# EE478 HW3

nijatabbasov06

Apr 2025

## Table of Contents

# 1 Problem 1

Note: This file resides in the Reports subfolder of the zip file. And videos can be found at Videos subfolder

I created a ee478_hw3.launch and christmas_tree.launch files. Those files are similar with mavros_posix_sitl.launch, the only difference being the gazebo world model they are using and the iris model they are using. ee478 launch file is launching the ee478_hw3.world and the christmas launch file is launching the christmas_tree.world file. Both these launch files are using depth camera.


Figure 1: ee478_hw3.launch


Figure 2: christmas_tree.launch

Moreover, I modified the kino_replan.launch file in the following way:

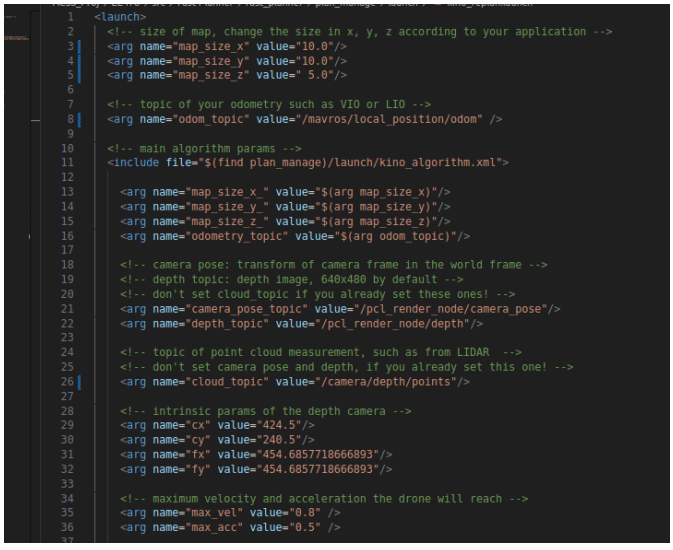



Figure 3: kino_replan.launch

Here, I am setting the odom_topic parameter to /mavros/local_position/odom and the parameter cloud_topic parameter to /camera/depth/points. In the launch file, I am running the following node to be able to publish a transform between camera_link and the camera_depth_optical_frame:

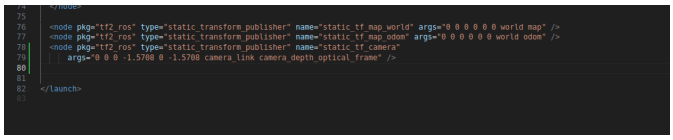



Figure 4: publishing static transform

Then, I changed the code in the sdf_map.cpp file the following way:

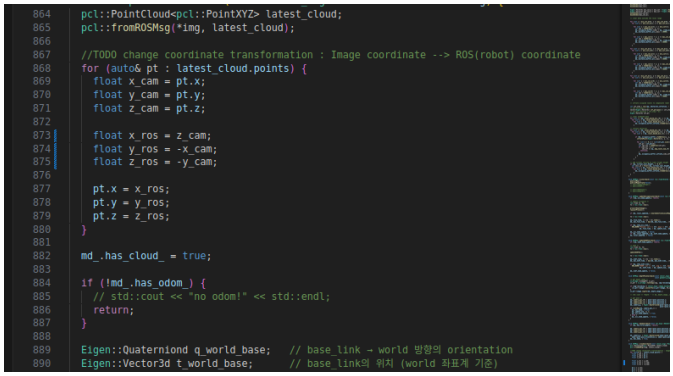



Figure 5: Changes in sdfmap

# 2 Problem 2

I cam subscribing to the following topics to get the drone's pose data and the path data outputted from the fast_planner node.


Figure 6: Subscribed topics

In the offboard velocity file, I am subscribing to the pose topic of the mavros to get my current location and I subscribe to the lookahead_waypoint topic published by the global_planner to get the next target to fly my drone into.

Moreover, I am publishing the velocity data to mavros for my drone to fly with that speed. To determine the published velocity and angular acceleration, P controller is being used.

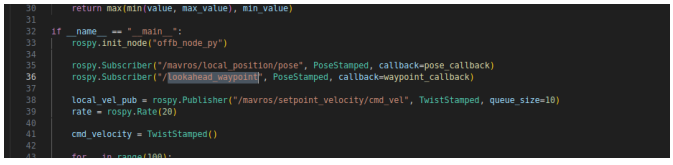



Figure 7: offboard velocity file

# 3 Problem 3

As the FOV of the camera is limited, P controller for Yaw is implemented. I filled the empty spaces the following way:

```
68      cmd_velocity.twist.linear.z = clamp(vz, -MAX_VEL_Z, MAX_VEL_Z)
69
70      # --- yaw control ---
71      current_yaw = get_yaw_from_orientation(current_pose.pose.orientation)
72      target_yaw = get_yaw_from_orientation(target_pose.pose.orientation)
73      error_yaw = yaw_error(target_yaw, current_yaw)
74      wz = K_YAW * error_yaw
75
76      cmd_velocity.twist.angular.z = clamp(wz, -MAX_YAW_RATE, MAX_YAW_RATE)
77
78      local_vel_pub.publish(cmd_velocity)
```

Figure 8: Yaw Control

We are doing the above, because we want our camera to be looking at the position we want to arrive. Because, in that case, our fast planner is able to generate a path.

# 4 Problem 4

## 4.1 Paper Review

There are a lot of common methods for path planning, one of them is GTO - Gradient based Trajectory Optimization. This optimization method is using safety, dynamic feasibility, and smoothness as bunch of factors to determine the most suitable paths. However. this method has some problems, even though GTO enables us to find the solution that minimizes the loss function, it may find a solution that is locally minimal but not globally minimal. One example of the failure of the GTO is the following:

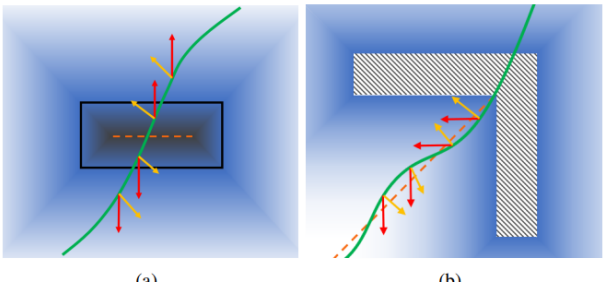


Figure 9: GTO failure

When our initial passes through the vicinity of a valley or ridge, gradients of the ESDF points to oposite directions near the valley or a ridge, for this reason, applying gradient descent does not push the trajectory out of the obstacle, and instead decreases its smoothness level and the trajectory is still inside the obstacle.

For this reasons, different approach is used in the paper. We are using a method named Path Guided trajectory Optimization - PGO for our purpose. Firstly, here we are using B-spline algorithm to represent the trajectories. One property of B-spline is that we can represent pathes by set of vertexes and changing one vertex does not affect the other vertexes. PGO method is similar with GTO however, while GTO was only using gradient of ESDF (Euclidean Signed Distance Field) to smooth out the given initial trajectory and push it out of the obstacle, this method first

generates topological paths that correspond to different homologies initially and after that warmup trajectory is generated by utilizing additional information. The mentioned additional information is the output of the classical methods such as A* or RRT, or PRM. As we know that the results of these method never goes inside the obstacle, we use this information to push the topological path out of the obstacle. We create a loss function as the following and by using that, we solve the tradeoff issue. To elaborate, A* algorithm may output a trajectory that is very close to the obstacle and a trajectory that is not dynamically feasible as in not smooth. For this reason, we have to decide on a tradeoff between these properties:

$$f_{p1} = \lambda_{1s} f_s + \lambda_{1g} f_g$$

Figure 10: Loss function

In the above function, fp is the loss function, $\lambda_{1s}$ and $\lambda_{1g}$ are the parameters and the fs and fg are the properies explaining the smoothness and how far our new trajectory is from the initial trajectory, for example A* trajectory. These are the equations describing them:

$$f_s = \sum_{i=p_b-1}^{N-p_b+1} \|\mathbf{Q}_{i+1} - 2\mathbf{Q}_i + \mathbf{Q}_{i-1}\|^2$$

Figure 11: Fs the smoothness level

where $Q\_i$ are the vertexes and pb is the polynomial degree of a B-spline between two vertexes. The above equation can be thought as every vertex is an object and all those objects are connected with a spring to two objects at the adjacent vertices. This way, our fs is the sum of the magnitude of the forces felt by each vertex between pb - 1 to N - pb + 1.

Then, we have fg:

$$f_g = \sum_{i=p_b}^{N-p_b} \|\mathbf{Q}_i - \mathbf{G}_i\|^2$$

Figure 12: f_g definition

Here $G_i$ are the reference vertexes as in vertexes oututted from A*. That is why we are summing the distances between related verteces to represent how far we are from the reference trajectory.

After solving this loss function, we get a trajectory named warmup trajectory. After this, we are applying the GTO method to our trajectory to get a smooth trajectory which is also far from the obstacles. To do that, we use the B-spline optimization method to minimize the following loss function:

$$f_{p2} = \lambda_{2s} f_s + \lambda_{2c} f_c + \lambda_{2d} \left( f_v + f_a \right)$$

Figure 13: Loss function by B-spline optimization

here, fc is the collision cost calculated by using the ESDF and fa and fv are the penalties that explain the dynamic infeasibility of velocity and acceleration values.

### 4.1.1 Finding topological path

There are well known methods such as A* to find the initial trajectory to guide the PGO method, however, that would result in us only finding a locally optimal solution, for this reason, we guide the PGO with multiple topological pathes. However, we need to specify what classes should those pathes be seperated by, as otherwise, we would end up with infinite possible pathes not knowing what to do. Even though, concept of homotopy is used widely for classifying pathes, it results in insufficient number of pathes. Another method is VD - visibility deformation. However, this method is computationally extensive, for this reason, this paper defines a method called UVD - uniform visibility deformation to classify topological paths. For checking the equivalence between two pathes the following is used for both VD and UVD:

**Definition 1.** Two trajectories $\tau_1(s)$, $\tau_2(s)$ parameterized by $s \in [0, 1]$ and satisfying $\tau_1(0) = \tau_2(0), \tau_1(1) = \tau_2(1)$, belong to the same *uniform visibility deformation* class, if for all $s$, line $\overline{\tau_1(s)\tau_2(s)}$ is collision-free.

Figure 14: Equivalence Checking

However, the difference between UVD and VD is that in UVD we are only checking the line between points on both trajectories which has the same parameter s. This, results in less computation time for our method. The following picture can be analyzed for deeper understanding:

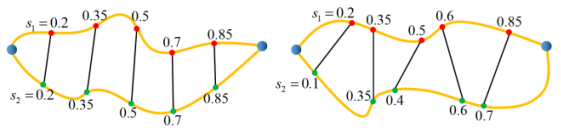



Fig. 6. Comparison between UVD (left) and VD (right). Each red point on one path is transformed to a point (green) on the other path. Any two associated points correspond to the same parameter $s$ in UVD, but not in VD.

e

Figure 15: Equivalence Checking difference

To find those distinct paths, the following algorithm is used:

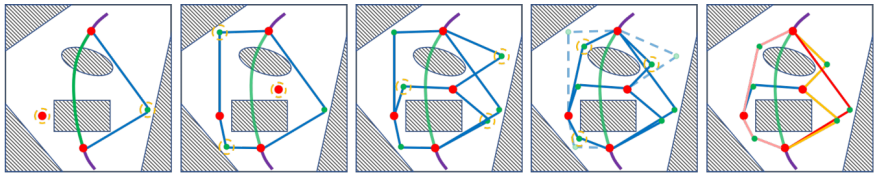
**Algorithm 1:** Topological Roadmap

**Initialize**()
**AddGuard**$(\mathcal{G}, s)$, **AddGuard**$(\mathcal{G}, g)$
**while** $t \leq t_{max} \wedge N_{sample} \leq N_{max}$ **do**
  $p_s \leftarrow$ **Sample**()
  $g_{vis} \leftarrow$ **VisibleGuards**$(\mathcal{G}, p_s)$
  **if** $g_{vis}$.**size**() $== 0$ **then**
    **AddGuard**$(\mathcal{G}, p_s)$
  **if** $g_{vis}$.**size**() $== 2$ **then**
    $path_1 \leftarrow$ **Path**$(g_{vis}[0], p_s, g_{vis}[1])$
    $distinct \leftarrow True$
    $\mathcal{N}_s \leftarrow$ **SharedNeighbors**$(\mathcal{G}, g_{vis}[0], g_{vis}[1])$
    **for each** $n_s \in \mathcal{N}_s$ **do**
      $path_2 \leftarrow$ **Path**$(g_{vis}[0], n_s, g_{vis}[1])$
      **if Equivalent**$(path_1, path_2)$ **then**
        $distinct \leftarrow False$
        **if Len**$(path_1) <$ **Len**$(path_2)$ **then**
          **Replace**$(\mathcal{G}, p_s, n_s)$
        break
    **if** $distinct$ **then**
      **AddConnector**$(\mathcal{G}, p_s, g_{vis}[0], g_{vis}[1])$

Figure 16: Path finding

Here we are introducing two graph nodes named guard and connector similar to the Visibility-RPM. Guard are responsible for exploring and finding distinct pathes. Moreover, we are using DFS - Depth First Search algorithm to find the available paths and use visited nodes list to do that. An example to DFS may be backtracking.

The following is the picture providing more information on how this method is being utilized:



(a) (b) (c) (d) (e)

Fig. 7. Generation of the topological roadmap. Red and green nodes represent the *guards* and *connectors* respectively. (a)-(c): *Guards* are added to occupy different regions of space, and *connectors* are added to form new connections between the *guards*. (d): new *connectors* replace the old ones, making the connections shorter. (e): some of the paths found by the depth-first search. Both the red and orange paths belong to the same UVD class, while the pink path is the only member of its UVD class.

Figure 17: DFS

## 4.2 Code Review

### 4.2.1 launch

In the launch folder of the repo, we see the followings:

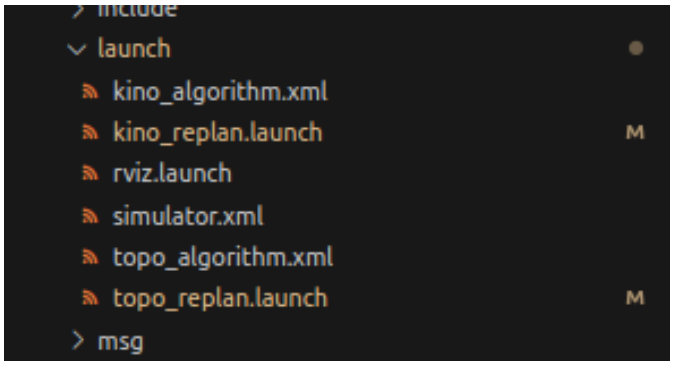



Figure 18: Launch

In the fast planner folder there are two methods for path searching one is kinodynamic[2]. [3], and the other is topological path searching. The launch files with the topo and kino name inside, means running the node with the ros parameters necessary for topological and kinodynamic path searching, respectively. Same rule applies to the .xml files as in having a name topo means they are responsible to setup the topological path searching properties and the same for kinodynamic path searching. Those xml files are being called by the corresponding launch files and they are the files that initiate the nodes.

Moreover, rviz.launch launches the rviz node with the saved rviz file to not start the graphs again. And the simulaton.xml loads the map that is created by the HKUST researchers. This is the reason we are not using simulation.xml as we have our own tree world.

### 4.2.2 src

In the src folder of the repo, there are multiple packages:

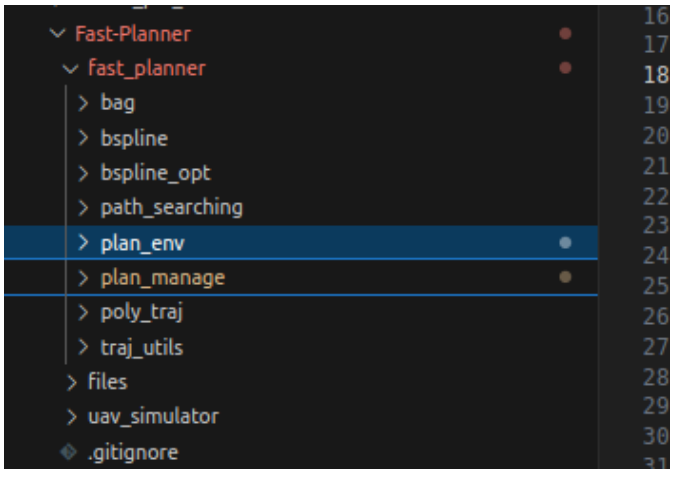



Figure 19: packages

The most important to us is the plan_manage package as that is being directly called from the launch files. The fast_planner_node initializes the node and then calls kino_replan_fsm oir topo_replan_fsm depending on which roslaunch file we called. Inside those cpp files planner_manager.cpp is called. Planner_manager initiates the planner, and activates the B-spline optimization classes

to start planning the paths. This file also calles the nodes inside the env_plan package to set up the environments, frames, transforms. Moreover, we can see inside the plan_env package that Astar algorithms are being used to determine initial pathes. The codes inside the path_searching are responsible for searching the initial paths, and the nodes in the bspline and bspline_opt is responsible for applying ESDF and using B-spline optimization using its convex hull property. The kino_replan_fsm file itself is subscribing to the odometry data and all the other topics that publish the data needed to run these nodes.

traj_server is responsible for publishing the necessary data to the topics for the other ros topics to utilize. For example, we need to subscribe to the topic that provides us with the waypoints.

Some advantages of the past planner is that it is working well in real time, even though other methods may fail to do so. Moreover, it is able to avoid the local minima issues by using topological path searching and is able to do it without causing time complexity issue.

However, some of the disadvantages is that this method fails to generate a trajectory when there is not sufficient odometry data. Moreover, when generating the trajectories, to account for smoothness, it goes with a difficult trajectory instead of going forward. This causes issues as it hits the ground from time to time because smooth trajectory are sometimes trajectories that go through the ground. Also, sometimes, the code ignores the trajectories that are generated in the first loop. Sometimes these are the correct solutions, but the program ignores those solutions. This may be bad in environments where there are no obstacles. Also, it would be much better if the source code would create a map of the environment, as in that case there would be no need for the drone to rotate to take a measurement again. Also, the drone maybe unaware of the obstacles in the back, and right and left, so this would result in suboptimat paths. Even though this would not be a good thing to do in dynamic environment, as our environment is static, I think we may do so.

### 4.3 My code

My code for the fast planner is used in the exploration_planner.py. It has some issues with the topics, but the logic should be correct.

These code subscribes to the cloud topic and as it gets the point data, it pushes those data to the tree_points set. It is basically same with creating a map of the environment, as now, we can track the point data representing the tree. After this, detect obstacle function is called if the detected point amount is bigger than ten. This method sums all the coordinates of the points and averages them to find the center of the object. Moreover, finding the maximum distance between the two points, max_radius is defined, and we are trying to avoid going inside that cube. Also, to avoid time complexity, points on the ground are ignored, we do this by creating a small threshold as in if the point cloud point is below the threshold, we are not adding it to our tree_points set. Moreover, we represent the space as 3d grid, where grid resoluition is 0.25. This way, if our grid is not in the range that represents grids for the tree, we are rotating around the tree.

## 5 Assignment 3

For this assignment, I created the following global_planner list:

```python
class GlobalPlanner:
    def __init__(self):
        rospy.init_node("global_planner")

        # Parameters
        self.lookahead_distance = rospy.get_param("~lookahead_distance", 1)

        self.global_waypoints = [
            Point(5, -3, 1),
            Point(12, 0, 1),
            Point(7, 8, 4),
            Point(0, 0, 1),
            Point(5, -3, 1),
            Point(12, 0, 3),
            Point(4, 4, 3.4),
            Point(0, 0, 4),
            Point(4, -4, 9),
            Point(12, 0, 3),
            Point(4, 4, 3.4),
            Point(0, 0, 4),
            Point(12, 5, 4.5),
            Point(0, 5, 4.75),
            Point(0, 0, 5),
            Point(0, -5, 5.25),
            Point(12, -5, 5.5),
            Point(12, 5, 5.75),
            Point(0, 5, 6),
            Point(0, 0, 6),
            Point(0, -5, 6),
            Point(12, -5, 6),
            Point(12, 5, 6),
            Point(0, 5, 5.75),
            Point(0, 0, 5.5),
            Point(0, -5, 5.25),
            Point(12, -5, 5),
            Point(12, 5, 4.75),
            Point(0, 5, 4.5),
            Point(0, 0, 4.25),
            Point(0, -5, 4),
            Point(12, -5, 3.75),
            Point(12, 5, 3.5),
            Point(0, 5, 3.25),
            Point(0, 0, 3),

        ]
        self.global_index = 0

        # Current state
        self.cur_position = None
        self.path = None
```

Figure 20: global planner

Sometimes, the drone stops in the given global position and the fast planner fails to generate a path. In that case turning of the scripts and running it again may help to overcome the issue.

```python
                return

        rospy.logwarn("No lookahead point found")
        # self.speed = TwistStamped()
        # self.speed.header.stamp = rospy.Time.now()
        # self.speed.header.frame_id = "odom"
        # self.speed.twist.linear.x = (self.global_waypoints[self.global_index].x - self.cur_position.x)/10
        # self.speed.twist.linear.y = (self.global_waypoints[self.global_index].y - self.cur_position.y)/10
        # self.speed.twist.linear.z = (self.global_waypoints[self.global_index].z - self.cur_position.z)/10
        # self.speed.twist.angular.x = 0.0
        # self.speed.twist.angular.y = 0.0
        # self.speed.twist.angular.z = 0.0
        # self.local_vel_pub.publish(self.speed)


    def check_and_publish_global(self):
        if self.cur_position is None or self.global_index >= len(self.global_waypoints):
            return
```

Figure 21: waypoint

The above commented code is also used to overcome the issue. This code publishes the global setpoint manually to the lookahead waypoint to get the fast planner out of the issue and make it compute the trajectory again.

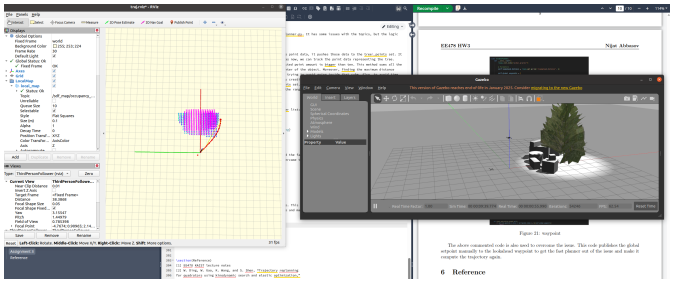


Figure 22: environment

# 6 Reference

[1] EE478 KAIST lecture notes [2] W. Ding, W. Gao, K. Wang, and S. Shen, "Trajectory replanning for quadrotors using kinodynamic search and elastic optimization," in 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018, pp. 7595–7602. [3] S. Liu, N. Atanasov, K. Mohta, and V. Kumar, "Search-based motion planning for quadrotors using linear quadratic minimum time control," in Proc. of the IEEE/RSJ Intl. Conf. on Intell. Robots and Syst.(IROS), Sept 2017, pp. 2872–2879.