Autonomous Systems Challenge

Sarper Gürbüz, Ali Karakullukcu, Valentin Merle, Mateus Salomão, Glenn Tungka

March 4, 2025

Contents

1 Introduction

The aim of this project was to develop an autonomous pipeline for cave exploration using a drone within the Unity simulation environment. The developed system achieves two main objectives: firstly, the drone precisely detects and locates four lanterns inside the cave, and secondly, it maps the traversed environment by generating an occupancy grid map of its surroundings.

The source code of the project can be found in: https://github.com/valemer/AS_challenge

2 ROS Workspace Structure

Figure 1 illustrates the overall structure of the developed ROS workspace. The self-developed nodes are green and explained in more detail in the sections 3 and 4. The following two packages are mainly self-developed: `perception`, `planning`. The packages `control` and `mapping` contain third-party code. For the mapping, we use OctoMap. OctoMap is a probabilistic 3D mapping framework based on OcTrees [HWB+13]. The trajectory planner is adjusted to our needs but based on the work of Richer et al. [RBR16]. To control the drone, we use a control node similar to the work of Lee et al. [LLM10], and the trajectory publisher was a given node in the course `Autonomous Systems`.

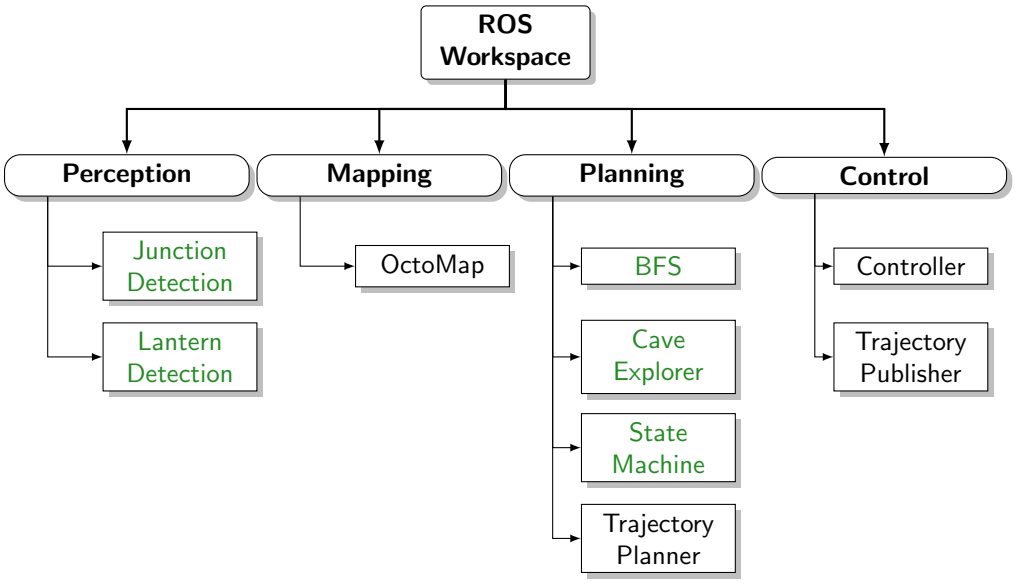

Figure 1: Main parts of the ROS Project and Corresponding ROS Nodes

Additionally, the workspace contains or uses the following packages:

- Perception:
  - depth_image_proc: Contains nodelets for processing depth images and convert them to 3D point clouds.
- Control:
  - mav_trajectory_generation: This package contains the implementation of the trajectory generation library developed by ETH Zürich [BOAS15, RBR16] used in the `Trajectory Planner`.
  - nlopt: This package contains the implementation of the nonlinear optimization library, which is used in the trajectory generation library to optimize the trajectory parameters.
- Simulation: This package contains the simulation provided by the course `Autonomous Systems` including the environment and the drone configuration to solve the task of the challenge.
- additional helper packages: These packages contain additional helper functions, checks, and build routines.

Module	Node	Contribution
Perception	Junction Detection	Merle , Karakllukcu
	Lantern Detection	Merle, Gürbüz
Mapping	OctoMap	Merle
Planning	BFS	Tungka
	RRT Star	Salomão
	Cave Explorer	Merle, Karakullukcu
	State Machine	Merle, Gürbüz, Tungka
	Trajectory Planner	Merle

Table 1: Contribution overview grouped by nodes

As the ROS graph shown in section 2.4 is quite big, we decided to provide table 1 to show the work distribution.

Obs: The RRT star based path planning was implemented and functional but was substituted by the BFS planner for efficiency reasons.

2.1 Perception

Lantern Detection

Under the Perception Package, The **lantern_detection** node is responsible for detecting lanterns using image segmentation and point cloud information. It synchronizes segmented images from a semantic camera with depth information from a stereo camera's point cloud to identify and localize lanterns.

- **Subscribed Topics:**
  - `/unity_ros/Quadrotor/Sensors/SemanticCamera/image_raw`: Color images for yellow lantern detection.
  - `/camera/depth/points`: Point cloud data for 3D localization of detected lanterns.
- **Published Topics:**
  - `/lantern_detection_world_coordinates`: Publishes the world coordinates of the detected lantern.
  - `/lantern_detection_marker_array`: Publishes markers for RViz visualization of detected lanterns (red balls in the provided rviz configuration).
  - `/all_detected_lantern_locations`: Publishes the poses of all detected lanterns as a PoseArray.

Junction Detection

The **junction_detection** node is responsible for detecting junctions (openings or branching points) in the cave using data from an OctoMap 3D occupancy grid. It processes 3D point clouds to find frontier points (free space adjacent to unknown space), clusters them to identify potential junctions, and publishes the centroids of these clusters as potential navigation goals.

- **Subscribed Topics:**
  - `/octomap_full`: Receives the full 3D occupancy grid to detect frontiers and junctions.
  - `/current_state_est`: Receives the drone's current position for setting the cave entry point.
  - `/control_planner`: Controls the activation of junction detection based on the planner's state.
- **Published Topics:**
  - `/clustered_junction_points`: Publishes the clustered points representing potential junctions.
  - `/cluster_centroids`: Publishes the centroids of the clusters as potential junctions.
  - `/goal_point`: Publishes the most recent junction centroid as a navigation goal.

2.2 Planning

State Machine

The **state_machine** node is responsible for managing the high-level decisions of the drone, controlling its autonomous mission in the whole environment through a sequence of states: `TAKE_OFF`, `FLY_TO_CAVE`, `EXPLORE`, `FLY_BACK`, and `LAND`. The state machine navigates the drone from takeoff to exploring the cave for lanterns, then it returns to the starting point after detecting the required number of lanterns and landing.

- **Subscribed Topics:**
  - `/current_state_est`: Receives the drone's current position and orientation for navigation and state transitions.
  - `/all_detected_lantern_locations`: Monitors the number of detected lanterns to decide when to transition from `EXPLORE` to `FLY_BACK`.
- **Published Topics:**
  - `/max_speed`: Sets the maximum speed of the drone depending on the current state.
  - `/global_path`: Publishes the global path for the drone to follow.
  - `/control_planner`: Activates or deactivates the planner for the `EXPLORE` and `FLY_BACK` states.
  - `/fly_back_start_points` and `/fly_back_goal_points`: Define the start and goal points for the return journey during `FLY_BACK`.
- **Service Used:**
  - `/octomap_server/reset`: Used to reset the OctoMap after entering the cave.

Cave Explorer

The **cave_explorer_node** is responsible for the autonomous exploration of the drone through the unknown cave environment by sampling points in front of the drone and constructing a global path to the goal. It samples points on a directed sphere, evaluates them based on their distance from obstacles, the start, and the goal, and reconstructs a path recursively.

- **Subscribed Topics:**
  - `/current_state_est`: Receives the drone's current position and orientation.
  - `/octomap_point_cloud_centers`: Receives point cloud data to avoid obstacles and walls.
  - `/control_planner`: Controls the activation and deactivation of the planner.
  - `/goal_point`: Receives the global goal point from other nodes.
- **Published Topics:**
  - `/global_path`: Publishes the global path for the drone to follow.

Trajectory Planner

The **trajectory_planner_node** generates smooth and dynamically feasible trajectories for the drone by optimizing polynomial paths. It converts high-level global paths into trajectories, ensuring smooth motion by considering maximum speed and acceleration constraints. The node supports trajectory planning for inside and outside the cave. It optimizes up to the 4th order derivative (SNAP), facilitating a smooth flight.

- **Subscribers:**
  - `/max_speed`: Receives dynamic maximum speed adjustments.
  - `/global_path`: Gets the global path for trajectory planning.
  - `/current_state_est`: Subscribes to the drone's current position and velocity.

- **Publishers:**
  - `trajectory`: Sends the optimized trajectory.

This node is used during exploration to navigate towards goal points and during fly-back to return to the starting location.

BFS Node

The **bfs_node** is responsible for path planning while flying back to the entrance using the Breadth-First Search (BFS) algorithm to navigate the drone from a start point to a goal point by exploring a graph representation of the environment. It builds a graph dynamically as the drone explores the environment. The BFS algorithm then finds the shortest path from the start node to the node closest to the goal. The node publishes the planned path for the drone to follow. This node is mainly used when flying back to the cave entrance when all lanterns are found.

- **Subscribed Topics:**
  - `/current_state_est`: Receives the drone's current position and orientation.
  - `/fly_back_start_points`: Receives the starting point for BFS path planning.
  - `/fly_back_goal_points`: Receives the goal point for BFS path planning.
- **Published Topics:**
  - `/global_path`: Publishes the planned global path for the drone to follow.

2.3 Control

The `control` package contains the **controller_node**. The controller node is a geometric controller, designed to follow a desired trajectory and maintain the orientation of the drone. It computes position, velocity, orientation, and angular velocity errors to generate the required forces and torques for flight control.

- **Subscribed Topics:**
  - `command/trajectory`: Desired trajectory to be followed by the drone
  - `/currrent_state_est`: Estimated current position, orientation and angular velocity of the drone
- **Published Topics:**
  - `/rotor_speed_cmds`: Angular velocities of each of the drone´s propellers

2.4 RQT Graph and Relations

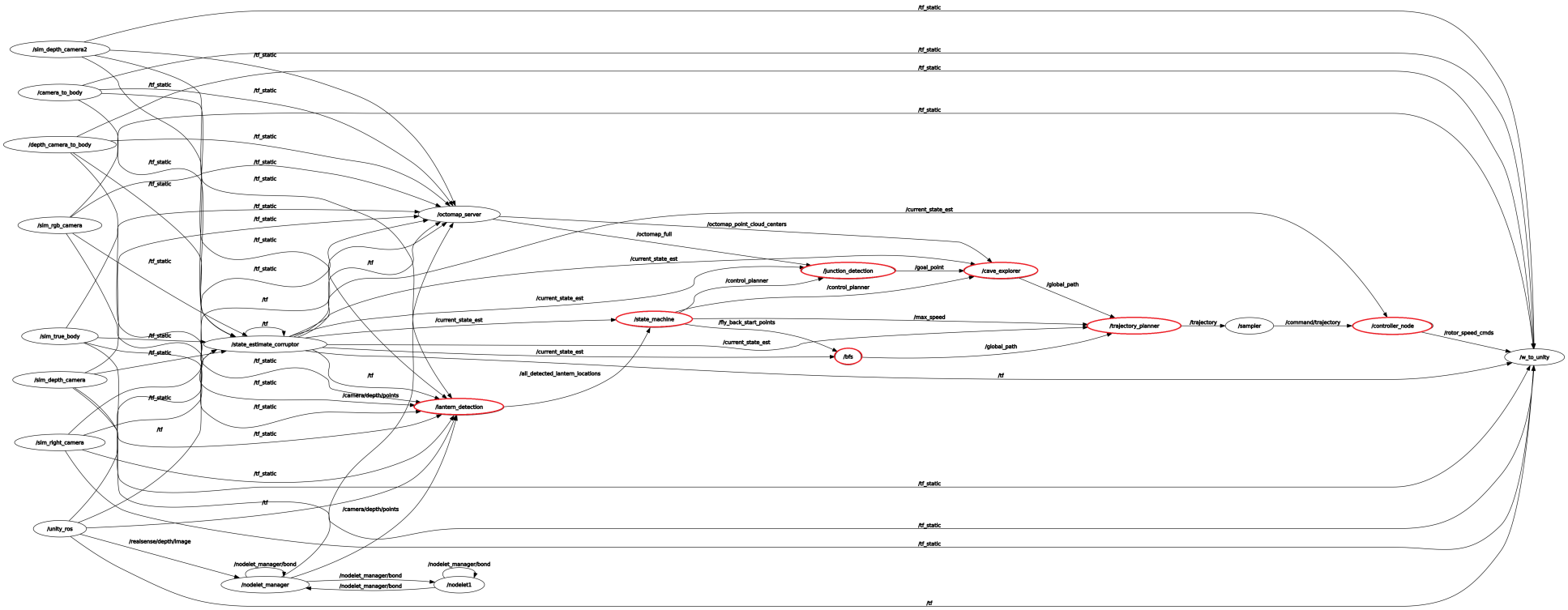

Figure 2: RQT graph which summarizes the relations of the nodes.

3 Detailed Explanation of the Perception Nodes

3.1 Lantern Detection Node

The **lantern_detection** node processes RGB images from the semantic camera and point cloud data from the depth camera to detect yellow lanterns and estimate their 3D world positions. For accurate matching between image and depth data, messages are synchronized using `sync_` function, ensuring both data types are from the same timestamp.

3.1.1 How the Node Works:

The callback function is the core of this node, while functions such as computeCentroid() and updateTrackedObjects() are the secondary helper functions.

Step 1: Converting the Data

Firstly, the ROS image message is converted into an OpenCV format for easier processing. Moreover, ROS depth data is converted into PCL format, which allows processing of 3D coordinates.

Step 2: HSV Format and Finding Yellow Regions in the Image

Secondly, the image is converted to HSV format because it is more practical to do color detection with HSV rather than RGB. After that, a binary mask is created where yellow regions appear white (255) and all other regions appear black (0). The code below explains this procedure:

```
cv::Mat hsv_image;
cvtColor(cv_image->image, hsv_image, cv::COLOR_BGR2HSV);
cv::Scalar lower_yellow(20, 100, 100), upper_yellow(30, 255, 255);
cv::Mat mask;
inRange(hsv_image, lower_yellow, upper_yellow, mask);
```

From this code, we can also see how the upper and lower bounds for the yellow color are set.

Step 3: Finding the Biggest Yellow Region

After we have obtained the mask, openCV's contours are identified in the binary mask.

```
std::vector<std::vector<cv::Point>> contours;
findContours(mask, contours, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
```

The function above is used to do the procedure explained.

Step 4: Finding the Center of the Yellow Object

After finding the largest contour, the centroid of this contour is computed as shown above:

```
cv::Moments moments = cv::moments(largest_contour);
int x = moments.m10 / moments.m00;
int y = moments.m01 / moments.m00;
```

But this is a 2D position. We need to get the 3D position of this centroid to locate the lantern. Using the depth data coming from the camera, the 2D centroid is mapped to a 3D coordinate, as shown below:

```
int index = y * cloud->width + x;
pcl::PointXYZ pcl_point = cloud->points[index];
```

However, after this operation, we need to convert this 3D point from the camera frame to the world frame, so we use the TF2 transformation.

3.1.2 Tracking and Publishing Lanterns

Updating the List of Tracked Lanterns

We use the function below for this operation:

```
updateTrackedObjects(world_point.point);
```

Then, we check whether the detected lantern is new or already tracked by using this function updateTrackedObjects. This function is shown below:

```
void LanternDetectionNode::updateTrackedObjects(const geometry_msgs::Point&
    detected_point) {
    bool matched = false;
    for (auto& [id, obj] : tracked_objects_) {
        if (distance(obj.position, detected_point) < max_distance_same_detection_) {
            obj.detections.push_back(detected_point);
            obj.position = computeCentroid(obj.detections);
            matched = true;
            break;
        }
        if (distance(obj.position, detected_point) < min_distance_new_detection_) {
            return;
        }
    }

    if (!matched) {
        const auto new_id = tracked_objects_.size();
        tracked_objects_[new_id] = {detected_point, {detected_point}};
    }
}
```

We can see that if a new detection is close to an existing tracked lantern, its position is updated, but if it is far enough, it is registered as a new lantern.

Computing the Lantern's Centroid

The below function is used for centroid calculation:

```
geometry_msgs::Point LanternDetectionNode::computeCentroid(const std::vector<
    geometry_msgs::Point>& points)
```

The centroid of a lantern is computed by averaging its detected positions over multiple frames, which helps to smooth noise.

Step 4: Publish Detected Lanterns

Finally, we publish the detected lanterns:

```
pub_world_coordinates_.publish(world_point);
pub_markers_.publish(marker_array);
pub_all_lanterns_.publish(all_lanterns_msg);
```

3.2 Junction Detection Node

The **junction_detection** is responsible to find where are the possible directions to go.

3.2.1 Old Method

Before our current method, we used a convolutions created from an occupancy grid at the altitude of our drone to detect junctions. From this, it constructed a 2D binary map. A skeletonization step then yields a "thinned" version of navigable space. Next, a small 3×3 convolutional kernel K is applied to the skeleton, as shown in (3.2.1), to detect branching points (junctions):

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 10 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Each cell gets a value from this kernel and a branching threshold determines whether $s(x, y)$ is deemed a junction candidate. For each such junction candidate, the node samples multiple outward directions and checks whether there is sufficient free space along each direction to count it as an "outgoing" corridor.

Once a junction has been consistently observed enough times around a proximity, we include in the planner its position and the detected outgoing orientations. A visualization of this 2D junction-finding process is depicted in 3.

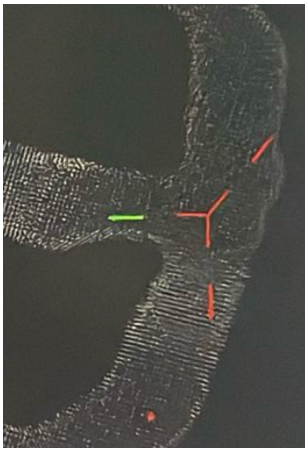

Figure 3: Visualization of detected junction with the old convolution algorithm.

The method worked nearly all the time and we used a planner using these entrance points but we wanted go into an even more robust algorithm which will be explained below. Additionally, with this approach we had the downsides that a dead-end detection node and more logic to detect loop-closures are needed.

3.2.2 How the currently used Node Works

The node continuously processes an incoming `Octomap` to identify unexplored "outgoing" regions within a cave. It converts the Octomap to Octree format using free leaf cells adjacent to unknown space (frontier points) and filters them based on local density. Those frontier points are then clustered, and cluster centroids far enough from the cave entry are labeled as junction points. Finally, a goal point from the latest "outgoing" can be published to guide the drone's navigation. These junction points are pursued to discover new regions in the cave. This approach also solves the problem of dead-ends as the frontier in front of the drone disappears after all walls are detected, and the last not discovered "outgoing" of the junctions already visited is published as the goal point. A visualization of the approach in 2D is shown in fig. 4.

Step 1: Extracting Frontier Points from the Octomap.

The core logic runs when a new octomap data arrives. First, the `Octomap` message is converted into an `OcTree` to traverse over the leaves to look for frontier points.

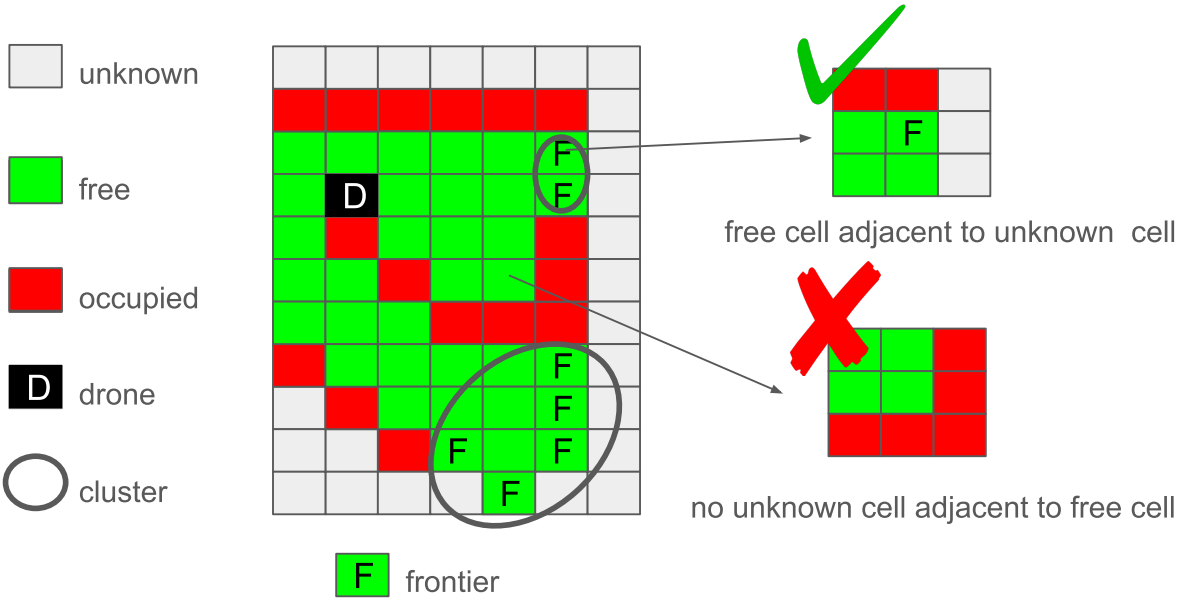

Figure 4: Visualization of the Junction detection algorithm in 2D. We first find the frontier points by checking if an unknown cell exists near the free cells. Then filter and cluster the frontiers to obtain junction points.

```
latest_octree_ = dynamic_cast<octomap::OcTree*>(tree);
```

Then, we scan through the free leaf cells:

```
for (auto it = tree.begin_leafs(); it != tree.end_leafs(); ++it) {
    if (tree.isNodeOccupied(*it)) continue;
    octomap::point3d pt = it.getCoordinate();
    bool isFrontier = false;

    // Offsets in X, Y, Z for neighbors
    std::vector<octomap::point3d> neighbors = {
        {pt.x() + resolution, pt.y(), pt.z()},
        {pt.x() - resolution, pt.y(), pt.z()},
        {pt.x(), pt.y() + resolution, pt.z()},
        {pt.x(), pt.y() - resolution, pt.z()},
        {pt.x(), pt.y(), pt.z() + resolution},
        {pt.x(), pt.y(), pt.z() - resolution}
    };
```

and mark them as frontiers if any of their 6-axis neighbors (3D) is unknown.

```
    for (auto& nbr : neighbors) {
        if (tree.search(nbr) == nullptr) {
            isFrontier = true;
            break;
        }
    }

    if (isFrontier) all_junctions->push_back({pt.x(), pt.y(), pt.z()});
}
```

Step 2: Filtering Frontier Points Based on Local Density.

To eliminate noise, each frontier point must have a minimum number of neighbors within a search radius. A KD-Tree is used for efficient lookup:

```
pcl::KdTreeFLANN<pcl::PointXYZ> kdtree;
kdtree.setInputCloud(all_junctions);

for (size_t i = 0; i < all_junctions->size(); ++i) {
    if (hasEnoughNeighbors(kdtree,
                           (*all_junctions)[i],
                           search_radius_,
                           min_neighbors_)) {
        filtered_junctions->push_back((*all_junctions)[i]);
    }
}
```

Here, `hasEnoughNeighbors` checks if the point has at least `min_neighbors_` points within `search_radius_`:

```
bool JunctionDetection::hasEnoughNeighbors(
    const pcl::KdTreeFLANN<pcl::PointXYZ>& kdtree,
    const pcl::PointXYZ& point,
    float radius,
    int min_neighbors)
{
    std::vector<int> point_indices;
    std::vector<float> point_distances;
    return (kdtree.radiusSearch(point,
                                radius,
                                point_indices,
                                point_distances)
            >= min_neighbors);
}
```

Step 3: Clustering the Filtered Points and Computing Centroids.

The Euclidean Cluster Extraction in PCL groups close points.

```
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance(10.0);
...
ec.setInputCloud(junctions);
ec.extract(cluster_indices);
```

For each cluster, a centroid is calculated and it is checked, whether this cluster is not close to the cave entry as they would be false positives:

```
for (const auto& cluster : cluster_indices) {
    float sum_x = 0, sum_y = 0, sum_z = 0;
    int count = 0;
    for (int idx : cluster.indices) {
        sum_x += junctions->points[idx].x;
        sum_y += junctions->points[idx].y;
        sum_z += junctions->points[idx].z;
        ++count;
    }
    pcl::PointXYZ centroid(sum_x / count, sum_y / count, sum_z / count);

    // Filter out centroids near the cave entry
    double dist = std::sqrt(
        std::pow(centroid.x - cave_entry_.x(), 2) +
        std::pow(centroid.y - cave_entry_.y(), 2) +
        std::pow(centroid.z - cave_entry_.z(), 2)
    );
    if (dist > cave_entry_clearance_) {
        current_frame_centroids->push_back(centroid);
    }
}
```

Step 4: Maintaining and Updating a Global List of Centroids.

When new centroids are detected, they are merged into a global list `detected_centroids_`. Each new centroid is matched with existing ones within a threshold distance, ensuring the list remains up to date:

```
for (auto & old_c : detected_centroids_) {
    for (size_t i = 0; i < new_centroids->size(); ++i) {
        double dx = old_c.x - (*new_centroids)[i].x;
        double dy = old_c.y - (*new_centroids)[i].y;
        double dz = old_c.z - (*new_centroids)[i].z;
        double dist_sq = dx*dx + dy*dy + dz*dz;

        if (dist_sq < update_threshold_*update_threshold_) {
            old_c = (*new_centroids)[i]; // Update position
            matched_new[i] = true;
            break;
        }
    }
}
```

Any truly new centroids that do not match an existing entry are added to the global list.

Step 5: Publishing the Latest Detected Junction as a Goal. After updating the global list of centroids, the node publishes the newest centroid as a `fla_msgs/GlobalPoint` to the `/goal_point` topic:

```
if (!detected_centroids_.empty()) {
    publishGoalPoint(detected_centroids_.back());
}
```

3.2.3 Summary

In summary, this Junction Detection Node takes an `Octomap`, extracts "frontier" points (free cells near unknown space), filters them for density, clusters them, and computes the clusters' centroids. The newest centroid that is sufficiently far from the cave entrance is published as a high-level *goal point*, which can then be used by other planners in the system to guide exploration or navigation within the cave.

4 Detailed Explanation of the Planning Nodes

4.1 State Machine Node

The **state_machine** node manages the UAV's operational states, ensuring smooth transitions between different flight phases. It consists of the following states:

4.1.1 How the Node Works:

The node subscribes to the drone's current position and the lantern detection topic to determine its state. Based on the drone's state, this node publishes a control signal (boolean flag) to other nodes or trajectory points. Below is a summary of the states in the state machine. Figure 5 illustrates the behavior of the drone.

- **Take Off**: The UAV initiates flight and ascends to a predefined altitude.
- **Fly to Cave**: The UAV navigates towards the cave entrance using a predefined trajectory.
- **Explore Cave**: The UAV explores the cave, searches for lanterns, and maps the environment.
- **Fly Back**: After completing the exploration, the UAV returns to the starting position.
- **Land**: The UAV executes a controlled landing sequence.
- **Stop**: The mission is completed, and the UAV stops all operations.

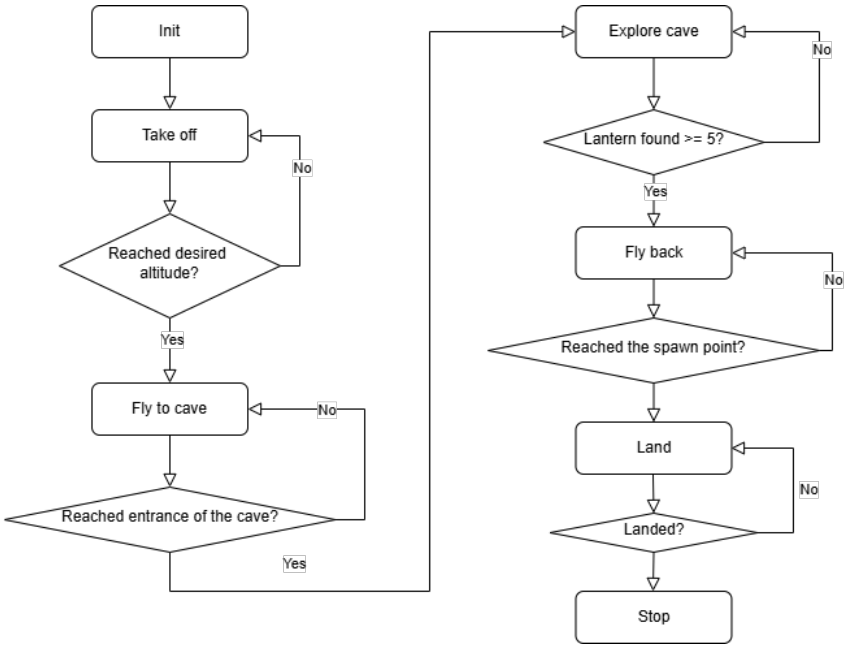

Figure 5: Flowchart of the state machine

4.2 Cave Explorer Node

The **cave_explorer_node** is a global planner (compared to local planning: only space dimension, no time dimension) that is based on the work of Chen et al. [CRK15], transferred to a 3D space.

4.2.1 How the Node Works:

The node has two modes. It can explore the space with and without a goal point. If there is no goal point, an artificial goal point is placed in front of the ego position with twice the distance as the max planning distance and the computation stops whenever a max planning distance is reached. If there is a

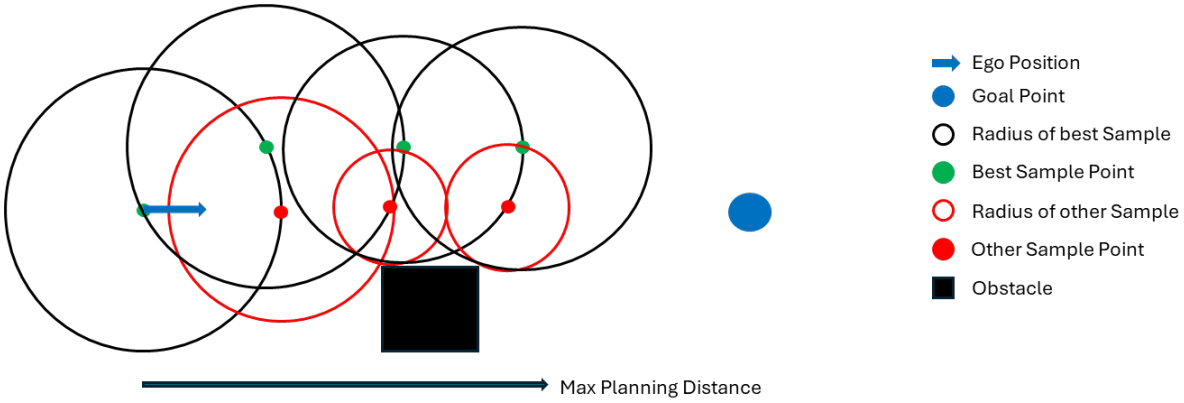

Figure 6: Visualization of the basic concept of the Cave Explorer in 2D.

goal point given, the algorithm stops as soon as the max planning distance or the goal point is reached. A visualization of the basic concept is shown in fig. 6.

Step 1: Filtering Point Cloud

First, we filter the Point Cloud to improve the computation time for cases with large maps by a sphere with two times the max planning distance as radius. Additionally, we save the points in a k-d Tree to improve the runtime of nearest-neighbor lookups.

```
# Filter point cloud to only include points in a region around the drone
max_distance = 2 * self.max_planning_distance

filtered_cloud = [point for point in full_cloud
                  if np.linalg.norm(point - self.current_position) <= max_distance]

# Build a KD-Tree for fast nearest-neighbor lookups
if len(filtered_cloud) > 0:
    self.kd_tree = cKDTree(np.array(filtered_cloud))
else:
    self.kd_tree = None
```

Step 2: Entering the Exploring Loop

As the planner daisy-chains nodes based on a heuristic, they are saved as dictionaries with 'position', 'father', and 'radius'. The maximum radius calculation of a node is based on its position and the closest occupied cell and a `max_radius` parameter.

```
point_nodes = [{
    'position': start_position,
    'father': None,
    'radius': 0.0
}]
best_node = point_nodes[0]
best_node['radius'] = self.calculate_max_radius(best_node["position"])
total_distance = 0.0

# Inner loop
while total_distance < self.max_planning_distance:
```

Step 3: Sampling new nodes and evaluating the 'best' node

To get new nodes, they are sampled on the maximum-radius-sphere of the 'best_node'. This is achieved by rotating the point directly in front of the node in yaw and pitch direction. The rotation steps and limits are implemented as parameters and determine the number of samples. The evaluation of the heuristic is also parameterized. A node is better if it is farther from the starting node, closer to the goal point, and has a larger radius. After iterating over all samples, the next iteration of the exploration

loop continues with the 'best' node and checks whether the max planning distance or the goal point is reached. If one of the limits is reached, the algorithm leaves the exploration loop.

```
for sampled_point, max_radius in sampled_points:

    #skip points that are too close to obstacles
    if max_radius < self.min_radius:
        continue

    #Calculate distances(to the goal point and start point) to use for heuristic
    distance_to_goal = np.linalg.norm(sampled_point - goal_point)
    distance_from_start = np.linalg.norm(sampled_point - start_position)

    # Heuristic function: reward progress, goal approach, and open space
    value = ((self.distance_from_start_value * distance_from_start)
             - (self.distance_to_goal_value * distance_to_goal)
             + (self.max_radius_value * max_radius))

    if value > best_value:
        best_value = value
        best_candidate_node = {
            'position': sampled_point,
            'father': current_node,
            'radius': max_radius
        }
```

Step 4: Reconstructing the Path

In this part of the code, the path for the drone is reconstructed. Starting with the best node, the fathers are recursively added to the path. Since we started with the best node, the path needs to be reversed. The orientation of each waypoint is pointing to the next waypoint based on the yaw dimension.

```
path_nodes = []
current_node = point_node # start from end
while current_node is not None:
    path_nodes.append(current_node) # add the node to the path
    current_node = current_node['father'] # get the father of each node

# reverse it to obtain the path from start to end
path_nodes.reverse()

# create waypoint from these points
for i in range(1, len(path_nodes) - 1):
    node = path_nodes[i]
    next_node = path_nodes[i + 1]
    self.add_arrow_marker(next_node['position'], node['position'], i)

    global_point = GlobalPoint()
    global_point.point = self.to_geometry_msg_point(node['position'])

    # Calculate orientation as the heading to the next point
    global_point.orientation = np.arctan2(
        next_node['position'][1] - node['position'][1],
        next_node['position'][0] - node['position'][0]
    )
```

4.2.2 Summary

Using this heuristic-based approach allows to keep the drone stable in the middle of the cave and avoids risky behavior by flying close to a wall if it is not necessary.

4.3 Breadth-First Search (BFS) Path Planning Node

The **bfs_node** is responsible for planning a return path for a drone using the Breadth-First Search (BFS) algorithm. It constructs a graph of visited locations and determines the shortest path from a start position to a goal position. This is done by subscribing to the drone's current position while exploring the cave and appending this to the graph after a certain distance from each other. When the fly-back state is running, the state machine node shall publish the drone's current path as a starting point and

the entrance of the cave as the goal point. The BFS then receives these points and computes the shortest path using BFS. This path is then published to a topic subscribed by the trajectory planner node.

4.3.1 How the Node Works:

The node continuously listens for the drone's position and dynamically builds a graph of visited locations. It subscribes to start and goal points and runs BFS to find the shortest path.

Step 1: Subscribing to Necessary Topics and initializing publishers

The node subscribes to the following topics:

- `/current_state_est` (Odometry): Provides the drone's real-time position.
- `/fly_back_start_points` (Point): Receives the start position for BFS.
- `/fly_back_goal_points` (Point): Receives the goal position for BFS.

Additionally, the node publishes the planned path to `/global_path` for the trajectory planner node and visualizes the graph in `/graph_visualization`.

Step 2: Storing Visited Locations

As the drone moves, the node records its positions and ensures that new positions are only added if they are at least 20 meters away from existing nodes. This helps in reducing redundant graph nodes.

```
if not any(np.linalg.norm(self.current_position - np.array(loc)) < self.
    min_distance_between_nodes for loc in self.visited_locations):
    self.visited_locations.append(self.current_position)
    self.update_graph()
```

The adjacency list is updated based on the proximity between nodes, connecting new nodes to nearby ones within a 35-meter radius.

Step 3: Finding the Closest Nodes

Before executing BFS, the closest nodes to the start and goal points are identified using Euclidean distance:

```
for i, location in enumerate(self.visited_locations):
    dist = np.linalg.norm(np.array(location) - point)
    if dist < min_dist:
        min_dist = dist
        closest_index = i
```

Step 4: Running BFS to Find the Shortest Path

Once the start and goal indices are determined, the BFS algorithm is executed to find the shortest path:

```
q.append(start)
visited[start] = True
while q:
    curr = q.popleft()
    if curr == goal:
        break
    for neighbor in self.adj.get(curr, []):
        if not visited[neighbor]:
            visited[neighbor] = True
            parent[neighbor] = curr
            q.append(neighbor)
```

After BFS completes, the path is reconstructed by backtracking through the parent nodes.

Step 5: Computing Orientations

For each waypoint in the path, the orientation (yaw) is calculated to ensure the drone faces the next waypoint:

```
dx = path_points[i+1][0] - path_points[i][0]
dy = path_points[i+1][1] - path_points[i][1]
pose.orientation = math.atan2(dy, dx)
```

Step 6: Publishing the Planned Path

The planned path is visualized using a `MarkerArray`, where nodes appear as green spheres and edges as red lines. The path itself is published as a blue line strip:

```
path_marker.type = Marker.LINE_STRIP
path_marker.color.b = 1.0
path_marker.color.a = 1.0
```

The waypoints are then published in batches to guide the drone in real-time. In RVIZ, the nodes and edges are illustrated in Figure 7

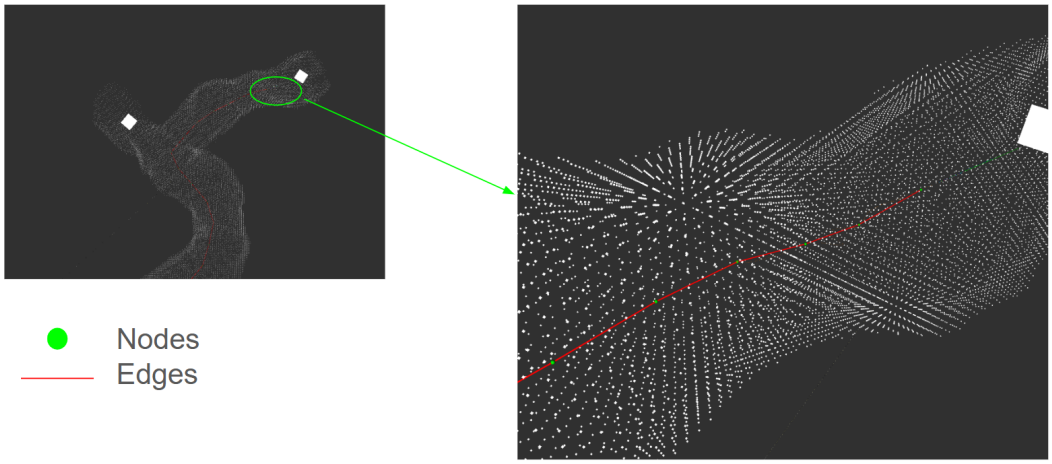

Figure 7: Nodes and edges from the BFS node

Step 7: Updating the Graph and Visualization

Whenever a new location is added, the graph structure is updated, ensuring connectivity among nodes within the maximum allowed distance.

```
for i in range(current_index):
    if np.linalg.norm(self.visited_locations[current_index] - self.visited_locations[i])
         <= self.max_distance_between_nodes:
        self.add_edge(current_index, i)
```

Finally, the updated graph is visualized to reflect the latest state of the exploration.

4.3.2 Summary

This BFS-based path planner enables a drone to navigate back to a goal position efficiently by dynamically constructing a graph of visited locations and computing the shortest path using BFS.

5 Images From Simulation

In this section, important instances from the simulation are shown with screenshots from RViz and explained briefly.

Step 1: Taking-Off

The drone takes-off and detects the lantern (red ball) next to the starting position.

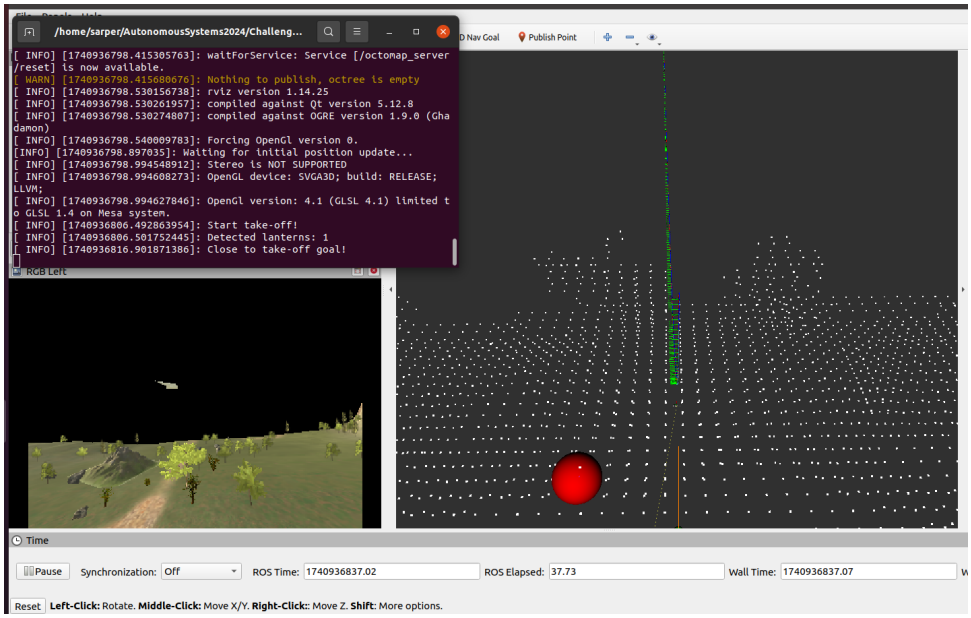

Figure 8: Take-Off

Step 2: Entering the Cave

The drone approaches the cave entrance and enters the cave

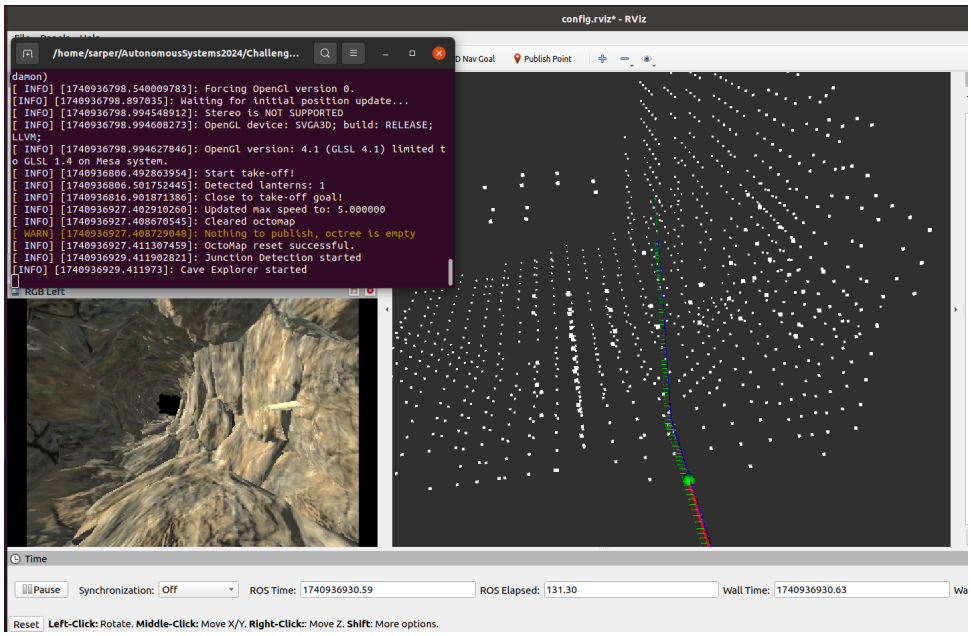

Figure 9: Cave Entrance

Step 3: Lantern Detection 1

The drone detects the first lantern in the cave and prints an information message.

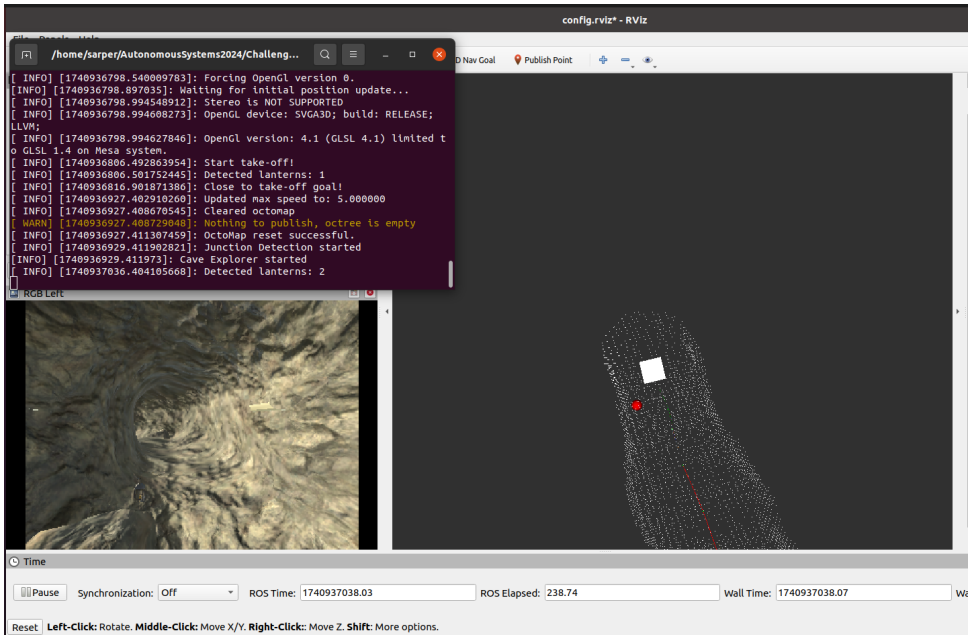

Figure 10: Lantern 1 detection

Step 4: First Junction Detection

The drone detects the first junction in the cave and proceeds from one of the outgoings.

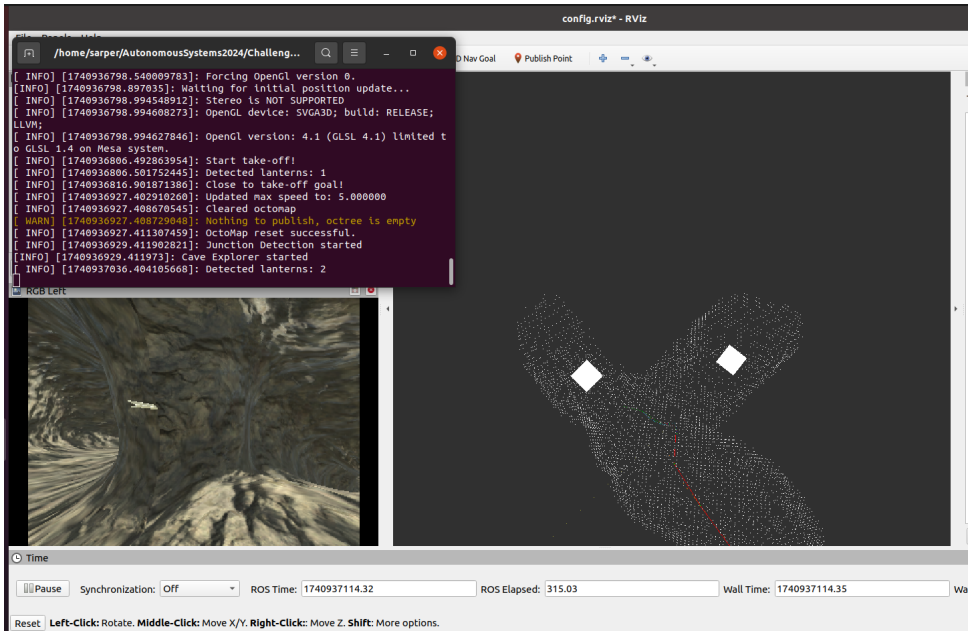

Figure 11: First Junction

Step 5: Lantern Detection 2

The drone detects the second lantern in the cave and prints an information message.

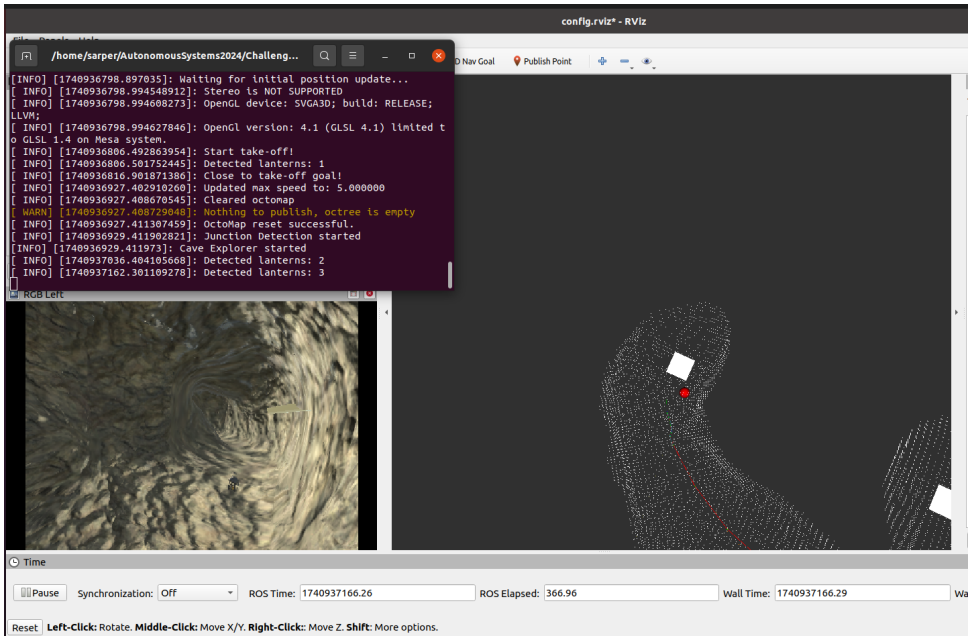

Figure 12: Lantern Detection 2

Step 6: Second Junction Detection and Entrance to Dead-end

The drone detects the second junction and proceeds with one of the outgoings which goes to the dead-end.

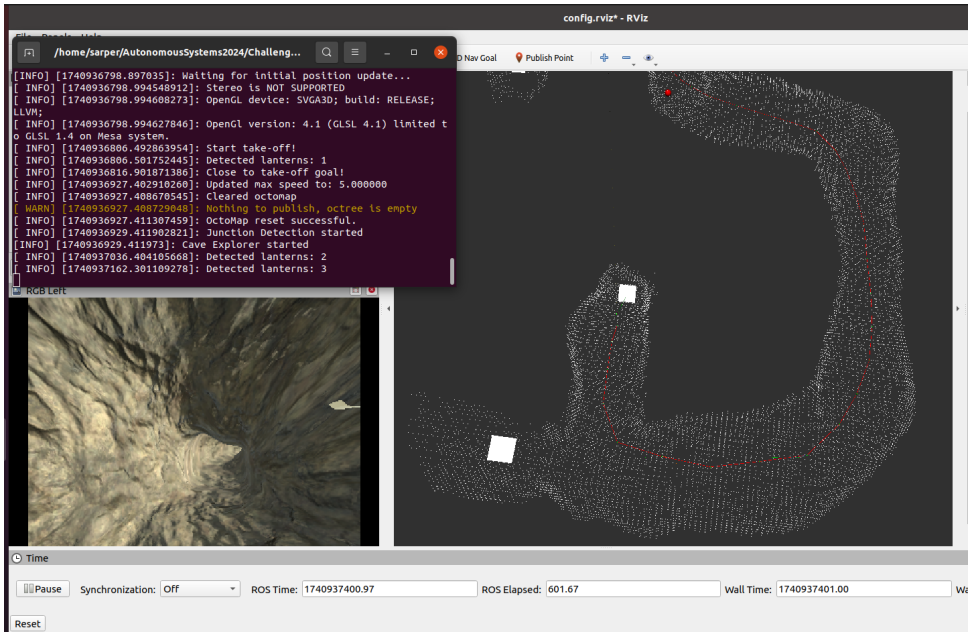

Figure 13: Junction 2 Detection

Step 7: Lantern Detection 3

The drone detects the third lantern in the cave and prints an information message.

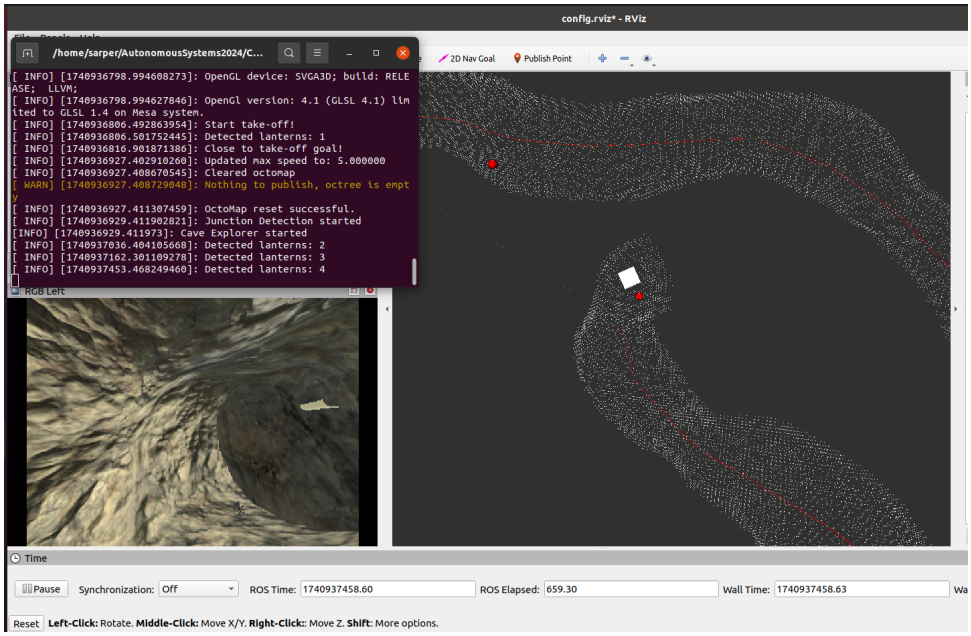

Figure 14: Lantern detection 3

Step 8: Exiting From Dead-End

The drone gets out from the dead end because it needs to go to the last undiscovered outgoing which is shown with the white square.

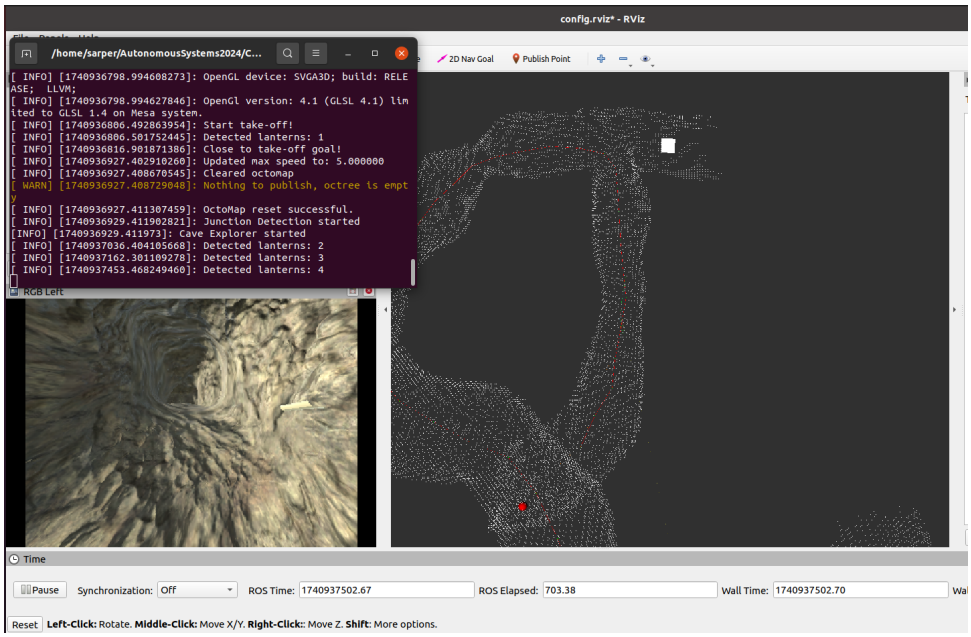

Figure 15: Exiting the dead-end

Step 9: Lantern Detection 4 and Triggering of Fly-Back

The drone detects the fourth lantern in the cave and prints an information message. And Flying back is triggered because the drone discovered all the lanterns.

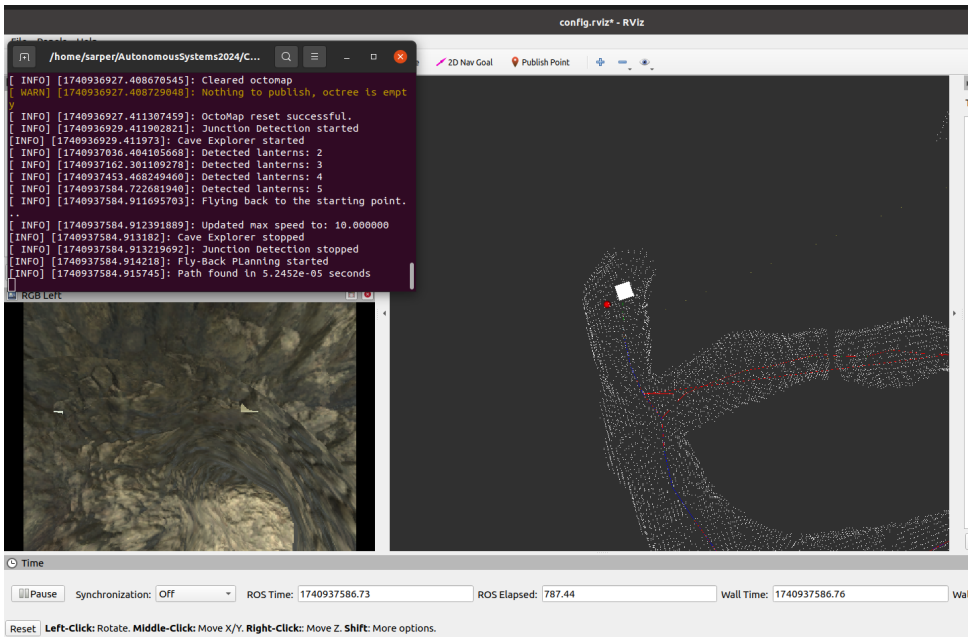

Figure 16: Last Lantern detection and Flying Back

Step 10: View of Whole Detected Cave Until Finding 4 Lanterns

In this image, the whole discovered cave and all 4 detected lanterns (red spheres) can be seen.

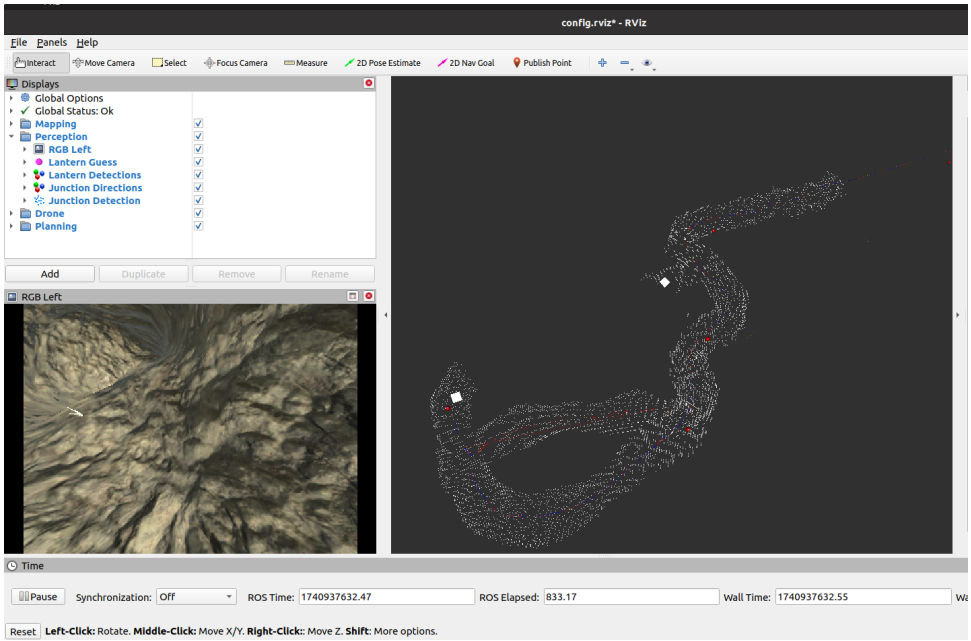

Figure 17: Whole cave system

Step 11: Exiting The Cave

The drone successfully reaches the exit of the cave.

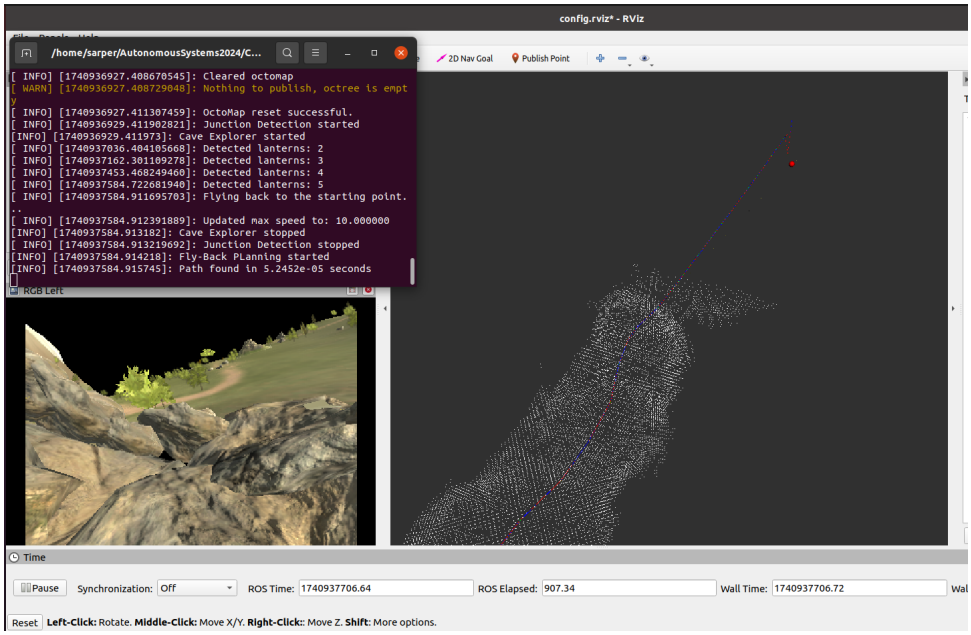

Figure 18: Exiting the cave

Step 12: Landing and Lantern Location Log

The drone successfully lands and prints the locations of all lanterns.

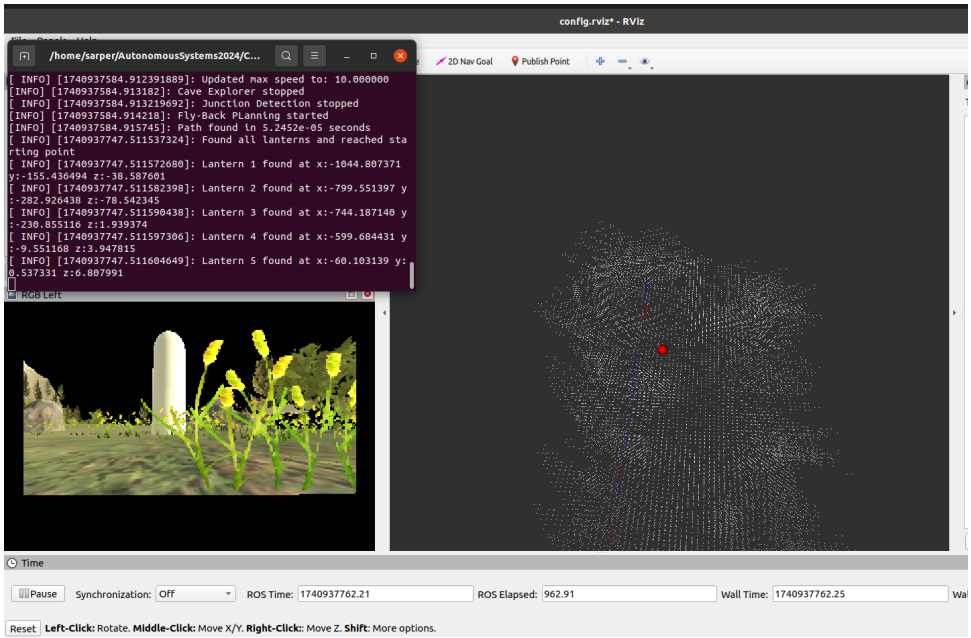

Figure 19: Landing

6 Conclusion

In this project we implemented autonomous cave exploration using a drone within a simulated Unity environment. The system successfully integrates four modules namely perception, planning, control and mapping to achieve two primary objectives: detection of the lanterns inside the cave, and a search algorithm using a detailed 3D occupancy map to support navigation.

The perception modules, including the Lantern Detection Node and the Junction Detection Node, process visual and depth data to identify key features in the environment. Techniques such as image segmentation, point cloud filtering, and clustering allow the system to accurately detect lanterns and junction points to pursue.

The planning modules, which comprise the BFS Path Planner, State Machine, Trajectory Planner, and Cave Explorer, enable the drone to dynamically construct a global path, plan safe trajectories, and manage mission states effectively.

The control modules, enable our drone to achieve desired acceleration, velocity, orientation and position, whereas mapping module OctoMap creates the map of the environment so that planning and perception modules work properly.

Overall, integration of perception, planning, and control components, led to a system that can autonomously navigate in this complex and unstructured cave environment.

References

[BOAS15] Michael Burri, Helen Oleynikova, Markus W. Achtelik, and Roland Siegwart. Real-time visual-inertial mapping, re-localization and planning onboard mavs in unknown environments. In *Intelligent Robots and Systems (IROS 2015), 2015 IEEE/RSJ International Conference on*, Sept 2015.

[CRK15] Chao Chen, Markus Rickert, and Alois Knoll. Path planning with orientation-aware space exploration guided heuristic search for autonomous parking and maneuvering. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1148–1153, 2015.

[HWB+13] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. Software available at https://octomap.github.io.

[LLM10] Taeyoung Lee, Melvin Leok, and N. Harris McClamroch. Geometric tracking control of a quadrotor uav on se(3). In *49th IEEE Conference on Decision and Control (CDC)*, pages 5420–5425, 2010.

[RBR16] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In *Robotics Research*, pages 649–666. Springer, 2016.