Hierarchical Planning for Indoor Mobile Robots in a Dynamic Environment

Project Report

Submitted by
Sai Rohitth Chiluka
ED18B027

in partial fulfilment of requirements
for the award of the dual degree of

BACHELOR OF TECHNOLOGY in
ENGINEERING DESIGN

and

MASTER OF TECHNOLOGY in
ROBOTICS

DEPARTMENT OF ENGINEERING DESIGN
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI 600 036

May, 2023

CERTIFICATE

This is to certify that the project titled **Hierarchical Planning for Indoor Mobile Robots in a Dynamic Environment**, submitted by Sai Rohitth Chiluka **(ED18B027)** to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology in Engineering Design and Master of Technology in Robotics**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Name of Guide: Prof. Bijo Sebastian

Signature:

Name of student: Sai Rohitth Chiluka

Signature 

ACKNOWLEDGEMENT

I would like to express my deep and sincere gratitude to my guide Prof. Bijo Sebastian, for giving me the opportunity to work under him and providing me with invaluable guidance throughout.

My completion of this project could not have been accomplished without the support of my labmates, Gokul, Hari Shankar and Naren. Thanks for helping me whenever needed with my hardware and ROS related concerns.

Finally, I would like to thank the Department of Engineering Design and the Autonomous Systems Laboratory (ASL) for providing me with the opportunity and the resources needed to fulfill this IDDD Robotics Project.

ABSTRACT

KEYWORDS: Autonomous Mobile Robots, Hierarchical Planning, Dynamic Obstacles

Autonomous Mobile Robots are gradually becoming a reality in many applications. The most important Decision Making module that goes into it is Planning and Navigation. This work tries to implement a Hierarchical Planning algorithm to navigate around Dynamic Obstacles. First, a Global Planner operates on a low resolution Occupancy Grid, using the A* algorithm, to create a high level plan in the form of waypoints that the bot should follow. A Local Planner then operates within the high resolution perceptive field around the bot to plan for uncertainties in the present state. The Local Planner uses an Ego-Graph approach to generate locally optimal paths described by Motion Primitives that are easy to follow and optimized for various cost functions like cross track error and curvature error. Dynamic Obstacles are handled by forward propagating the current motion of said obstacles and then performing a collision check through time. The aim is to reach the goal moving from waypoint to waypoint while avoiding static and dynamic obstacles. Algorithms to reject unnecessary waypoints are also used. All of the above is simulated and tested using a Differential Drive based Indoor Mobile Robot.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

ROS - Robot Operating System

SLAM - Simultaneous Localization and Mapping

LIDAR - Light Detection and Ranging

ASL - Autonomous Systems Lab, ED Dept.

ED - Engineering Design Dept., IIT Madras

UAV - Unmanned Aerial Vehicle

YAML - Yet Another Markup Language (Config File)

PGM - Portable Gray Map (Image Format)

CHAPTER 1 : INTRODUCTION

1.1 Background

Robots are useful in a variety of areas like Defence, Space Exploration, Supply Chain, Transportation and even in our Homes. While there are remotely controlled robots that have proved to be useful in various situations, they have been limited either by the skills of the pilot or the range and latency of communications. In some tasks, the information processing may be too fast for humans to interpret and hence the pilot may not be of much use. In these cases it becomes important for a robot to function on its own, i.e., act autonomously.

Making a robot autonomous comes with its own set of challenges like figuring out where the bot is, figuring out where to go and how to get there and figuring out how the surroundings interact with the bot. One of the most common tasks of an autonomous mobile robot is to plan and execute a path from a source to a target destination. There has been considerable effort in creating algorithms for a static environment but many of them are not able to give similar results in a dynamic environment with moving obstacles.

This is what this work tries to address. Past work on this front include [4] Markov Chain Position Prediction combined with an Improved Artificial Potential Field, [5] Ant Colony Optimization using Dynamic Window Approach as the local planner, [6] Dynamic Window Approach combined with Virtual Manipulators for obstacle path prediction.

1.2 Objective of the Work

This work plans to implement a Hierarchical Planner using an Astar Global Planner along with an Motion Primitives and Ego-Graph [1] based Local Planner. Motion Primitives are desired because it gives us Kinematically feasible paths that makes it possible for lower level controllers to track them more easily.

Chapter 2 talks about the Global Planner, how the map is processed, the A Star algorithm and how unnecessary waypoints are discarded. Chapter 3 talks about the Local Planner, how the Motion Primitives and Ego-Graphs are generated and how the optimal one is picked and how dynamic obstacles are handled. Chapter 4 talks about Mapping the ED Dept. using Hector SLAM. Chapter 5 talks about the Perception stack used to detect and track dynamic obstacles. Chapter 6 talks about the code implementation using Python, ROS and Gazebo. Chapter 7 shows the results produced. Chapter 8 talks about any future work and improvements possible and concludes the project.

CHAPTER 2 : GLOBAL PLANNER

Given a source location and a destination location an autonomous robot has to plan a high level path without bothering about the finer details. The reason for the latter is two-fold. One, the computation gets really high as we start increasing resolution. Two, the map and the environment may keep changing. So it is not worth the computational cost unless you know for 100% certainty where each obstacle on the map is going to be at each instant of time. Hence we let the local planner take care of the finer details.

The below section talks about the algorithms involved in creating the global planner.

2.1 Processing the Map

2.1.1 Downsampling

Lidar point cloud data is used to generate the map used for planning. This can be done using ROS' gmapping node. This is then downsampled by a specified factor by using Inter Area Interpolation. The result is then Binary Thresholded to represent the obstacles.

2.1.2 Costmap

The above map can directly be used but it would result in paths that are very close to obstacles as it wants to minimize the cost as much as possible. A solution to this is to use a costmap, i.e. some regions should be harder to reach than others. This is accomplished by convoluting the previous map with a Gaussian Filter and then combining the both of them by placing them on top of each other. The below image shows the previous and final map after processing it.

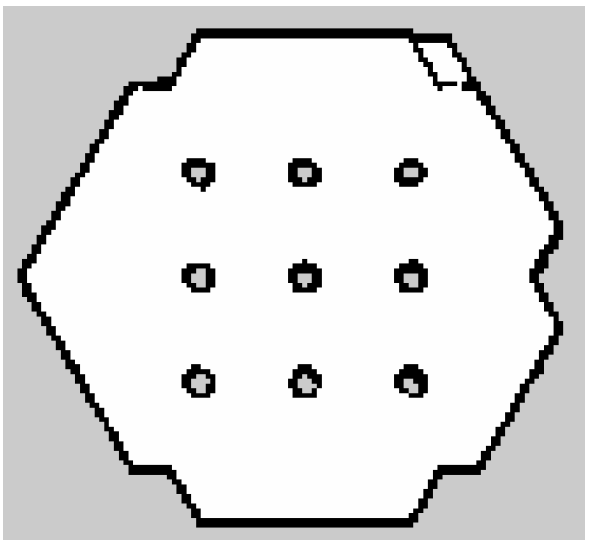

Fig. 2.1: Original Map

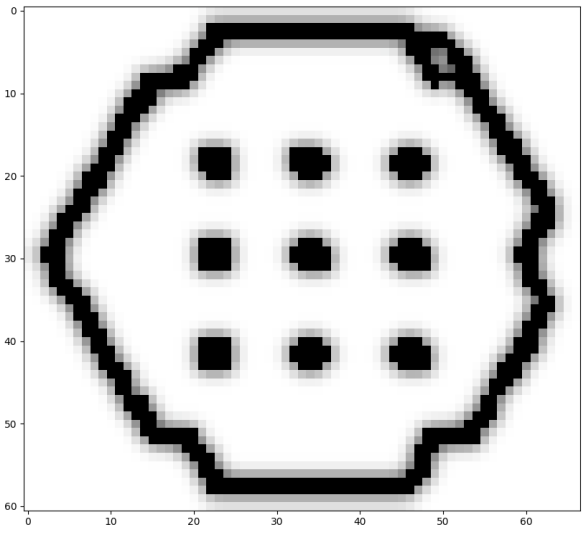

Fig. 2.2: Processed Map

2.2 A Star Algorithm

A Star is one of the most famous planning algorithms as it provides us with optimality guarantees. It is a graph search algorithm which works by expanding that node which has the least cost f where,

$$f(n) = g(n) + h(n)$$

g here represents the accumulated cost from the start and h represents the heuristic to the goal. For a Graph based A* to be optimal the heuristic has to be consistent, whereas for a Tree based A* admissibility is enough. For a heuristic to be consistent it has to satisfy the following, where N is the current node, P is the next node, G is the goal node and c is the cost to go from N to P.

$$h(N) \leq c(N, P) + h(P)$$
$$h(G) = 0.$$

The consistent heuristic can be something like Euclidean distance, Manhattan distance or others. The below image shows the resultant waypoints of performing A* with manhattan distance as the heuristic on the map shown above.

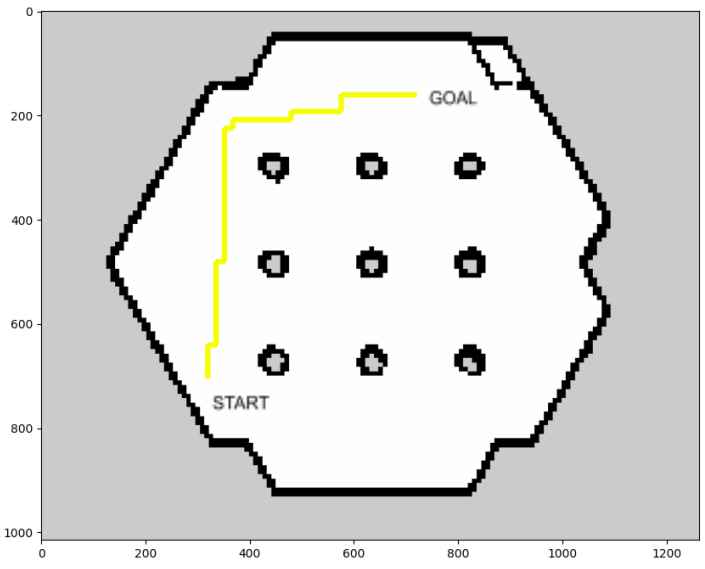

Fig. 2.3: Global Path

2.3 Unnecessary Waypoint Rejection

This step of the algorithm is more targeted towards the local planner than the global planner. The aim of the local planner is to move from one waypoint to the next. The waypoints act as global navigators hence they should be located only at important points on the path. The rest of them are unnecessary as they add nothing but more memory and computation.

If 2 consecutive waypoints move in the same direction, i.e., they have the same slope then the first point can be removed. This is repeated for the whole path until only the important points are left. The below images show the waypoints before and after this step. The cells coloured in red are the waypoints and the cells coloured in yellow are not.

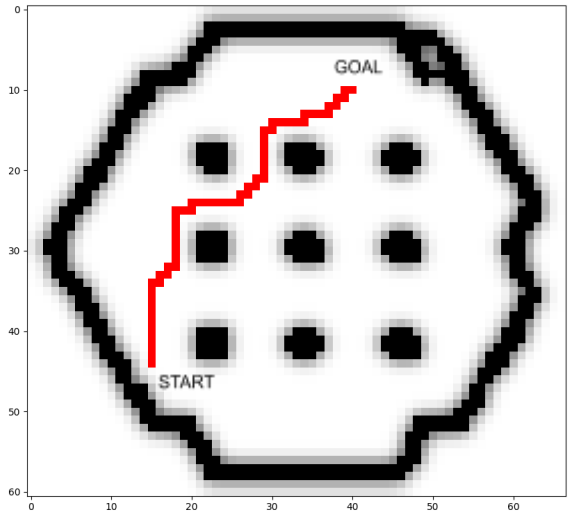

Fig. 2.4: Before Waypoint Rejection

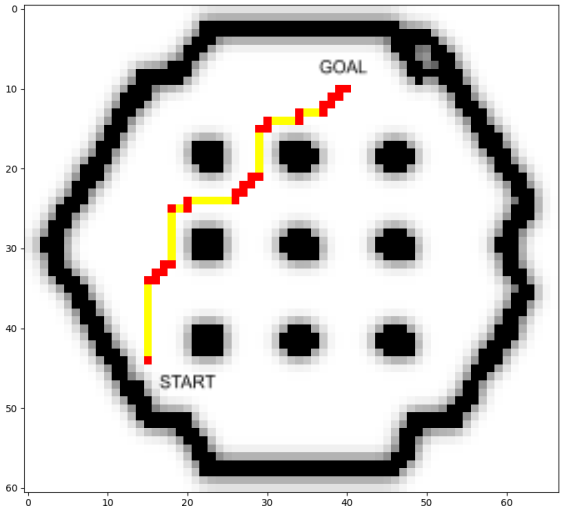

Fig. 2.5: After Waypoint Rejection

2.4 Modified Astar

This is another step targeted at improving the performance of the local planner. The performance of the local planner depends on the waypoints given out by the global planner. The local path might oscillate or move in a jagged fashion if the global waypoints themselves are in such a way. This can be solved by assigning costs to moving left or right in the Astar algorithm. This results in more straight paths and fewer waypoints as seen below.

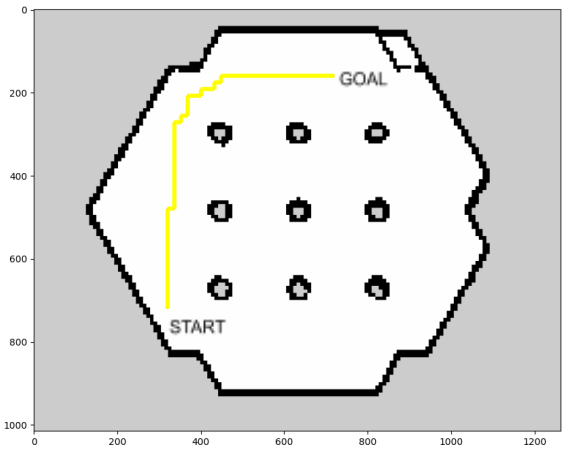

Fig. 2.6: Before Modified Astar

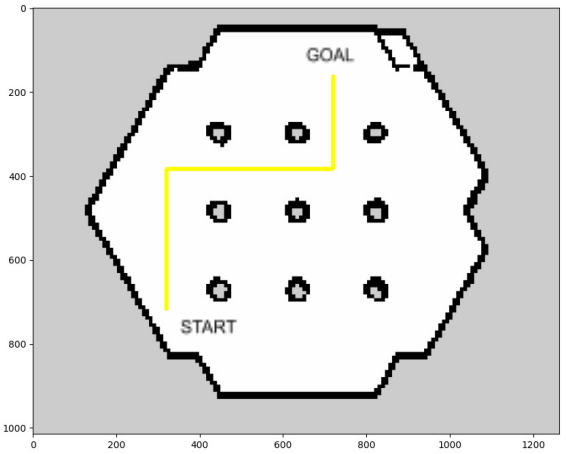

Fig. 2.7: After Modified Astar

Cost assignment for moving to a particular state is done as below:

$g = g_1 + g_2 + g_3 + g_4$

$g_1 = accumulated\ cost$ = parent.g

$g_2 = cost\ of\ moving = 1$

$g_3 = cost\ of\ turning = 10\ \ if\ (left\ or\ right)\ else\ 0$

$g_4 = cost\ of\ inflated\ obstacles = grid[i, j]$

CHAPTER 3 : LOCAL PLANNER

There is a lot of useful information that the sensors on a robot provide about its environment. The region inside which the robot can *see* is called the perceptive field of the robot. The local area surrounding the robot, which is a subset of the perceptive field, is usually used for real-time, lower-level navigation of an unknown, uncertain or dynamically changing environment. This is the job of the local planner.

This section talks about the algorithms involved in local planning. It talks about what Motion Primitives are, how Ego-Graphs are created, how a locally optimal path is found and how dynamic obstacles are handled.

3.1 Motion Primitives

Motion Primitives are Kinematically feasible paths or trajectories which in turn makes it easier for the lower level controllers to track them. This is one of the reasons why this was chosen. These pieces of paths or trajectories can be either analytical or discrete.

For the current version of the work, simple circular arcs of different curvatures have been used as motion primitives. Two are curved rightward, two left ward and one straight making a total of 5 options to choose from.

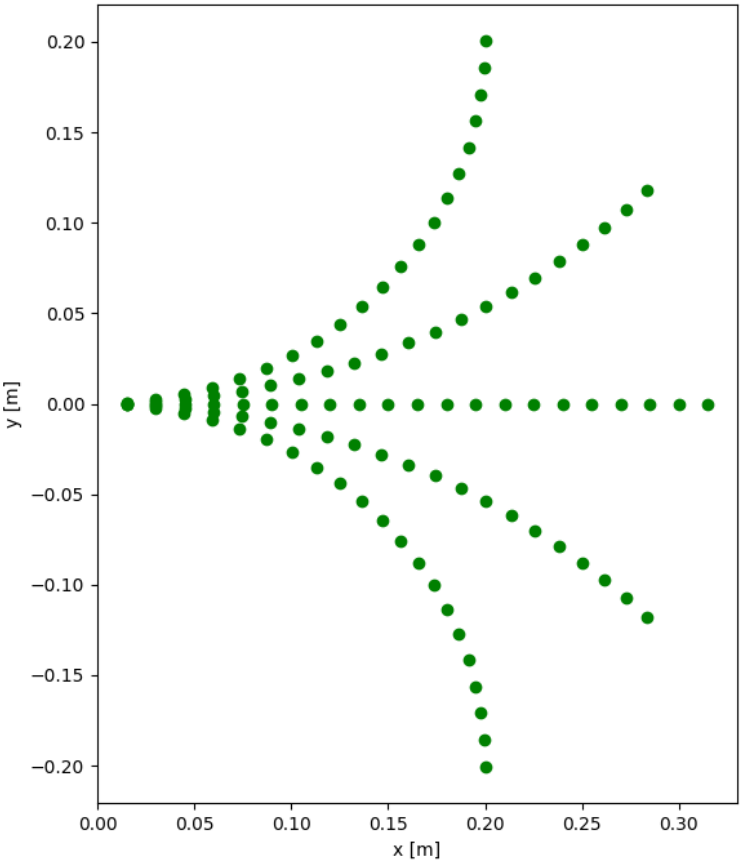

Fig. 3.1: Motion Primitives

3.2 Ego-Graph

The motion primitives from above can be connected one after the other, layer after layer, to create a Kinematically feasible space to work with. This is called the Ego-Graph [1]. These Ego-Graphs can then be used to perform graph search based local planning.

Depending on the number of layers, the velocity of the robot and time dedicated to each motion primitive, we get a certain "coverage" around the robot. In this case, a constant velocity of **0.3 m/s** with **1 sec** for each motion primitive and **5 layers** of the graph has been used, giving a "coverage" of around **1.5 m** in distance and **5 secs** in time around the robot. The images below show the Ego-Graph layer by layer for easier viewing.

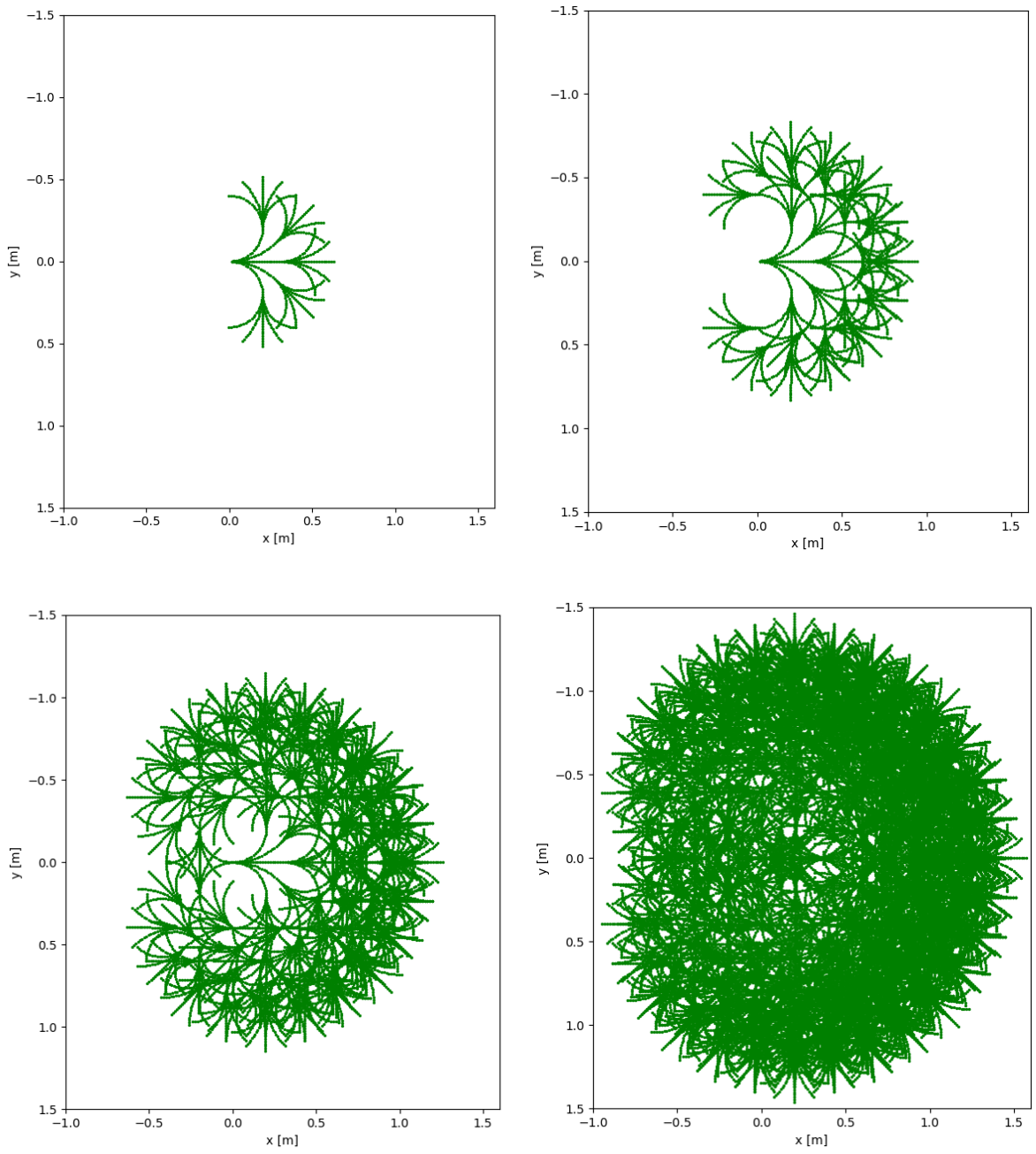

Fig. 3.2: Ego Graph. 2, 3, 4, 5 layers respectively

3.3 Ego-Graph Optimization using Astar

After generating our Ego-Graph we need a way to choose the most optimal path that follows the globally optimal path as much as possible accounting for unforeseeable things like obstacles, errors and uncertainties.

We use the Astar algorithm again to find the most optimal path in this local space. The heuristic (h) used here is the Euclidean distance from the tip of the motion primitive in consideration. The costs (g) used here are the Cross Track Error and the Curvature Error which are explained below.

$$g = g_1 + g_2 + g_3$$

$g_1 = accumulated\ cost$ = parent.g

$g_2 = cost\ of\ deviating = cross\ track\ error$

$g_3 = cost\ of\ turning = curvature\ error$

3.3.1 Cross Track Error

Cross Track Error calculates the amount of deviation of the motion primitive from the global path given by the global planner. Here, since the preferred path is always a straight line, the calculation has been done by rotating the line to align with the x-axis and then summing up the modulus of the y components of the motion primitive.

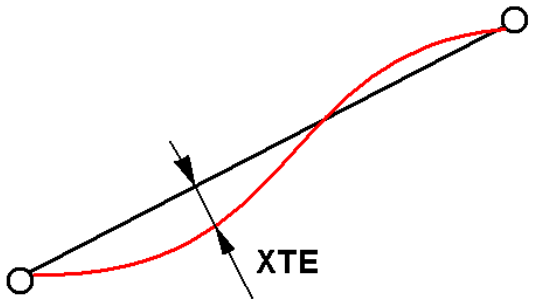

Fig. 3.3: Cross Trajectory Error

3.3.2 Curvature Error

Curvature Error just returns the curvature of the motion primitive in consideration. For a straight path curvature is 0. Hence, the planner would rather go in a straight line and only turn when needed. This reduces oscillations in the final path.

3.3.3 Planning Loop

At each planning instant, we start growing the tree from the current robot pose. The ego-graph in this case is actually a tree. One implementation detail is that, since the motion primitives are fixed, we can just translate and rotate the motion primitives while growing the tree instead of analytically generating them at each pose. Once we find the optimal path, we execute the first motion primitive in the 5 layer path.

One can either wait till the motion primitive is executed or move to the next planning instant faster. This depends on how static/dynamic the environment is. Here a rate of **5 hz** has been used. Once you are close enough to a target waypoint you can mark it as done and move on to the next target. Here that parameter is set to **0.2 m**. If there are other waypoints that are within

reach then they are automatically skipped and we can move on to the next one. We stop once we reach the goal waypoint.

3.4 Dynamic Obstacles Collision Checking

Dynamic obstacles have to be handled while growing the ego-graph itself. This is done by pruning away branches from the tree which could have a possible collision with an obstacle. For this to be implemented we need to be able to predict the motion of moving obstacles. Assuming that based on a certain perception system velocity of obstacles is known, then the motion of these obstacles can be forward propagated to predict a rudimentary straight line motion. The details of this perception system will be discussed later.

Using this information, we do collision checking in time, i.e., we check at each instant of time whether the point location of the motion primitive lies within the polygon of the obstacle at that instant.

3.4.1 No Solution Case

If it so happens that all the branches lead to a collision then we would send a stop signal to the robot. Assuming that the dynamic actors are smart enough to give way to the robot, we wait till there is a feasible solution before proceeding.

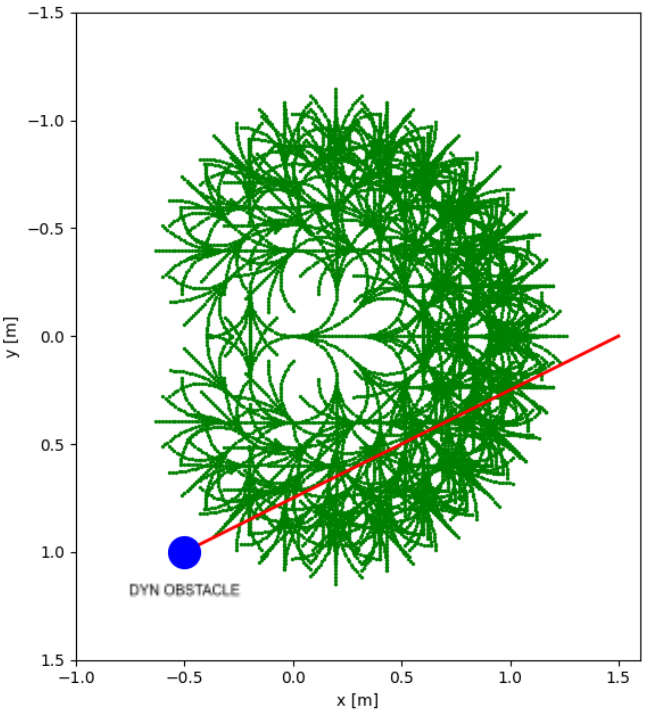

Fig. 3.4: Collision Checking in Time

CHAPTER 4: MAPPING

For implementing the Planner on hardware we would need the availability of a map of the physical location. This would require us to generate a map prior to planning. This can be done through techniques such as Hector SLAM or GMapping.

Here, Hector SLAM has been used since it depends solely on the LIDAR data (from the RPLidar) and not on Odometry. The below section explains the algorithm and the results obtained.

4.1 Hector SLAM

Hector SLAM is a SLAM approach that can be used without odometry as well as on platforms that exhibit roll/pitch motion (of the sensor, the platform or both). It leverages the high update rate of modern LIDAR systems and provides 2D pose estimates at scan rate of the sensors. By using a fast approximation of map gradients and a multi-resolution grid, reliable localization and mapping capabilities, a variety of challenging environments are realized. While the system does not provide explicit loop closing ability, it is sufficiently accurate for many real world scenarios. The system has successfully been used on Unmanned Ground Robots, Unmanned Surface Vehicles, Handheld Mapping Devices and logged data from quadrotor UAVs.

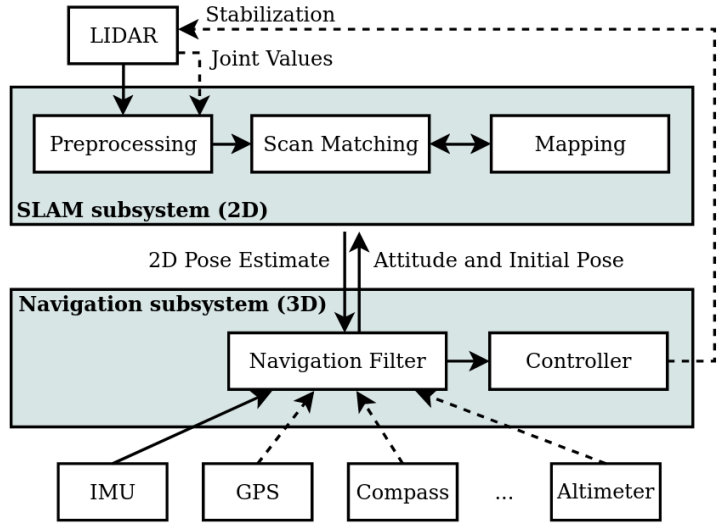

Fig. 4.1: Hector SLAM Flow Diagram

4.2 Maps Produced

Maps were produced by remotely moving the ASL Indoor Mobile Robot around the 3rd floor of the ED Department at a steady pace while running Hector SLAM. The data was collected at night so as to not cause interference due to sunlight for the LIDAR.

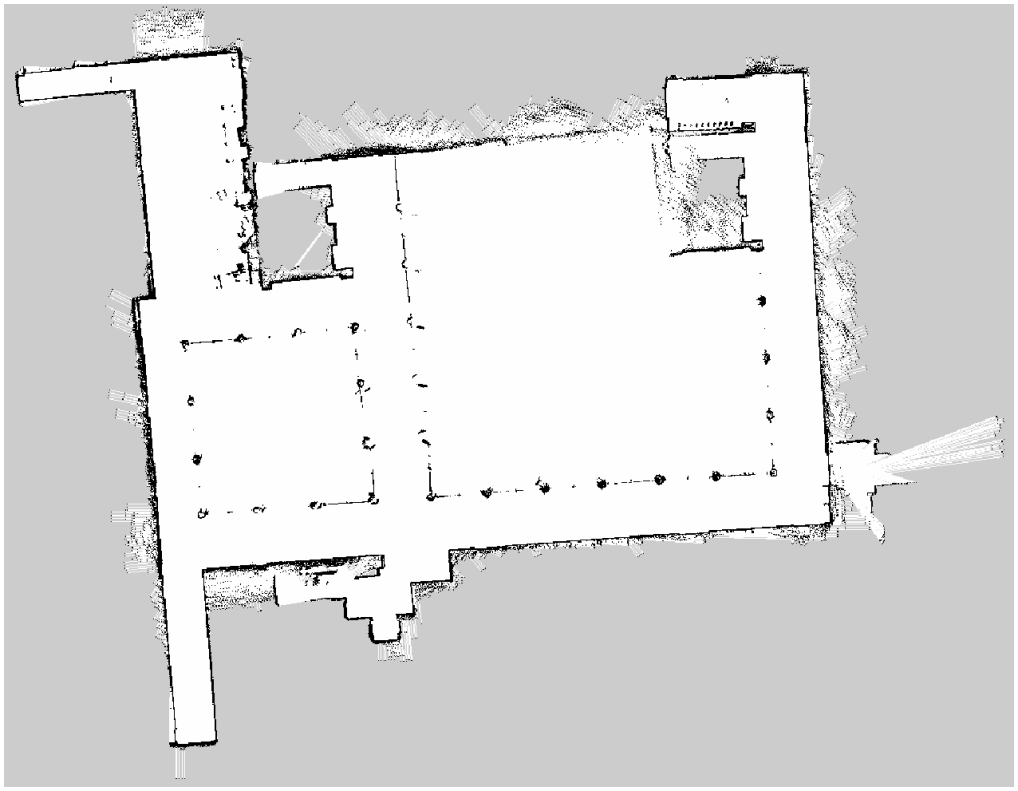

Fig. 4.2: ED 3rd Floor Raw Map File

The 3rd floor contained railings which were surprisingly captured by the lidar and can be seen as faint lines on the map. This creates openings where there shouldn't be one. This is tackled by introducing artificial boundaries manually by editing the PGM file. The new map is as shown below.

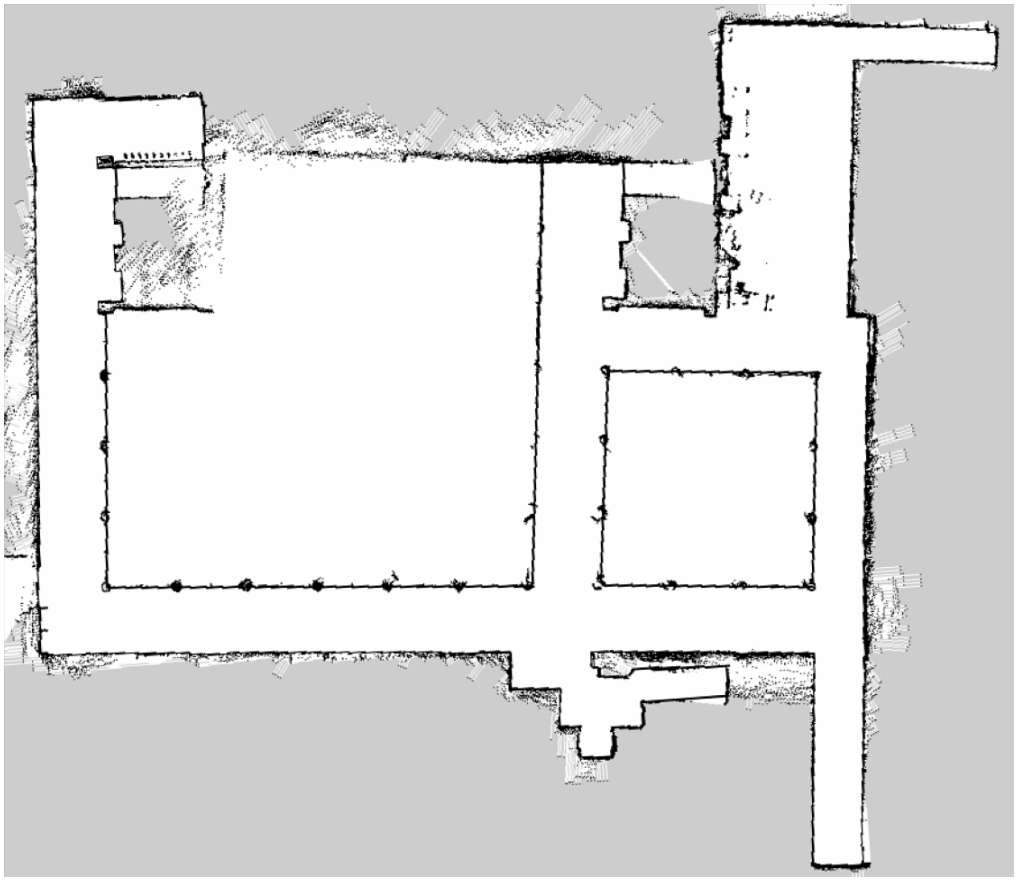

Fig. 4.3: ED 3rd Floor Edited Map File

CHAPTER 5: DETECTING DYNAMIC OBSTACLES

To introduce dynamic obstacles into the motion planning algorithm, we need a mechanism to detect as well as track these obstacles. Detection is needed so that we can run some sort of collision checking. Tracking because we need the velocity vector to predict the motion of obstacles.

obstacle_detector is an open source ROS package that allows us to do the above. It provides utilities to detect and track obstacles from data provided by 2D laser scanners. Detected obstacles come in the form of line segments or circles. Circular obstacles are subject to a tracking algorithm based on the Kalman filter. A by-product of this is the information on tracked obstacles velocity. The package was designed for a robot equipped with two laser scanners therefore it contains several additional utilities.

The format in which the obstacles as published is as below:

```
Header header

obstacle_detector/SegmentObstacle[] segments
obstacle_detector/CircleObstacle[] circles
```

where Circle Obstacles is of the format:

```
geometry_msgs/Point center      # Central point [m]
geometry_msgs/Vector3 velocity  # Linear velocity [m/s]
float64 radius                  # Radius with added margin [m]
float64 true_radius             # True measured radius [m]
```

and Segment Obstacles is of the format:

```
geometry_msgs/Point first_point  # First point of the segment [m]
geometry_msgs/Point last_point   # Last point of the segment [m]
```

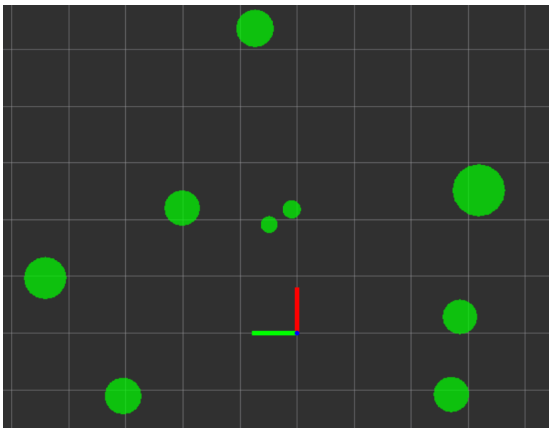

Fig. 5.1: Dynamic Tracked Obstacles

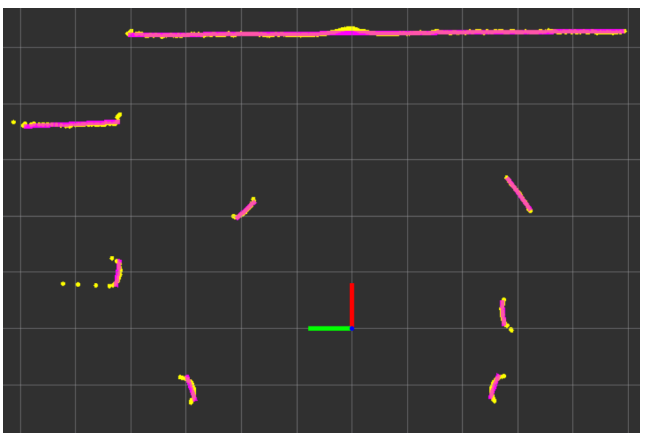

Fig. 5.2: Raw Point Cloud with Processed Segments

CHAPTER 6: IMPLEMENTATION

For the initial static map the YAML and PGM file were generated using the LIDAR sensor with either the *gmapping* or the *hector_slam* node by moving the robot manually around the real or simulated environment.

The code has been implemented using ROS, written in Python and simulated in Gazebo. The below images show the code structure, the different classes and how they interact with each other.

The code for this project can be found in this repository.

Robot Setup

- Velocity: 0.3 m/s
- Time per motion primitive: 1 sec
- Time Step: 0.05 sec
- Motion Primitive Config: [-0.2, -0.4, inf, 0.4, 0.2]
- Planning Frequency: 5 Hz
- Sufficient Distance to Target: 0.2 m
- Simulation Map: Turtlebot3 World
- Simulation Map Resolution per pixel: 0.0069
- Simulation Map Downscaling Factor: 1/16
- Physical Map: ED Department 3rd Floor
- Physical Map Resolution per pixel: 0.05 m
- Physical Map Downscaling Factor: ⅛
- ROS Version: ROS 1
- Python Version: 3.8.10

Simulation Setup

- Robot Model: Turtlebot3 Burger
- Gazebo Version: 11.1
- Dynamic Obstacles: Gazebo Actors [*walking.dae*]

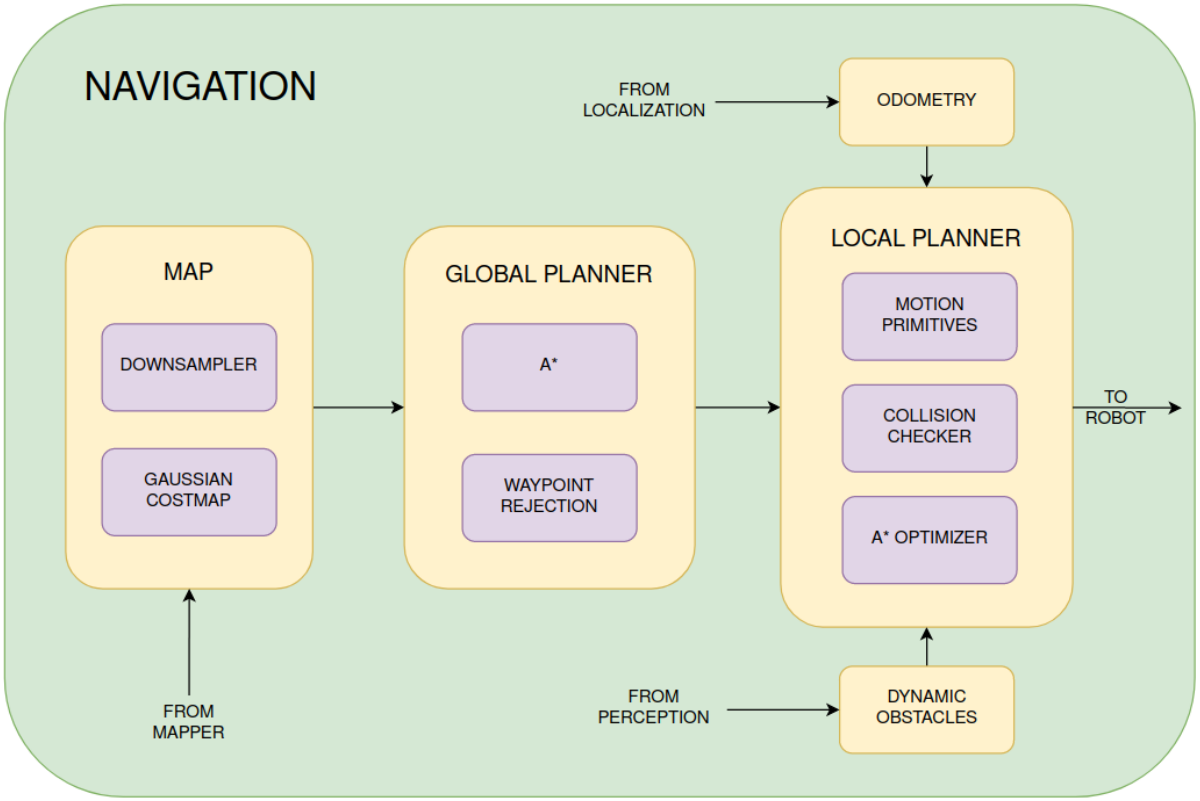

Fig. 6.1: Navigation Module Flow Diagram

```
class GlobalPlanner:
    def __init__(self, map, source, goal): ...
    def neighbors(self, node): ...
    def solve(self): ...
    def euclidean_dist(self, state): ...
    def backtrack(self, node): ...
    def clean(self): ...
    def show_path(self): ...
```

```
class Map:
    def __init__(self, pgm_file="map.png"): ...
    def get_value(self, i, j): ...
    def read_yaml(self): ...
    def downsample(self, factor=1 / 16, cropx
    def show(self): ...
    def get_pos_from_indices(self, i, j): ...
    def get_indices_from_pos(self, x, y): ...
```

```
class LocalPlanner:
    def __init__(self): ...
    def generate_motion_primitives(self): ...
    def show_motion_prims(self): ...
    def show_ego_tree(self): ...
    def dyn_obs_callback(self, obstacles): ...
    def is_inside_circle(self, xo, yo, r, x, y): ...
    def is_in_collision(self, path, t): ...
    def rot_trans_matrix(self, pose): ...
    def neighbors(self, node): ...
    def optimize(self): ...
```

```
    def euclidean_dist(self, path): ...
    def cross_track_error(self, path): ...
    def curvature_error(self, idx): # delta theta
    def cost_map_error(self, path): ...
    def backtrack(self, node): ...
```

```
class Navigation:
    def __init__(
    ): ...
    def generate_global_path(self): ...
    def odometry_callback(self, data): ...
    def start(self): ...
```

Fig. 6.2: Class Structure

CHAPTER 7: RESULTS

7.1 TurtleBot3 World Static

With starting point as **(-1.8m, -1.5m)** and goal point as **(0.7m, 0.2m)**, the below images show the global path output, the actual path taken, a snapshot of the local planner output and gazebo simulator at the same moment. The complete animation can be found here.

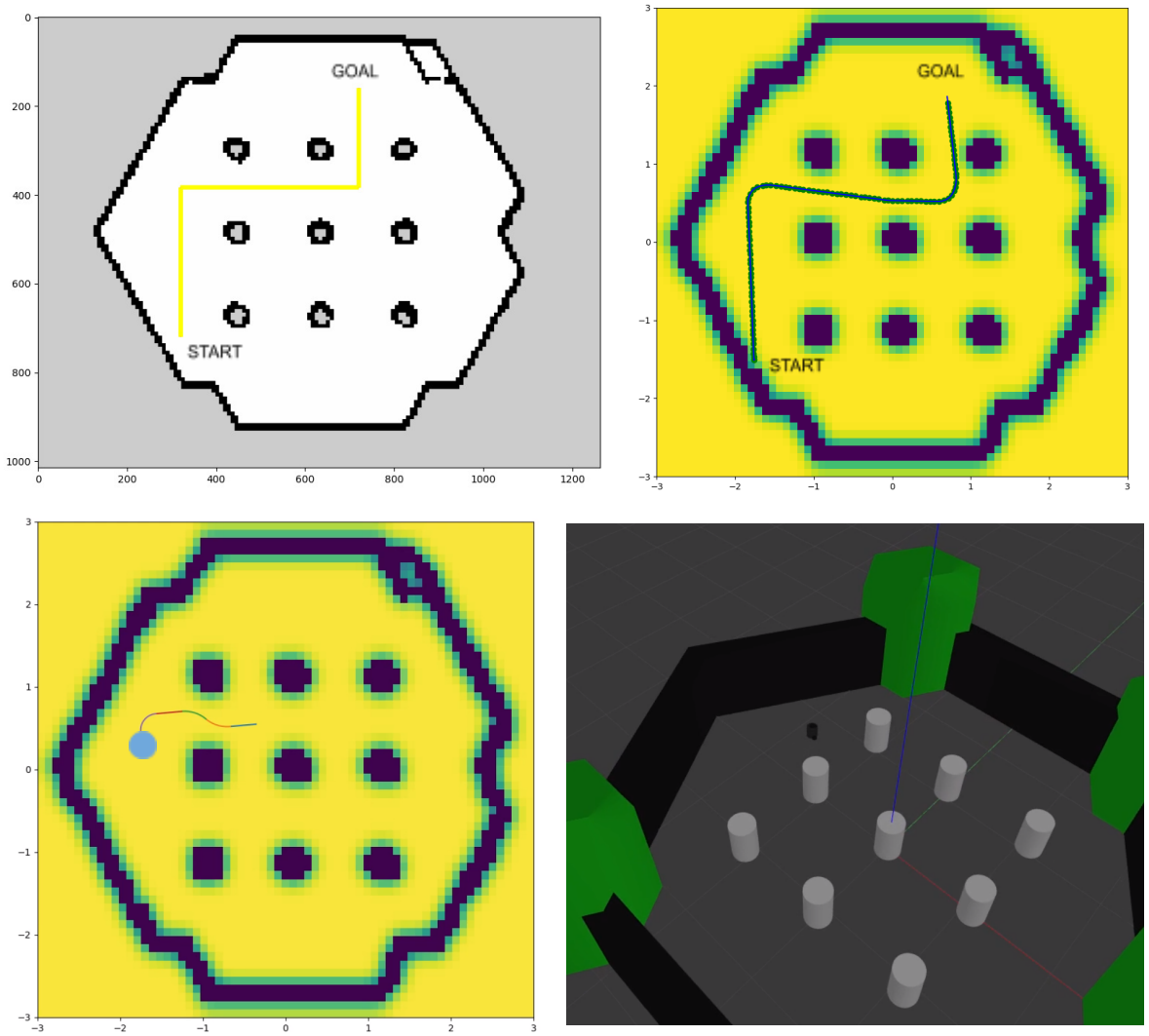

Fig. 7.1: Result 1

7.2 ED 3rd Floor Static

With starting point as **(1.192m, 1.347m)** and goal point as **(33m, 24.3m)**, the below images show the global path output, the actual path taken, a snapshot of the local planner output and gazebo simulator at the same moment. The complete animation can be found here.

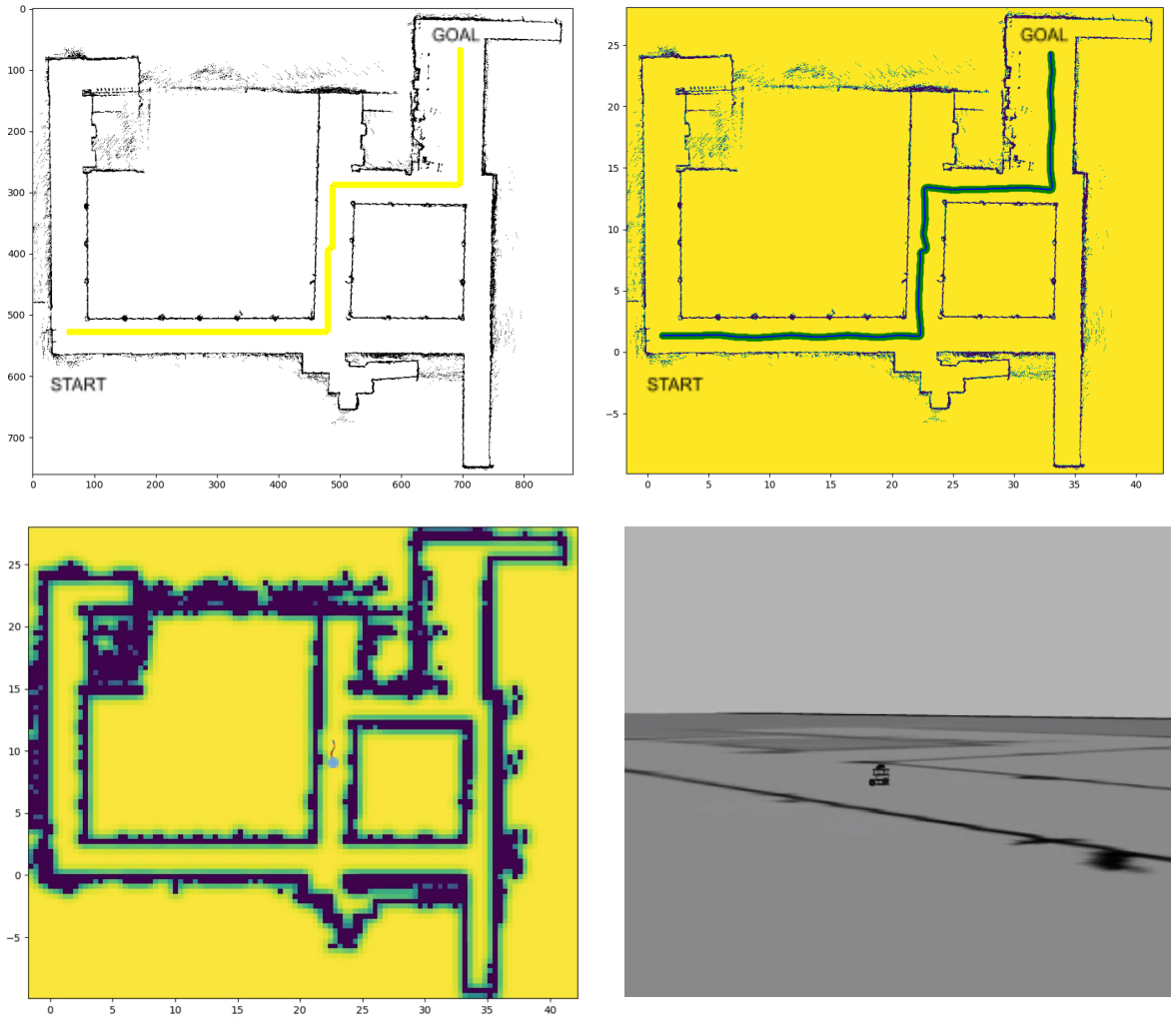

Fig. 7.2: Result 2

7.3 ED 3rd Floor with Dynamic Obstacles

With a similar global path as before but with added dynamic actors that move at the same speed as the robot, the below images show the actual path taken, a zoomed version of the same, a zoomed snapshot of the local planner output and gazebo simulator at the same moment. The complete animation can be found here.

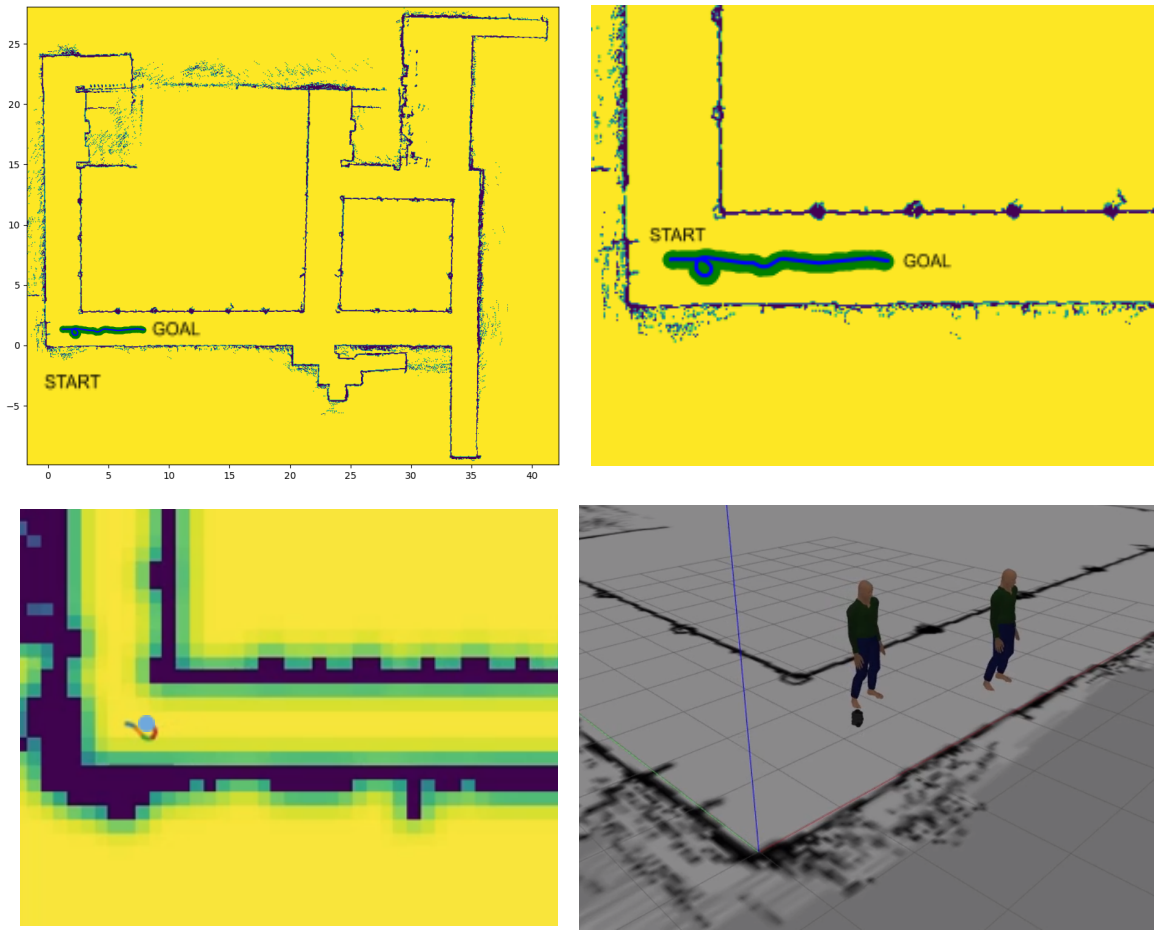

Fig. 7.3: Result 3

CHAPTER 8: FUTURE WORK AND CONCLUSION

8.1 Future Work

Motion Primitives is one thing that can be experimented with a lot since simple circular arcs were chosen here. Motion primitives with different velocities can be added to tackle obstacles and turns better. More complex motion primitives can be added to satisfy various boundary conditions like curvature, acceleration, etc.

Other costs can be experimented with at the local optimization stage. Things like costmap error can be used to keep a safe distance from obstacles.

The way the global waypoints are represented for use in computing cross track error can be improved. Currently only straight line segments for each consecutive pair of waypoints is considered, hence turns are not predicted earlier at the local planning stage.

The planner is as good as the perception data. Improving the perception would go a long way in improving the performance of the planner.

Another thing that can be done is to improve motion prediction for dynamic obstacles. Currently, a straight line is extrapolated using the velocities of the obstacles. Maybe Markov Chains could be useful.

8.2 Conclusion

A working end-to-end Hierarchical Planner has been built and tested. The robot has been seen to follow smooth trajectories based on the motion primitives as well as avoid dynamic obstacles. Experimenting on hardware and other model intensive robots might reveal more of its advantages and flaws. There are still various scenarios and situations to be worked upon. Hope this serves as a good background for others to take things forward.

REFERENCES

1. A. Lacaze, Y. Moscovitz, N. DeClaris and K. Murphy, "Path planning for autonomous vehicles driving over rough terrain," *Proceedings of the 1998 IEEE International Symposium on Intelligent Control (ISIC) held jointly with IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA) Intell*, Gaithersburg, MD, USA, 1998, pp. 50-55, doi: 10.1109/ISIC.1998.713634.
2. Howard, T.M., Green, C.J., Kelly, A. and Ferguson, D. (2008), State space sampling of feasible motions for high-performance mobile robot navigation in complex environments. J. Field Robotics, 25: 325-345. https://doi.org/10.1002/rob.20244
3. Howard TM, Kelly A. Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots. The International Journal of Robotics Research. 2007;26(2):141-166. doi:10.1177/0278364906075328
4. J. Feng, J. Zhang, G. Zhang, S. Xie, Y. Ding and Z. Liu, "UAV Dynamic Path Planning Based on Obstacle Position Prediction in an Unknown Environment," in IEEE Access, vol. 9, pp. 154679-154691, 2021, doi: 10.1109/ACCESS.2021.3128295.
5. M. Kobayashi and N. Motoi, "Local Path Planning: Dynamic Window Approach With Virtual Manipulators Considering Dynamic Obstacles," in IEEE Access, vol. 10, pp. 17018-17029, 2022, doi: 10.1109/ACCESS.2022.3150036.
6. Q. Jin, C. Tang and W. Cai, "Research on Dynamic Path Planning Based on the Fusion Algorithm of Improved Ant Colony Optimization and Rolling Window Method," in IEEE Access, vol. 10, pp. 28322-28332, 2022, doi: 10.1109/ACCESS.2021.3064831.
7. http://wiki.ros.org/hector_slam
8. https://github.com/tysik/obstacle_detector
9. [Cross Track Error Image] http://2.bp.blogspot.com/_1KkyQx4TJko/TQvCLJkRAbI/AAAAAAAAAWc/U2tLrz5oLQg/s1600/XTE.png
10. [Gazebo Actors] https://classic.gazebosim.org/tutorials?tut=actor&cat=build_robot
11. [Gazebo Actor Collisions] https://github.com/gazebosim/gazebo-classic/tree/gazebo11/examples/plugins/actor_collisions
12. [Image as Ground Plane in Gazebo] https://answers.gazebosim.org/question/4761/how-to-build-a-world-with-real-image-as-ground-plane/
13. http://wiki.ros.org/tracking_pid