped by PID, and it is to be kept in mind that the path is being planned dynamically so the bot position is being updated as it moves. Considering all these constraints, we can see that a curve over a small length dx will have very small change in slope i.e. It looks like straight line. So, to the bot, as it cannot see the whole path ahead, it moves in a straight line and get immune to subtle angle changes. This is problematic when there is a gradual turn must be taken. As the Bot will not change direction until the difference in angle is very large but now it's too near the obstacle to change course or due to curve fitting it takes large turning radius. This could have been solved with foresight in velocity control. Where more than one point is considered in velocity control and their linear combination is used to get final velocity. Again, this is redundant as we are dealing with velocities up t +- 0.2m/s, multiplying this with a number between 0 and 1 makes it even smaller making its contribution in velocity control negligible.

2. Another logical explanation could be that in due to the resolution of world we are working with (i.e. 1 unit = 0.05m) every quantity is small in magnitude; the points are too near. There are some other possible reasons for this behaviour: control points are too close to each other, control points are not aligned with the overall path direction, trying to fit a Bezier over too many waypoints (this can be solved with collinearity based smoothing but again the small scale makes this change not much noticeable)

The motion smoothness has been addressed by motion controller, discussed further ahead.

A wall timer is used to run the path planning algorithm every 67 milliseconds (15Hz). This will come handy later.

Node /Controller:

Subscriptions:

1. /planned_path: for way points.
2. /goal_pose: for goal coordinates.

Publishing:

1. /cmd_vel

This node subscribes to /planned_path, which contains way points by the A* algorithm. These way points are used to traverse around obstacles to get to the goal.

PID (Proportional Integral Derivative) Controller:

The PID controller is used to generate longitudinal controls for the robot. It has been tuned to avoid oscillations and understeering. The velocity is maintained in range +- 0.2 m/s. As the Robot reaches set distance from the goal the controller outputs 0 m/s to stop it.

The longitudinal controller is a hybrid of pure pursuit damped by PID. However, to make it move smoother over a curve and have a foresight, Lookahead controls were also tested. Where predicted velocity on next 3 points is calculated damped by PID then combined in form of linear combination, their coefficients numerically representing their combinations. But as discussed earlier, the contribution of further ahead points is negligible as (multiplying a number between +-0.2 with a number between 0 and 1). This also affects the robot's ability to handle aggressive turns (especially when right next to obstacle).

Stanley Controller:

Initially PID controller with one axis output was used to give yaw commands, but due to instability and difficulty in tuning the controller, Stanley controller is used. Stanley outputs heading error as the output for very low or 0 velocity. This caused problems as the Stanley was

immediate at correcting yaw, but PID was slower at giving velocity commands, causing it to go into oscillation and move in reverse.

To avoid this PID-Stanley controller was adapted, the output of Stanley fed into PID to control the speed and smoothness of angular velocity commands. But the PID controller was difficult to tune and caused more oscillations.

At the end Staney controller with capped angular velocity is utilised. The algorithm is set such that if the heading error is very aggressive (90 degrees>) the longitudinal velocity is set to 0 and the bot is allowed to take turn. The angular velocity is set to +- 45 degrees, but is the velocity is very less it can take up to 90 degrees turn. This solved the problem of oscillations induced by aggressive yaw outputs.

/cmd_vel frequency of publishing:

As the path is being published at 15Hz, the /planned_path subscriber gets updated with new way points at 15Hz, enabling it to publish velocity commands at same rate.

Conclusion:

Curve fitting, collinearity-based smoothing are very important in working of an autonomous robot but in this case due to scale constraints they don't seem to improve the performance. I believe there are better path smoothing algorithms and curve fitting techniques other than listed above that can be used for better performance in planning and motion control.

The current model uses Theta* to for optimised path planning. Foresight motion control damped with PID for longitudinal control, Yaw control by Stanley capped to avoid oscillation. The curve fitting has been included but not used in final implementation.

Commands in use:

- Launch world and gazebo: ros2 launch autonav robot.launch.py
- Launch slam toolbox : ros2 launch slam_toolbox online_async_launch.py
- Run the node for path planning: ros2 run navigation_cpp map_subscriber
- Run the motion controller node: ros2 run command_vel motion_controller
- Launch Rviz: rivz2

Sincerely,

Chirag Rao KV

REG: 240962180

Gmail: chiragrao.kv@gmail.com

GitHub: chiragraokv