Autonomous Systems Project

Group 4
Fabian Sommer ‖ Ramkrishna Chaudhari
Mohamad Alattar ‖ Berhan Sofuoglu

Technical University of Munich

1 Introduction

This report documents the project work of Group 4 in the Autonomous Systems course (WiSe 24/25) at TUM. The primary objective was to enable a drone to autonomously navigate and locate all the lanterns within a cave environment, provided as a Unity world.

As illustrated in Figure 1, autonomous navigation involves integrating multiple components, each playing a crucial role in the system.

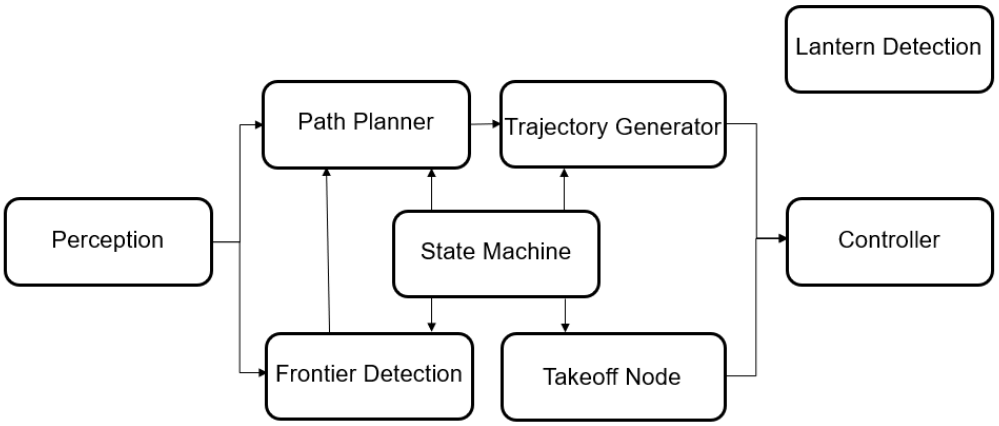


Figure 1: General Architecture

The code, accessible via `GitHub`[1], can be build by following the descriptions in the **README.md**. It can then be launched using the following command:

```
roslaunch simulation combined.launch
```

This command initializes all essential ROS packages and nodes, including: **State Machine, Perception, Frontier Detector, Path Planner, Trajectory Generator, Controller, Takeoff Node and Lantern Detector**.

Some components such as the State Machine, Perception, Trajectory Generator, Controller, and Lantern Detector run continuously throughout the simulation. However, the Frontier Detector and Path Planner are only activated while the drone is exploring the cave. The Takeoff Node just runs in the very beginning until the takeoff is completed.

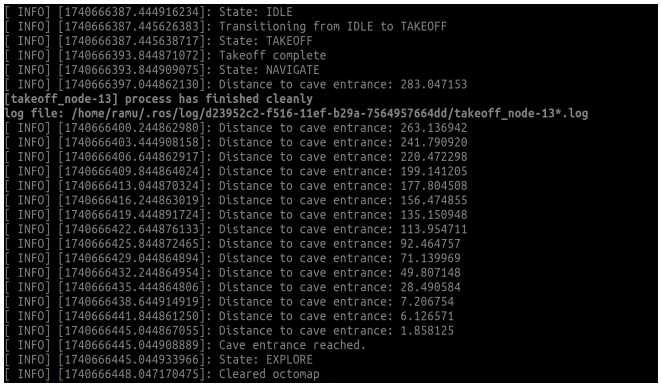
Details regarding individual team member contributions are outlined in Section 11, while a complete ROS graph is available in Appendix A.

[1] `https://github.com/Ramu691/Autonomous_sys_2025_G4`

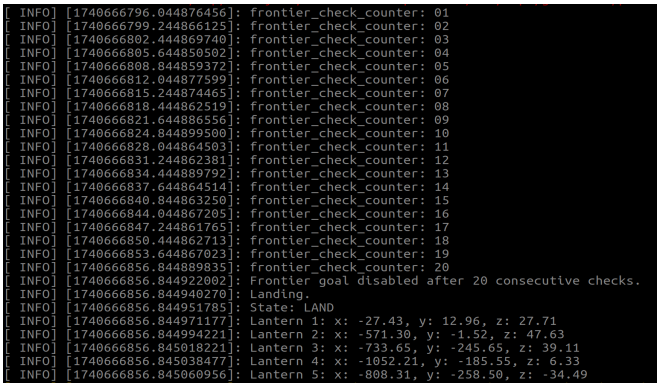
2 State Machine

This node serves as an intermediary to keep track of mission phases, manage the goals of the drone and provide debug and testing capabilities for overseeing the whole mission if necessary. It is under the package `system/src/`. It listens to `current_state_est, frontier_goal, num_lanterns, lantern_positions`. By tracking these topics, the state machine determines the current goal and updates it. It publishes to only 2 topics which are `stm_mode` and `traj_pub`. The state updates happen by the publishes to the `stm_mode`. There 5 states corresponding to 4 phases of the mission which are the following:

- `IDLE` was implemented as a placeholder fallback in the event of a failure.
- `TAKEOFF` marks the start of the mission. It tracks the drone's state while comparing it to parameter thresholds. When the drone is sufficiently far from the ground, the drone transitions into `NAVIGATE`.
- `NAVIGATE` phase flies the drone to the cave entrance. When the threshold parameters are met for the `NAVIGATE` phase are met the drone transitions into the `EXPLORE` phase.
- `EXPLORE` phase is where the drone explores the cave while looking for the lanterns. The state machine tracks the progress by listening to the aforementioned topics `frontier_goal, num_lanterns, lantern_positions`. After all lanterns are found and the frontiers being published are not at least 2 meters apart, this signals that the `EXPLORE` phase has ended and the drone transitions into the `LAND` phase.
- `LAND` handles the drone's slowl approach the ground and stop. This state marks the end of the mission. This phase does the only other publish than `stm_mode`, which is to the `traj_pub`.



(a) IDLE-NAVIGATE-EXPLORE



(b) EXPLORE-LAND and printing Lantern positions

Figure 2: State machine transitions as ROSINFO

3 Lantern Detection

This node is used to detect the lanterns in the environment. It receives image data through `sensor_msgs/Image`, odometry from `nav_msgs/Odometry`, and camera parameters from `sensor_msgs/CameraInfo`. Below is a concise overview of how these elements interact to detect and localize lanterns.

3.1 Concept of Object Detection

All lantern detection occurs in the `semanticCallback` function, which listens to a `MONO8` semantic image topic. The incoming data is converted to an `cv::Mat` using `cv_bridge/cv_bridge.h` (this bridge seamlessly transitions from `sensor_msgs/Image` to OpenCV data structures). Once the semantic image is available as a matrix, connected-component analysis (via OpenCV routines from `opencv2/imgproc/imgproc.hpp`) groups pixels that correspond to lanterns. A centroid is then computed for each such group:

$$(\bar{x}, \bar{y}) = \big(\tfrac{1}{N}\sum_{i=1}^{N} x_i,\ \tfrac{1}{N}\sum_{i=1}^{N} y_i\big).$$

Here, $\bar{x}$ and $\bar{y}$ indicate the average pixel coordinates of the detected blob.

3.2 Depth-Guided Position Estimation

For each centroid, the code invokes `pixelTo3D` to convert the 2D position $(\bar{x}, \bar{y})$ into 3D coordinates (X, Y, Z) in the camera frame. The depth image is also published as a `sensor_msgs/Image`, which again is turned into an `cv::Mat` through `cv_bridge`. The intrinsic camera parameters (focal lengths and principal point) come from `sensor_msgs/CameraInfo` and are accessed to implement the pinhole projection model:

$$X = \frac{(\bar{x} - c_x)\, Z_{\text{depth}}}{f_x}, \quad Y = \frac{(\bar{y} - c_y)\, Z_{\text{depth}}}{f_y}, \quad Z = Z_{\text{depth}},$$

with Z_{depth} in millimeters. If $Z_{\text{depth}} = 0$, the detection is discarded to avoid invalid 3D coordinates.

3.3 Transformation to the Global Frame

After computing (X, Y, Z) in the camera frame, the function `transformToWorldFrame` uses `tf2_ros` (from `tf2_ros/transform_listener.h` and `tf2_ros/buffer.h`) to look up the latest transformation from the camera coordinate system to a global "world" frame. This transformation, $\mathbf{T}_{\text{global}\leftarrow\text{camera}}$, is applied as

$$\begin{pmatrix} X_{\text{global}} \\ Y_{\text{global}} \\ Z_{\text{global}} \end{pmatrix} = \mathbf{T}_{\text{global}\leftarrow\text{camera}} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

Thus, each detection in camera coordinates is mapped to global coordinates. The node also retrieves the drone's own global pose via `nav_msgs/Odometry`, providing the drone position for further filtering.

3.4 Duplicate Detection and Distance Thresholding

With each lantern candidate placed in the global frame as a `geometry_msgs/Point`, two checks then decide whether it should be stored or ignored:

1. *Maximum Distance from Drone:* If the Euclidean distance (computed by `distanceBetween`) between the drone's global position (from `nav_msgs/Odometry`) and the new lantern point exceeds a specified threshold, the candidate is skipped as irrelevant.
2. *Duplicate Check:* The function `isNewLantern` compares the new point to a list of previously found lanterns. If the 3D Euclidean distance to any known lantern is below a threshold, the code classifies it as a re-detection rather than a new discovery.

Distances are calculated according to

$$d(\mathbf{p}_1, \mathbf{p}_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}.$$

If it is identified as new, the system publishes positions via `std_msgs/Float32MultiArray` and textual messages with `std_msgs/String`.

3.5 Calibration and Error Minimization

The code depends on reliable calibration parameters (intrinsic and extrinsic) so that the pinhole projection model accurately reflects physical reality. Typically, these parameters are refined through a maximum likelihood or least-squares approach that minimizes reprojection error between known 3D references and their observed 2D image coordinates. In this node, it is assumed such calibration has already been done, and that the final values are available in `sensor_msgs/CameraInfo` (for intrinsics) and through `tf2_ros` transforms (for extrinsics). This ensures lantern detections are consistently placed in the global frame for subsequent use in exploration and navigation.

4 Takeoff Node

The node has the very simple task to perform the takeoff of the drone. It listens to the `/stm_mode` topic to determine the current state of the drone. If the state is "TAKEOFF" it publishes the takeoff command in the form of takeoff point on the `/desired_state` topic, which is then performed by the controller described in section 9. After the state progresses to "NAVIGATE", the node is killed to not occupy processing power.

5 Perception

The perception package consists of two functionalities. First of all it rectifies the depth camera image. Afterwards the image is integrated into an map, which then can be used by other nodes.

5.1 Image Processing

Image processing tasks are handled by the `perception_nodelet_manager`, which is responsible for managing multiple nodelets to ensure optimal performance [1]. Image processing tasks are spread out over multiple nodelets, which each perform small tasks. One nodelet perform the image rectification. This corrects the distortion from the camera model, ensuring accurate processing of other tasks using the depth camera image. The launch file sets up the following nodelet of type `image_proc/rectify` for image rectification[2]. Specifically for the depth camera, the following topics exist as input, and are then remapped to their corresponding topics in the simulation:

- `camera_info` to `/realsense/depth/camera_info`, which includes information about the distortion model
- `image_mono` to `/realsense/depth/image`, which is the topic where the image is received from the Unity simulation
- `image_rect` to `/realsense/depth/image_rect`, which is the topic where the rectified image is published

Additionally, the `cloud_converter` nodelet converts the rectified depth images into point cloud data of type `sensor_msgs/PointCloud2`, which is then published on the topic `/realsense/depth/image_rect` and used for creating the map[3].

5.2 Creation of a map

The creation of the map is performed using the `octomap_server` node, which uses the simulated depth camera to construct a three-dimensional occupancy map [4]. The octomap server subscribes to the `cloud_in` topic and uses the following parameters:

- **frame_id**: Specifies the reference frame for the map, set to `world`.
- **resolution**: Defines the resolution of the octomap, set to 4.5 m.
- **sensor_model/max_range**: Sets the maximum range 40 m.

A static transform publisher, `ros_to_realsense_depth_camera`, is necessary for providing the correct transformation to the drone to ensure that the point cloud is integrated in the right pose. In summary, the perception package manages the rectification of the depth image and the integration of 3D point cloud data into a global occupancy map. This map is then later used by other parts of the system to perform further tasks.

5.3 Visualization

To ensure proper functionality, RVIZ is used to verify the correct occupancy map [5]. In addition to that, RVIZ can be used to display further information provided by other packages for a better understanding and supervision of the current state of the system. The full octomap, displayed in RVIZ, is shown in Figure 7 in Appendix B.

6 Frontier Detection

Frontier-based exploration is a widely adopted method in robotic mapping, where the boundary between known and unknown regions, termed as "frontiers," serves as an exploration goal [6][7]. The fundamental idea is that a robot should navigate toward the frontier to maximize information gain and expand its mapped environment. This method enables autonomous systems to incrementally explore unknown spaces in an efficient manner. Frontiers are critical in autonomous navigation of the drone in this project since they guide the drone towards unexplored areas while minimizing redundant motion [8][9][10].

6.1 Frontier Identification and Selection

6.1.1 Frontiers identification

This process starts once the drone has reached to the `cave_entry_position` and the state is changed to `EXPLORE`. The OctoMap is reseted initially once to ensure the clustering algorithm to work. Otherwise it can lead to detection of multiple Frontier between the starting point and the cave entry. A frontier is defined as the boundary between explored and unexplored regions in a mapped environment [6]. In an OctoMap-based representation, a voxel is classified as a frontier if it meets the following criteria:

- It is labeled as free space (unoccupied and within navigable areas).
- It is adjacent to at least one unknown voxel.

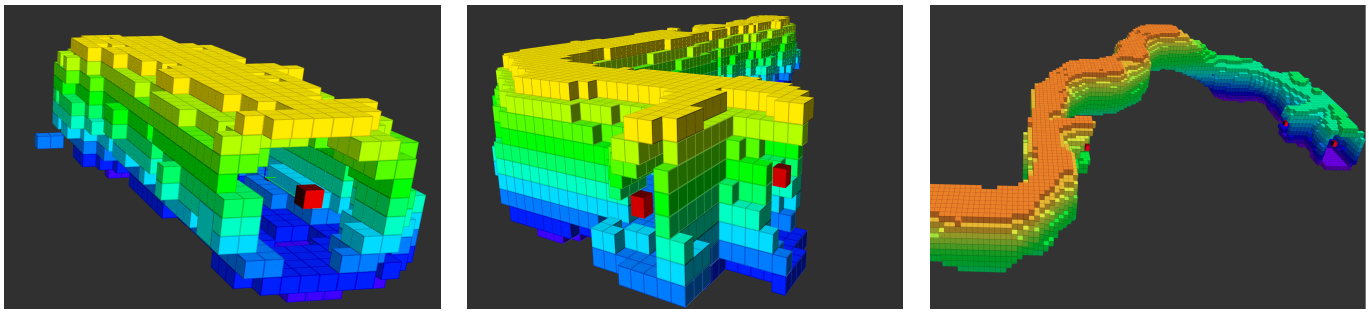
The `frontier_detector_node` from `frontier_detection` package systematically converts the OctoMap into a a new OcTree and applies this criterion to detect all frontiers in the available OctoMap. Once detected, these frontiers become candidates for further clustering and selection.

6.1.2 Frontier Clustering Using Mean-Shift

Since raw frontier points can be scattered, clustering is required to group nearby frontiers into distinct exploration targets. The `mean_shift_algorithm` is employed for this purpose due to its ability to identify clusters without predefining their number. The mean-shift process works as follows:

1. Each `frontier point` is treated as a data point in a `multi-dimensional space`.
2. A `Gaussian` kernel function assigns a weight to each point based on its proximity to other points.
3. Each point iteratively shifts toward the `weighted_mean` of nearby points until `convergence`.
4. The resulting clusters represent distinct `frontiers` position as `geometry_msgs/Point`.

Mean-shift clustering offers a significant advantage due to its adaptive nature, as it dynamically identifies the number of clusters based on density rather than requiring a predefined cluster count. This adaptability aids in selecting optimal exploration points while minimizing computational redundancy. The algorithm clusters points in regions with the highest density, allowing it to identify cluster centers. By measuring how many frontier voxels converge to these high-density points, an estimate of the cluster size can be obtained. Consequently, the clusters with the largest number of frontier points represent the most promising areas for the drone to explore. The frontiers are then published on topic `:/frontiers` as `visualization_msgs::MarkerArray` which can be visualized in RVIZ as red boxes for better situational awareness.



(a) Frontier during exploration (b) Frontiers at branches (c) Frontiers at branches

Figure 3: Frontier Visualization

As shown in Figure 3a, the algorithm identifies a single clustering point in cave exploration areas. However, when the drone reaches cave branches, as illustrated in Figure 3b and Figure 3c, it detects multiple clustering points typically corresponding to the separate branches. This behavior is expected and beneficial, especially in cases where the cave may contain small holes or gaps due to inaccuracies in the voxel grid. By focusing on the densest regions, the algorithm helps prevent false frontier detections along cave walls.

6.1.3 Frontier Scoring and Selection

Once frontiers are clustered, they are ranked based on a `heuristic score` that prioritizes:

- **Proximity**: Closer frontiers are preferred.
- **Visibility**: Frontiers with a high number of adjacent known free-space voxels are favored.
- **Yaw Alignment**: The drone prefers frontiers that require minimal rotation.

The scoring function is formulated as:

$$S(f) = -k_d \cdot d(f) + k_n \cdot N(f) - k_y \cdot |\theta(f) - \theta_{drone}| \tag{1}$$

where:

- $d(f)$ is the Euclidean distance from the drone to the frontier.
- $N(f)$ is the number of adjacent frontiers.
- $\theta(f)$ is the yaw alignment difference.
- $k_d(1), k_n(0.1), k_y(55.0)$ are weight coefficients used in this project.

The selected frontiers are then published on the `:/frontier_goal` topic as `custom_msgs::FrontierGoalMsg`, where `custom_msgs` contains the frontier positions information as `geometry_msgs/Point` and raytracing information as `isReachable`, enabling safe navigation within the cave environment.

7 Path Planning

Path planning involves generating a path from a `:/current_position` to the `:/frontier_goal` while avoiding obstacles. This structured pipeline allows seamless integration of `frontier_detector_node` and `rrt_planner_node` from `rrt_planner` package, enabling autonomous drone exploration in unknown environments. The path can only be planned if the state is `EXPLORE` and there is a `frontier` remaining.

7.1 RRT* Path Planning Algorithm

The RRT* (Rapidly-exploring Random Tree Star)[11] algorithm is employed from OMPL[12] due to its ability to efficiently explore high-dimensional spaces while ensuring asymptotic optimality. The configuration of State Space Representation is defined in three dimensions. The algorithm builds a tree structure in the configuration space to find a collision-free path. It follows these steps:

1. **Sampling**: A random sample is drawn from the free space.
2. **Nearest Node Selection**: The nearest node in the existing tree is identified.
3. **Steering**: The algorithm moves a fixed step toward the sampled point.
4. **Collision Checking**: The new edge is validated against the occupancy map to ensure obstacle-free movement.
5. **Rewiring**: If the new path provides a shorter cost, nearby nodes are reconnected to optimize the path.
6. **Path Extraction**: Once the goal is reached, the optimal path is backtracked from the goal node.

7.2 Collision Checking Mechanism

Collision detection is performed using an occupancy grid query:

1. Each potential node is checked against the OctoMap.
2. If an occupancy probability exceeds a predefined threshold `node->getOccupancy() <= 0.7`, the node is considered occupied and rejected.
3. `Ray-casting` techniques are used to check for occlusions along potential path segments.

This ensures that no path segment intersects an occupied space, maintaining safety for navigation.

7.3 Path Planning Output

The generated path is a series of waypoints leading to the `frontier_goal` and is published as `:/rrt_path` as `nav_msgs::Path` while ensuring collision-free exploration of the cave. The published path is visualized as green line in RVIZ.

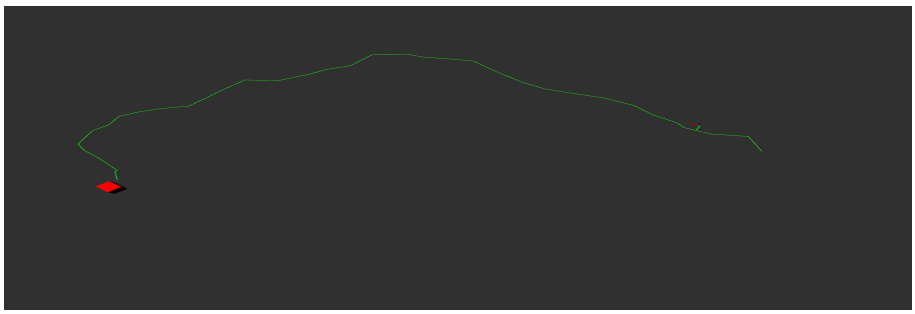


Figure 4: Path to the remaining Frontier at one of the branch

As shown in Figure 4, the path planner generates a path to the next frontier while considering the OctoMap for collision detection. Path optimization techniques, such as `snap` and `smoothing`, have not yet been implemented in this section. However, these techniques will be integrated into the `trajectory planner` in the next section to enhance flight performance.

8 Trajectory Generator

Trajectory generation is essential for ensuring smooth motion as the drone navigates through a cave environment. Instead of following waypoints with abrupt changes, the generated trajectory ensures continuous motion with smooth transitions. The primary objective is to compute a feasible trajectory that allows the drone to move efficiently while avoiding sudden accelerations that could destabilize the system.

8.1 Structure

The trajectory generator subscribes to the following topics:

- **/stm_mode (std_msgs::String)**: Indicates the current state of the state machine.
- **/rrt_path (nav_msgs::Path)**: Provides a series of waypoints representing a feasible path inside the cave.
- **/pose_est (geometry_msgs::PoseStamped)**: Provides the current estimated pose of the drone.

The generated trajectory is published to:

- **/desired_state (trajectory_msgs::MultiDOFJointTrajectoryPoint)**: Contains the computed position, velocity, acceleration, and orientation of the drone at each time step.

When the state machine is in the `NAVIGATE` mode, the trajectory generator constructs a trajectory to the cave entrance using the `generateCaveEntrancePath` function. Once the drone reaches the cave entrance, the state transitions to `EXPLORE`, where the trajectory generator follows the path provided by `/rrt_path` using the `buildTrajectory` function.

8.2 How it Works

8.2.1 Polynomial and Boundary Conditions

The trajectory is constructed using cubic polynomials of the form:

$$p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3, \tag{2}$$

which implicitly minimizes the Jerk. To ensure smooth transitions, the following boundary conditions are imposed at each waypoint:

$$p(0) = p_0, \quad p(T) = p_T, \tag{3}$$
$$p'(0) = v_0, \quad p'(T) = v_T. \tag{4}$$

These constraints ensure continuity in position and velocity while avoiding abrupt stops or changes. Acceleration boundary conditions were avoided as they introduced instabilities, leading to less reliable performance. The polynomial coefficients are computed using the `computeCubicCoeffs` function.
Jerk minimization was chosen over snap minimization due to its similar trajectory quality while requiring a lower-order polynomial, reducing computational complexity.

8.2.2 Yaw Representation

The yaw angle is computed using the velocity vector as:

$$\text{yaw} = \text{atan2}(v_y, v_x). \tag{5}$$

To ensure smooth rotation, the yaw is represented using quaternions. This conversion is handled via the `tf2` package and performed in the `publishPoint` function.

8.3 Special Cases and Exceptions in the Code

8.3.1 Path Comparison and Switching

When a new path is received via `/rrt_path`, the trajectory generator decides whether to switch to the new path using the `computePathDistance` function. If the new path is significantly shorter than the remaining distance on the current trajectory, it is accepted, and a new trajectory is generated using the `buildTrajectory` function.

8.3.2 Speed Determination

The speed along each segment is determined based on the segment length. The time allocation for each segment is computed as:

$$T = \max(1.0, d \times 0.15), \tag{6}$$

where is the segment length. This formula ensures that for short segments, the minimum time allocated is 1.0 seconds to avoid excessively high accelerations. For longer segments, the time scales proportionally with distance, maintaining a consistent speed. The resulting speed is:

$$v = \frac{d}{T}. \tag{7}$$

For segment lengths where $T = d \times 0.15$, the speed remains constant at:

$$v = \frac{1}{0.15} \approx 6.67 \text{ m/s}. \tag{8}$$

This value was chosen as it provided a reasonable balance between fast movement and stability, with no observed crashes during testing. The segment times are computed inside the `buildTrajectory` function

8.4 Trajectory Execution and Final Holding

The `timerCallback` function continuously evaluates the active trajectory segment and publishes the computed trajectory point. When the trajectory is completed, the drone holds its final position, ensuring it does not execute unnecessary movements.

9 Controller

The controller is subscribed to the `desired_state` and `current_state_est` topics to actually control the drone itself. The controller is a geometric controller, which is implemented based on the approach developed by Lee et al. [13] The parameter tuning provided by the base version of the source code was sufficient, and therefore the controller did not get adapted any further. The calculated rotor speeds for the motors are then published on the `rotor_speed_cmds` topic which ultimately controls the drone.

10 Result and Conclusion

The drone successfully navigated through the cave environment, flying smoothly and identifying all the lanterns. At the end of the mission, it also managed to land autonomously. However, there was an issue with detecting the lanterns. The semantic camera used for detection was recognizing lanterns even at a very distant range. To address this, the detection range was reduced to `30m` to improve lantern detection. Additionally, the drone may land only when all frontiers have been explored as well as all lanterns are detected and if the drone remains in the same frontier position for a consecutive period of time.

One of the main challenges faced was the non-deterministic nature of the path planning, despite large weighting towards the current yaw direction in the scoring function Equation 1. This sometimes led the drone to select alternative yaw directions rather than the most straightforward path.

Another issue arose with the RRT* planner, which occasionally generated error messages as shown in Figure 5 due to detecting potential collisions but this doesn't means that the drone is crashed or it cannot navigate. It a method used as `is_state_valid()` in rrt_planner which basically rewires again and will basically have a valid state. When a probable collision was identified, the planner attempted to compute a new path. This behavior may be attributed to the probability threshold used in the OctoMap, which is set to 0.7 in this project. The drone completed the mission in approximately 6 to 7 minutes at a velocity of 6.67 m/s, as stated in Equation 8. Lower velocities generally result in more stable flight performance, while higher velocities reduce stability. Table 1 shows the positions of the detected lanterns and are also published on the terminal as ROSINFO as shown in Figure 2b once the drone changes it state to `LAND`.

Lantern	**x**	**y**	**z**
Lantern 1	-27.43	12.96	27.71
Lantern 2	-571.30	-1.52	47.63
Lantern 3	-733.65	-245.65	39.11
Lantern 4	-1052.21	-185.55	6.33
Lantern 5	-808.31	-258.50	-34.49

Table 1: Coordinates of the Lanterns

Overall, while the system performed successfully in completing the exploration, Lantern detection and landing tasks, these issues highlight areas for further improvement in sensor calibration and path planning robustness.

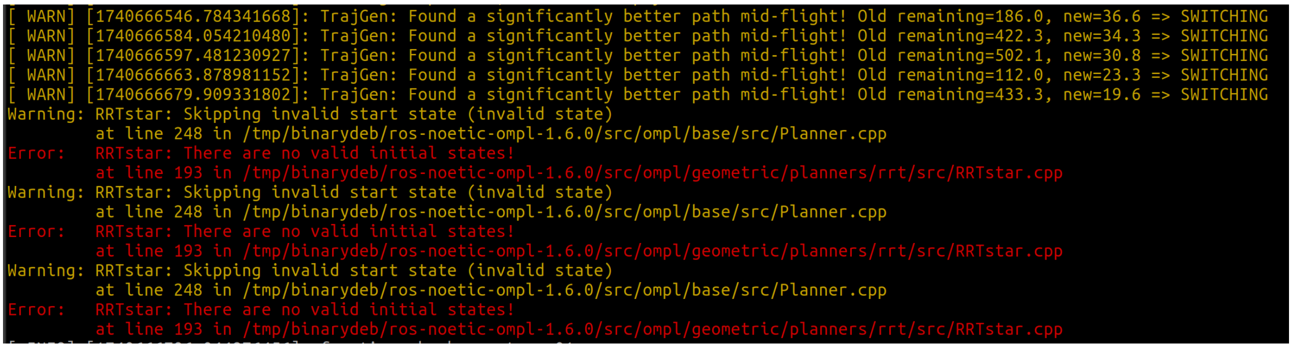


Figure 5: Warning and errors due to invalid states and new Path

11 Team Contributions

- **State Machine**: Berhan Sofuoglu
- **Takeoff Node**: Fabian Sommer
- **Perception**: Fabian Sommer
- **Frontier Detection**: Ramkrishna Chaudhari
- **Path Planning**: Ramkrishna Chaudhari
- **Trajectory Generator**: Mohamad Alattar
- **Lantern Detection**: Mohamad Alattar
- **General Testing and Finetuning**: Complete Team

Various team members handled other internal tasks, and additional support was generally provided by team members not specifically mentioned above.

References

[1] Open Source Robotics Foundation. *nodelet - ROS Wiki*. URL: http://wiki.ros.org/nodelet (visited on 26/02/2025).

[2] Open Source Robotics Foundation. *image_proc - ROS Wiki*. URL: http://wiki.ros.org/image_proc (visited on 26/02/2025).

[3] Open Source Robotics Foundation. $depth\_image_proc - ROSWiki$. URL: http://wiki.ros.org/depth_image_proc (visited on 26/02/2025).

[4] Open Source Robotics Foundation. *octomap_server - ROS Wiki*. URL: http://wiki.ros.org/octomap_server (visited on 26/02/2025).

[5] Open Source Robotics Foundation. *rviz - ROS Wiki*. URL: http://wiki.ros.org/rviz (visited on 26/02/2025).

[6] Jason Williams et al. "Online 3D Frontier-Based UGV and UAV Exploration Using Direct Point Cloud Visibility". In: *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. 2020, pp. 263–270. DOI: 10.1109/MFI49285.2020.9235268.

[7] Ana Batinovic et al. "A Multi-Resolution Frontier-Based Planner for Autonomous 3D Exploration". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4528–4535. DOI: 10.1109/LRA.2021.3068923.

[8] S. Thrun et al. "Autonomous exploration and mapping of abandoned mines". In: *IEEE Robotics Automation Magazine* 11.4 (2004), pp. 79–91. DOI: 10.1109/MRA.2004.1371614.

[9] T. Chung. *DARPA Subterranean (SubT) Challenge*. Online; accessed 2025-02-26. 2019. URL: https://www.darpa.mil/program/darpa-subterranean-challenge.

[10] LARICS. *UAV Frontier Exploration 3D*. Online; accessed 2025-02-26. 2025. URL: https://github.com/larics/uav_frontier_exploration_3d.

[11] OMPL Documentation. *OMPL: RRT* (Optimal Rapidly-exploring Random Tree)*. Online; accessed 2025-02-26. 2025. URL: https://ompl.kavrakilab.org/classompl_1_1geometric_1_1RRTstar.html#gRRTstar.

[12] OMPL Documentation. *OMPL: Geometric Motion Planners*. Online; accessed 2025-02-26. 2025. URL: https://ompl.kavrakilab.org/planners.html#geometric_planners.

[13] 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)*. 2010, pp. 5420–5425.

A Complete ROS Graph

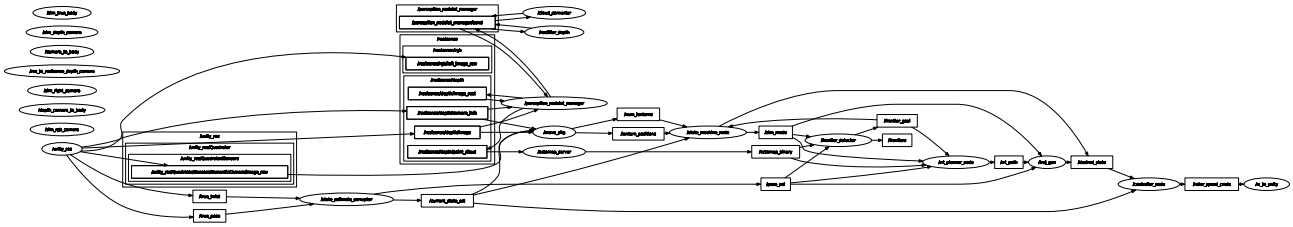


Figure 6: Complete ROS graph, as shown by `rqt_graph`.

B Complete OctoMap

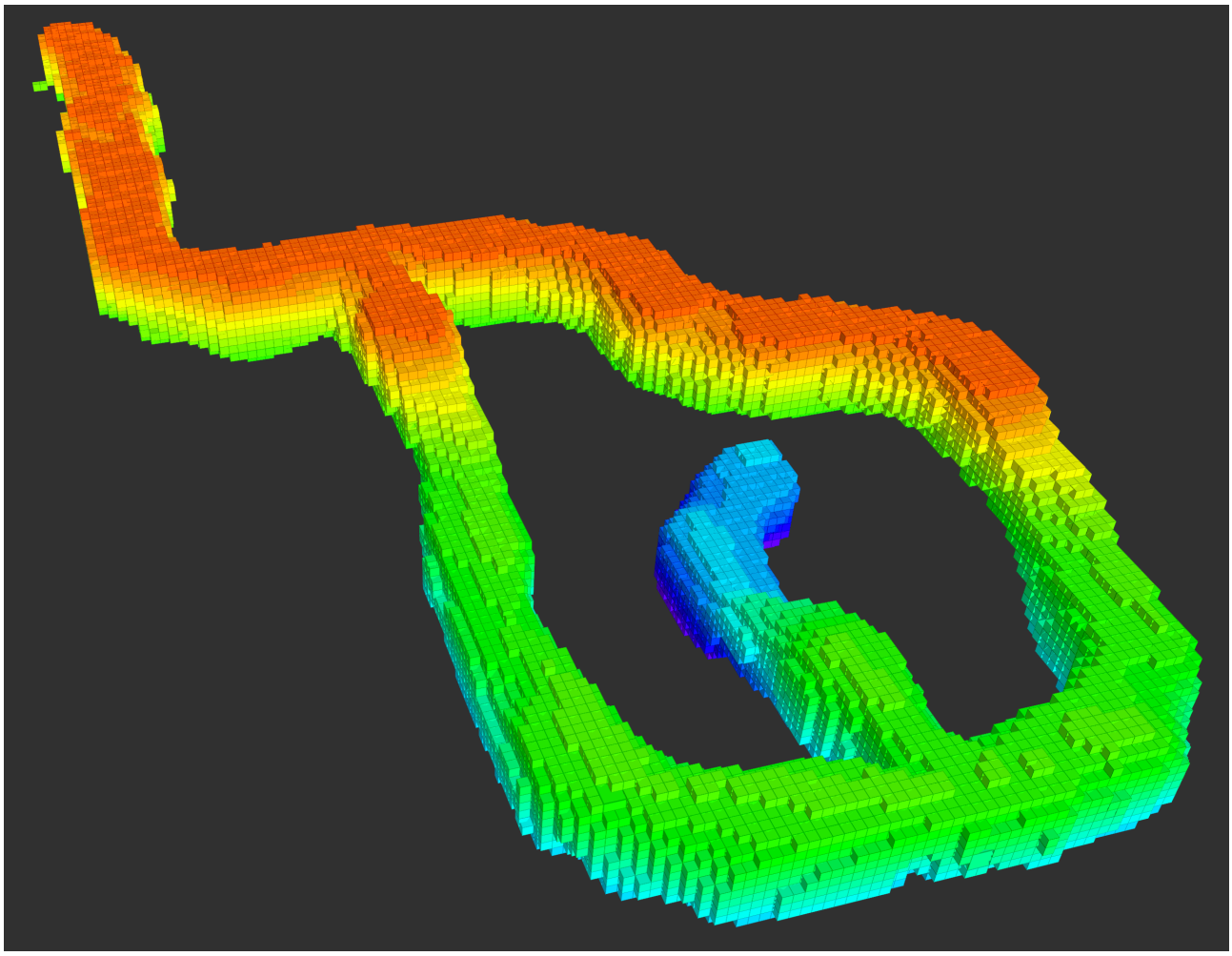


Figure 7: Complete OctoMap visualized in RVIZ