TUM 24WS Autonome Systeme: Final Project Report
Team 9

Wenjie Xie, Ziou Hu, Qianru Li, Kecheng Zhou, Dian Yu

***Abstract*—Autonomously navigating a drone through hazardous cave environments requires a streamlined integration of 3D mapping, frontier exploration, and robust trajectory control. This paper introduces a UAV system that constructs a color-enhanced voxel grid of the cave by converting raw depth images into an occupancy map using the OctoMap library. To locate unexplored regions, the system employs a cluster-based frontier selection method, ranking potential targets according to descending feasibility. For path planning, we adopt a flexible approach that allows the drone to choose among `PRM*`, `RRT*`, or Informed `RRT*`, ensuring adaptability to different cave structures. After generating a collision-free route, we leverage the `mav_trajectory_generation` package to produce a time-parameterized polynomial trajectory. Finally, an LQR controller refines the execution of the smoothened cubic spline trajectory, providing precise and stable flight maneuvers in narrow passages. Through this holistic pipeline, our UAV achieves efficient 3D frontier exploration while maintaining reliable performance in visually challenging confined spaces.**

***Index Terms*—Aerial system, Cave Exploration, 3D Path Planning, LQR Control, Perception and autonomy**

I. Introduction

Exploring cave environments with an unmanned aerial vehicle (UAV) presents two major challenges: generating a reliable internal map of unknown spaces and maintaining stable flight in confined areas. Our system addresses these challenges by first employing a color-based occupancy grid, constructed from depth-camera data, that captures both free and occupied regions. A clustering-based frontier detection algorithm then identifies voxels at the boundary of explored and unexplored areas, ranking them according to feasibility to ensure the UAV steadily expands its mapped coverage without focusing on inaccessible goals.

Once a viable frontier is selected, we employ a flexible motion-planning scheme capable of switching among `PRM*`, `RRT*`, and Informed `RRT*`, enabling the drone to adapt to varying obstacle densities and cave layouts. We then convert the resulting path into a smooth cubic-spline trajectory using a polynomial-time parameterization procedure. To execute this trajectory with precision and stability, we apply a partially revised LQR controller that refines each maneuver in real time and accommodates subtle adjustments in narrow passages. By combining color-based mapping, cluster-driven frontier exploration, and a robust planning and control strategy, our approach supports reliable autonomous navigation in visually challenging subterranean environments.

Wenjie Xie: ge63bip@mytum.de
Ziou Hu: ziou.hu@tum.de
Qianru Li: qianru.li@tum.de
Kecheng Zhou: ge53bug@mytum.de
Dian Yu: go69joc@mytum.de

II. Package overview and implementation

A. Perception Package

The Perception Package enables the UAV to percept and memorize the environment it has viewed. The key nodes are: one that transform the semantic image and depth image into point clouds of objects of interest, another that merges the point clouds into a combined representation for visualization, and a final node that detects and computes and light positions. Together, these nodes empower the UAV with the ability to merge percepted informations and locate the objects of interest.

B. Navigation Package

The Navigation Package equips the UAV with a complete exploration and motion control pipeline for cavelike environments. It comprises three key nodes: one that analyzes the drone's color-based occupancy map to discover uncharted 'frontiers', another that applies sampling-based path planning to generate a collision-free route, and a final node that converts these route waypoints into smooth trajectories with time parameterization. Together, these nodes ensure that the UAV systematically uncovers unknown regions, avoids obstacles, and maintains stable flight under velocity and acceleration constraints.

C. Controller Package

The Controller Package is dedicated to the low-level control of the UAV, tasked with executing the planned trajectory with high precision. It encompasses the Controller Node, which utilizes a Linear Quadratic Regulator (LQR) to compute optimal motor commands. By integrating the time-parameterized trajectory from the Navigation Package with real-time state feedback, the Controller Node generates the requisite thrust and torque to correct tracking errors, ensuring the UAV's stability and maneuverability in the confined and unpredictable cave environment. This package combines feedforward terms, derived from the desired trajectory, with LQR-based feedback corrections to calculate precise control inputs, which are then mapped to individual rotor commands. This approach enables robust stabilization and smooth flight maneuvers, critical for navigating narrow cave passages.

D. State Machine Package

The State Machine Package oversees the high-level mission execution of the UAV, coordinating its operational phases through a finite state machine (FSM). It includes the State Machine Node, which manages distinct flight states such as takeoff, navigation, hover, and landing. By monitoring sensor feedback and evaluating predefined trigger conditions, this node ensures seamless transitions between states, selecting and publishing the next goal as a ROS `PoseStamped` message to guide trajectory planning. This high-level control framework enables the UAV to systematically execute its cave exploration mission, adapting to real-time conditions while maintaining operational coherence.

III. Node overview and implementation

In addition to these three main packages, the functionality of our system is distributed across multiple ROS nodes, each responsible for a specific task.

A. Light Detection Node

Implemented in `light_detection.cpp`, this node projects the semantic image and depth image to 3D point clouds and then calculates the position of the light which is published to the topic `detected_points`. With a new light detection algorithm, the UAV can distinguish between old detected lights and newly detected positions of lights, avoiding miscounting of the total number for lights.

B. Waypoint Navigation Node

Located in `waypoint_navigation.cpp`, this node transforms discrete paths or final targets into smooth, time-parameterized trajectories using the `mav_trajectory_generation` library. It queries the drone's current state via odometry, sets up start and end vertices under specified velocity and acceleration limits, and solves for polynomial coefficients that minimize abrupt maneuvers. The outcome is published as a `PolynomialTrajectory4D` alongside marker arrays for visualization, ensuring that the drone maintains stable flight even in a confined cave environment.

C. State Machine Node

This node implements a finite state machine (FSM) that governs the UAV's high-level mission execution. It manages different flight states (e.g., takeoff, navigation, hover, landing) by monitoring sensor feedback and evaluating trigger conditions for state transitions. In each state, it selects the next goal, converts it into a ROS `PoseStamped` message (using a quaternion derived from the target yaw), and publishes it to guide subsequent trajectory planning.

D. Octomap Color Server Node

The Octomap Color Server Node takes the merged point cloud as input, producing a colored point cloud mapping as output. With the fine-tuning of `resolution` and `max_range` parameters, the generated `ColorOcTree` is then received and processed efficiently by navigation module.

E. Controller Node

This node refines the smooth trajectory into precise motor commands for stable flight. It receives the time-parameterized trajectory and current state feedback, computes the tracking error, and applies an LQR-based control law to generate corrective inputs. By combining feedforward terms with LQR corrections, it calculates the desired thrust and torque, which are then mapped to individual rotor commands, ensuring robust UAV stabilization in dynamic environments.

F. Planner Node

Defined in `Planner.cpp` and `Planner.h`, the Planner Node listens for new frontier goals and uses OMPL (`PRMstar`, `RRTstar`, or `InformedRRTstar`) to build a collision-free path through a voxel map. After checking that the target is feasible, it leverages the Flexible Collision Library (FCL) to perform bounding-box collision checks against the octree. If a route is found within the allotted time, the node publishes the resulting path as a `nav_msgs::Path`, optionally applying a B-spline smoothing pass for cleaner turns.

G. Exploration Node

Implemented in `frontier_exploration.cpp`, this node periodically inspects a `ColorOcTree` to determine which areas remain uncharted. It filters out unoccupied voxels in a bounding box around the drone's current position, marks those with unknown neighbors as *frontier points*, and then clusters them using OPTICS. By extracting a valid cluster centroid, it generates and publishes a single `frontier_goal`, thereby guiding the drone progressively into unmapped regions.

IV. Perception

A. Overview

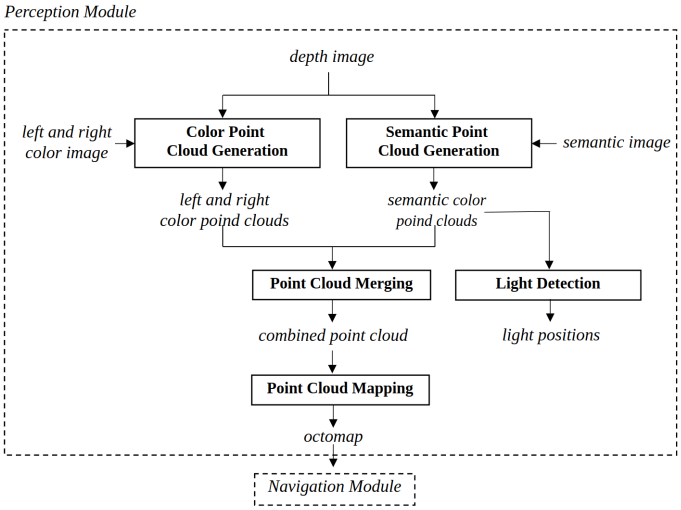

Fig. 1: Perception pipeline

In this module, the depth image, left camera color image, right camera color image and semantic camera image are used

as input; a merged point cloud mapping and detected light positions are generated as output. The detailed pipeline for perception is shown in figure 1. The module first generates two color point clouds for left camera and right camera separately using `depth_image_proc` package, then a semantic point cloud with only light points is produced from depth to point cloud using camera-to-world-frame projection. By merging these three point clouds with PCL package, a combined color point cloud is produced for visualization, indicating the position of different detected lights and the whole simulation environment. Through Octomap mapping [1], the point cloud is mapped and passed to navigation module.

B. Color Point Cloud Generation

For point cloud storage, PCL package's variable types are leveraged. For color point cloud generation, with left or right camera image, the module first aligns the depth camera frame and color camera frame utilizing `depth_image_proc/register` function, then produces the colored point cloud with `depth_image_proc/point_cloud_xyzrgb` function.

C. Semantic Point Cloud Generation

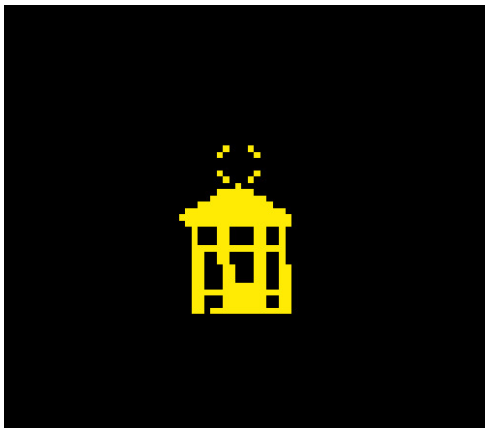

Fig. 2: Example of color image captured by semantic camera

For semantic point cloud generation, since the semantic information of a light is always with RGB value (4, 235, 255), and the remaining pixels for the semantic image are all pure black (as shown in figure 2), a masking mechanism is applied to filter the black pixels out when semantic camera image is received. After aligning the depth camera frame and semantic camera frame using `depth_image_proc/register` function, combined with depth image, the masked semantic image is then projected from 2D local image space to 3D global world space utilizing the pinhole camera coordinate transition model [2]. The pinhole camera model is formulated as follows:

$$\begin{aligned} x &= \frac{(u-c_x)}{f_x} \cdot z, \\ y &= \frac{(v-c_y)}{f_y} \cdot z \end{aligned} \tag{1}$$

where x, y, z are the world coordinates, u, v are the image plane coordinates, the focal lengths of a camera along the x and y axes are given by f_x, f_y, and the optical centers of the camera is represented by c_x and c_y respectively. The model can also be reformulated in a matrix multiplication form:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = z \cdot K^{-1} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2}$$

where

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

is the intrinsic matrix.

After conversing the semantic image into a colored semantic point cloud, the module then merges the three point clouds into a combined point cloud for visualization. This colored semantic point cloud is also used by light detection module for light position computation.

D. Point Cloud Merging

For point cloud merging [3], the module mainly uses `pcl::concatenatePointCloud` function to merge left camera color point cloud, right camera color point cloud and semantic color point cloud iteratively. The merged point cloud is then published to the system.

E. Light Detection

For light detection, the module leverages similar procedure used in IV-C to generate semantic point cloud voxels when receiving the semantic image. For the detection and the indexing of different lights scattered in the simulation environment, the module calculates the generated semantic point cloud's centroid coordinate as detected light position:

$$\text{Pos} = \frac{\sum_{i=1}^{N} P_i}{N} \tag{4}$$

where N is the total number of semantic 3D points in the point cloud, and P_i is the coordinate of one point. Since the same light can be captured by the semantic camera multiple times, the light positions calculated from these semantic images should account only for one detection. The algorithm for mitigating this situation is explained in algorithm 1.

F. Point Cloud Mapping

Point clouds offer a continuous representation of space without needing discretization, which makes mapping large areas feasible. However, they tend to consume significant memory and do not inherently classify regions as free, occupied, or undefined. Octomap addresses these limitations by efficiently generating probabilistic three-dimensional maps. `octomap_server` package is utilized for point cloud mapping. The generated Octomap [4] is then passed to navigation module for frontier exploration.

Algorithm 1: Pseudocode for light detection

Data: Detected light position `Pos`
Result: Boolean value indicating whether a new light is detected
for *each previously detected light position* `Pos_pre` **do**
  Compute the distance:
    `distance` ← MSE(`Pos`, `Pos_pre`);
  if `distance` < *threshold* **then**
    Save the new light position;
    return false;
return true;

V. Control

The control module of our UAV cave exploration system is structured into a high-level and a low-level layer. The high-level control manages mission execution and trajectory generation via a finite state machine (FSM) and a trajectory planner, while the low-level control refines these trajectories using interpolation and a Linear Quadratic Regulator (LQR) for precise motor commands. In the following sections, we detail each sub-component along with the underlying mathematical formulations.

A. High-level Control

High-level control provides mission management and smooth trajectory generation. It is divided into two parts: the *State Machine* and the *Trajectory Generation* module.

1) State Machine: The state machine implements a finite state machine (FSM) pattern to orchestrate the UAV's mission. Each mission phase is encapsulated as a separate state (e.g., `TakeoffState`, `ToCaveState`, `ForwardState`, `TurnState`, `HoverState`, `LandingState`). The FSM continuously monitors the UAV's position and orientation, and transitions between states based on pre-defined conditions (e.g., goal reached, heading error thresholds).

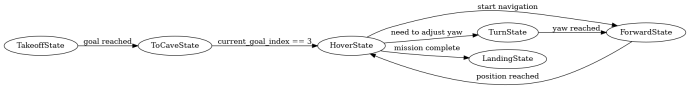

Fig. 3: State Machine layout Illustration

State Abstraction: A virtual class `StateBase` defines the interface for all states. Each derived state overrides the `execute` method, which contains the state-specific logic. In our code, the `execute` method not only performs the state's primary function but also checks for trigger conditions that determine whether a transition should occur.

Goal Management: The state machine maintains a list of goal points. In each state, it selects the next goal, converts it to a ROS `PoseStamped` message, and publishes it. The conversion involves transforming the desired yaw into a quaternion:

$$q = \left(\cos\frac{\theta}{2}, 0, 0, \sin\frac{\theta}{2}\right) \tag{5}$$

where θ is the target yaw angle.

State Transition and Trigger Mechanism: Our code incorporates explicit trigger functions that are invoked at the end of each state's `execute` routine. These trigger functions evaluate whether the UAV's current state meets the criteria to transition to the next state. Specifically, the transition condition is checked by comparing the current UAV position (x, y, z) with the target position (x_g, y_g, z_g) within a tolerance ε:

$$\begin{aligned} x &\in [x_g - \varepsilon, x_g + \varepsilon], \\ y &\in [y_g - \varepsilon, y_g + \varepsilon], \\ z &\in [z_g - \varepsilon, z_g + \varepsilon]. \end{aligned} \tag{6}$$

If the trigger condition returns true, the FSM invokes a state transition function that updates the current state pointer to the next state in the sequence. This mechanism ensures that state transitions occur deterministically based on real-time sensor feedback and pre-defined thresholds.

Mathematical Formulation: The state transition logic is essentially based on the condition:

$$\text{goal\_reached} = \mathbb{I}\left(\bigcap_{i\in\{x,y,z\}} \left|i - i_g\right| \le \varepsilon\right) \tag{7}$$

where $\mathbb{I}(\cdot)$ is the indicator function. Furthermore, the waypoint generation involves mapping a 3D point and yaw angle to a pose:

$$\textbf{pose} = (\textbf{position},\ q(\theta)) \tag{8}$$

with $q(\theta)$ computed as shown above. The additional trigger mechanisms embedded in the code rigorously enforce these rules, ensuring that the state transitions are both responsive and robust in dynamic operational conditions.

2) Trajectory Generation: The trajectory generation module converts the sequence of waypoints produced by the planner into a continuous trajectory. This trajectory is generated using a combination of PID control for error correction and cubic spline interpolation for smoothness. The resulting output is published as a ROS `MultiDOFJointTrajectory` message, which downstream controllers follow.

- **PID Control:** A classical PID controller is used to minimize the error between the current state and the desired waypoint. The control law is:

$$u(t) = K_p\, e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{9}$$

  where $e(t) = x_d(t) - x(t)$ is the tracking error.
- **Cubic Spline Interpolation:** Given a set of waypoints $\{(t_i, x_i)\}$ for each spatial coordinate, a cubic spline $s(t)$ is constructed such that:

$$s(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3, \quad t \in [t_i, t_{i+1}] \tag{10}$$

  ensuring continuity in position, velocity, and acceleration at the knots. Figure 4 illustrates the cubic spline interpolation used in our trajectory generation.

- **Temporal Parameterization:** The path is parameterized by a cumulative distance or time vector

$$s(t_i) = x_i, s'(t_i) = v_i, s''(t_i^+) = s''(t_i^-), \quad \text{for } i = 1, \ldots, n-1 \tag{11}$$

This yields the spline coefficients a_i, b_i, c_i, d_i for the piecewise polynomial. The PID control law is similarly implemented for each spatial coordinate, ensuring a robust trajectory generation even under disturbances.

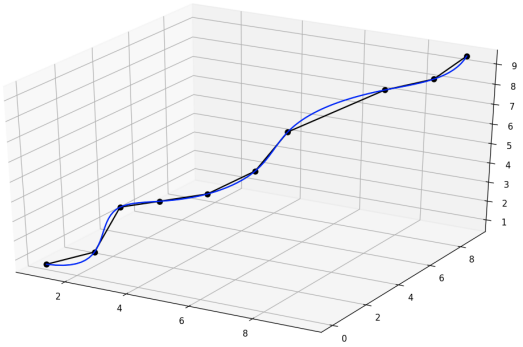

Fig. 4: Cubic spline interpolation used for trajectory generation.

B. Low-level Control

Low-level control refines the generated trajectory into precise motor commands. It consists of trajectory interpolation (for smooth motion planning) and an LQR-based controller (for optimal tracking performance).

1) Trajectory Interpolation: Trajectory interpolation further refines the coarse trajectory obtained from the high-level planner. Using cubic splines, the interpolation module generates a finely-sampled, smooth trajectory that can be accurately followed by the UAV. This process also supports end-extrapolation to ensure continuity in the case of limited waypoint data.

- **Cubic Spline Class:** The `CubicSpline` class is responsible for fitting a cubic polynomial through given waypoints. The spline is computed for each coordinate x, y, z, and the yaw angle separately.
- **Temporal Sampling:** Given the cumulative time vector $\{t_i\}$ and corresponding positions $\{x_i\}$, the interpolation generates a set of uniformly spaced time instants:

$$t_i^{\text{interp}} = t_0 + i\Delta t, \quad i = 0, 1, \ldots, N-1, \tag{12}$$

where $\Delta t = \frac{t_{\max} - t_0}{N-1}$.
- **Spline Evaluation:** For each interpolated time t_i^{interp}, the corresponding position is computed as:

$$\begin{aligned} x_{\text{interp}}(t_i^{\text{interp}}) = a_j &+ b_j(t_i^{\text{interp}} - t_j) \\ &+ c_j(t_i^{\text{interp}} - t_j)^2 \\ &+ d_j(t_i^{\text{interp}} - t_j)^3 \end{aligned} \tag{13}$$

where $t_j \le t_i^{\text{interp}} < t_{j+1}$.

Mathematical Formulation: The interpolation process is based on solving the following tridiagonal system for the second derivative coefficients c_i:

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3\left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right), \tag{14}$$

for $i = 1, \ldots, n-1$, where $h_i = t_{i+1} - t_i$. Once the c_i's are determined, the coefficients b_i and d_i are computed by:

$$\begin{aligned} b_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}), \\ d_i &= \frac{c_{i+1} - c_i}{3h_i}. \end{aligned} \tag{15}$$

This ensures that the interpolated trajectory has continuous first and second derivatives, leading to smooth motion.

2) Linear Quadratic Regulator (LQR) Control: The LQR controller is designed to minimize the error between the desired trajectory and the actual UAV state. It calculates a corrective force based on a quadratic cost function that penalizes deviations in both position and velocity. The controller then maps these corrections into rotor speed commands to achieve optimal stabilization. Figure 5 shows the block diagram of the LQR controller used in our system.

- **Error State Definition:** The state error is defined as:

$$\mathbf{e} = \begin{bmatrix} x - x_d \\ v - v_d \end{bmatrix}, \tag{16}$$

where x and v are the current position and velocity, and x_d and v_d are the desired values.
- **LQR Feedback:** The control law is computed as:

$$u_{\text{corr}} = -K_{\text{lqr}}\,\mathbf{e}, \tag{17}$$

where K_{lqr} is the gain matrix tuned to balance the trade-off between state error and control effort.
- **Force Computation:** The desired force $\mathbf{F}_{\text{des}}$ combines gravitational compensation, feedforward acceleration, and the LQR corrective term:

$$\mathbf{F}_{\text{des}} = m(g\,\mathbf{e}_3 + a_d) + u_{\text{corr}}, \tag{18}$$

with $\mathbf{e}_3 = [0, 0, 1]^T$.
- **Desired Orientation:** The desired thrust direction is determined by normalizing $\mathbf{F}_{\text{des}}$:

$$\mathbf{b}_3 = \frac{\mathbf{F}_{\text{des}}}{\|\mathbf{F}_{\text{des}}\|}. \tag{19}$$

A desired body-frame rotation R_d is then computed by aligning $\mathbf{b}_3$ with the UAV's z-axis and incorporating the desired yaw angle.
- **Attitude Error:** The rotation error is expressed as:

$$e_R = \frac{1}{2}\,\text{vee}\left(R_d^T R - R^T R_d\right), \tag{20}$$

where the vee$(\cdot)$ operator maps a skew-symmetric matrix to a vector.
- **Torque Command:** The final torque command is computed using:

$$\tau = -k_r\,e_R - k_\omega\,\omega + \omega \times (J\,\omega), \tag{21}$$

and the scalar thrust is obtained by projecting the desired force onto the current body z-axis:

$$f = \mathbf{F}_{\text{des}} \cdot (R\,\mathbf{e}_3). \tag{22}$$

- **Rotor Allocation:** An allocation matrix maps the total wrench (f, τ) into individual rotor commands. A custom function (e.g., `signed_sqrt`) ensures that the rotor speeds preserve the correct sign.

Mathematical Formulation: The overall cost function minimized by the LQR is:

$$J = \int_0^\infty \left(\mathbf{e}^T Q\mathbf{e} + u^T R u\right) dt, \tag{23}$$

where Q and R are weighting matrices. The solution yields the optimal gain matrix K_{lqr}. The corrective force then becomes:

$$u_{\text{corr}} = -K_{\text{lqr}} \begin{bmatrix} x - x_d \\ v - v_d \end{bmatrix}. \tag{24}$$

Subsequently, the control inputs are mapped to rotor speeds using the inverse of the allocation matrix F:

$$\text{rotor\_speeds} = F^{-1} \begin{bmatrix} f \\ \tau \end{bmatrix}. \tag{25}$$

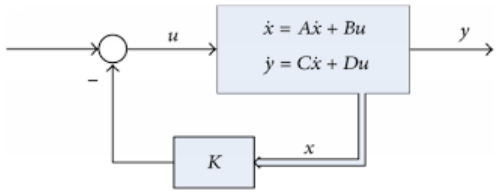

Fig. 5: Block diagram of the LQR controller.

VI. Navigation

Navigating in a GPS-denied cave environment requires continuously detecting unexplored areas, planning collision-free paths in 3D, and then executing those paths in a dynamically feasible manner. Our navigation subsystem implements these capabilities through the following:

- **Frontier Detection & Clustering:** Identifies unknown boundary regions (frontiers) on a color-based occupancy map and groups them into clusters using a density-based method [5], [6].
- **Iterative Cluster Selection:** Ensures stable exploration by testing each cluster's centroid in descending order of size until a valid goal emerges.
- **3D Path Planning:** Leverages OMPL [7] and FCL [8] to check for collisions on a voxel map, producing a safe route to the chosen frontier goal.
- **Polynomial Trajectory Generation:** Converts discrete waypoints or final poses into a smooth trajectory that respects velocity and acceleration constraints [9].

Figure 6 shows an overview of how these components interact.

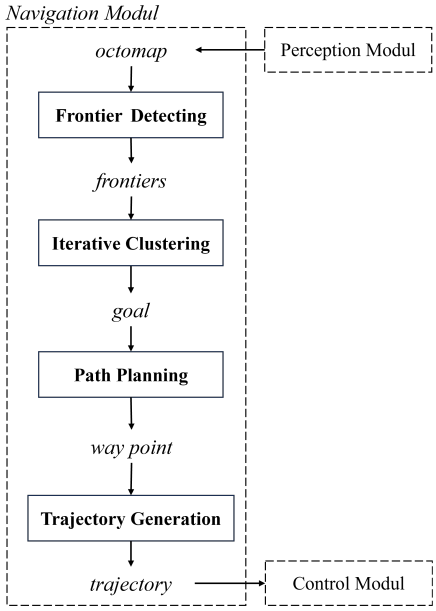

Fig. 6: Overall navigation pipeline: from frontier detection and clustering to path planning and smooth trajectory generation.

A. Frontier Detection and Clustering

- *Bounding-Box Frontier Search:* To determine promising unexplored areas in the cave, our system regularly receives a 3D color occupancy map (`ColorOcTree`) from the simulation via `octomap_full`. Within `frontier_exploration.cpp`, we begin by defining a bounding box around the current position of the UAV, using a configurable `max_range` parameter. Every unoccupied leaf node inside this box is examined; if at least one of its neighbors cannot be found in the map (`octree->search()` returns `nullptr`), that leaf node is labeled as a *frontier point*. Finally, all such frontier points are merged into a PCL point cloud to facilitate subsequent clustering.
- *Density-Based Clustering:* Because the resulting frontier point cloud can be extensive and irregular, we employ the `Optics::optics` algorithm [10] to group points according to their local density. In particular, the *reachability distance* of each point is calculated based on a user-defined radius ε and the minimum number of points (`minPts`). Points with mutually low reachability form cohesive clusters, each corresponding to a distinct frontier region. In this way, multiple frontier clusters can be identified in a single pass, laying the groundwork for iterative goal selection.
- *Iterative Cluster Selection:* Once the clusters have been extracted, we adopt a *descending-size* approach to select the most viable frontier. First, all clusters are sorted from the largest to the smallest based on their number of points. For each cluster in turn, we calculate its centroid by averaging the 3D coordinates of its constituent points, and then evaluate whether that centroid is feasible, i.e.,

whether it respects any workspace boundaries or flight constraints defined for this environment. As soon as a valid cluster is found, we publish its centroid as the new `frontier_goal` and halt our search. This step prevents the algorithm from being locked onto a single large cluster that could prove unreachable in practice. By moving on to the next largest cluster whenever the current one fails feasibility checks, the UAV steadily progresses through unexplored areas.

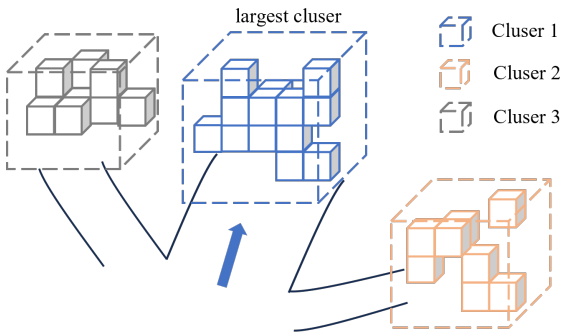

Fig. 7: Example clustering of frontier points. Each color represents one cluster. The system checks them in descending order of size to select a feasible goal.

B. 3D Path Planning with OMPL

When the `Planner` node receives a new `frontier_goal`, it first updates the drone's current position using the latest odometry data. Next, it queries OctoMap for its minimum and maximum coordinates (`getMetricMin` and `getMetricMax`), which define the feasible region in a `RealVectorStateSpace(3)` corresponding to (x, y, z). To handle collision checks within that region, the code creates a bounding-box model of the UAV, then leverages the Flexible Collision Library (FCL) to test whether each candidate sample intersects any occupied cell in the octree.

For the sampling-based planner, the user can specify one of three algorithms—`PRMstar` [11], `RRTstar` [12], or `InformedRRTstar` [13]—through a ROS parameter called `planner_method`. Once selected, the planner attempts to solve the motion-planning problem within a preset time limit (e.g., five seconds). If it discovers a valid path to the frontier goal, the resulting waypoints are converted into a `nav_msgs::Path` and published on the `planned_path` topic. To smooth out any abrupt turns, an optional B-spline refinement step (`smoothBSpline()`) may then be applied, creating a cleaner route that downstream trajectory generation nodes can follow. This approach ensures that the UAV remains within known free space while progressing toward each newly identified frontier.

C. Polynomial Trajectory Generation

Although a collision-free path describes the waypoints the UAV should follow, a smooth trajectory is essential for stable flight in a tight environment. In `waypoint_navigation.cpp`, we employ the `mav_trajectory_generation` library to convert these discrete waypoints into a continuous polynomial that respects both velocity and acceleration limits:

- *Waypoints to Polynomials:* To begin, the code treats the current position and velocity of the drone as the start vertex and uses the target position (and optional velocity) as the end vertex. Each vertex is specified with boundary conditions on position and velocity in a four-dimensional space (x, y, z, yaw). The function `estimateSegmentTimes()` then calculates an initial guess for each segment's duration, considering maximum translational speed and acceleration.
- *Minimum Snap Optimization:* After setting these vertices, a `PolynomialOptimizationNonLinear` object is configured to minimize high-order derivatives, thus generating trajectories that are easier to track and less prone to abrupt maneuvers. This step enforces the velocity and acceleration constraints by calling `addMaximumMagnitudeConstraint()` for each derivative level.
- *Trajectory Publishing:* Once the polynomial coefficients have been solved, the resulting trajectory is packaged as a `PolynomialTrajectory4D` message and published on the `Trajectory` topic. Simultaneously, the node creates a set of RViz markers (via `drawMavTrajectory()`) to visualize the path. By providing both a ROS message for flight execution and a marker array for debugging, the system enables precise online control while offering immediate insight into how the UAV's flight path evolves over time.

D. Mapping Results

Once our system completes the frontier exploration and path planning cycle, it maintains an updated color-enhanced occupancy map. In most cases, the UAV explores one branch of a cave junction and ends with a partially reconstructed environment. Figures 8 and 9 illustrate two typical outcomes in which the drone traveled in different branches.

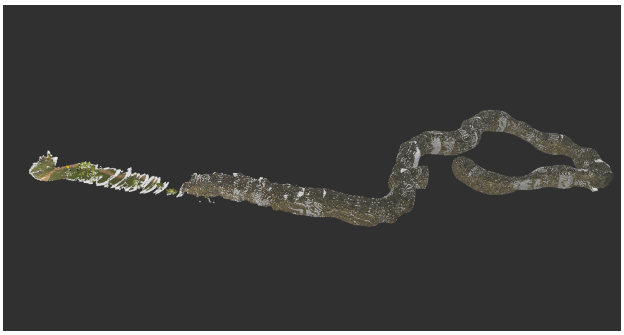

Fig. 8: Partial cave reconstruction after the UAV followed one branch.

Under ideal circumstances, if the UAV can plan a safe and collision-free route back to the junction, it will continue

Fig. 9: Another partial reconstruction showing a different cave branch.

exploring any remaining branches. This process can lead to a more complete map of the cave, as shown in Figure 10. However, modeling gaps within cave representation and uncertainties during each flight clustering often prevent such a scenario. Consequently, partial reconstructions-such as those in Figures 8 and 9—are observed more frequently.

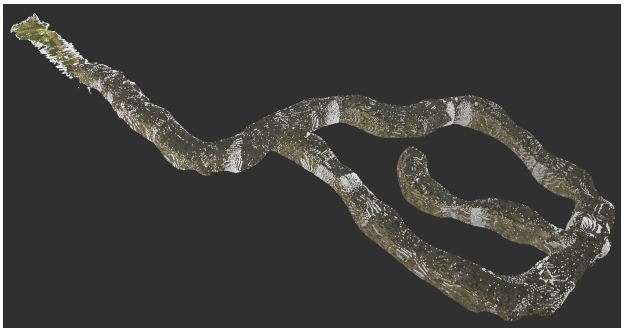

Fig. 10: Nearly complete cave mapping achieved when the UAV revisits and explores all branches.

E. Summary

In summary, our navigation approach integrates a color-based occupancy map (`ColorOcTree`) with frontier exploration, multi-cluster evaluation, sampling-based planning, and polynomial trajectory generation. Initially, the drone detects and clusters unexplored regions in the color map, iterating through potential frontiers in descending order of size to pick a feasible exploration goal. It then leverages one of several OMPL planners `PRM*`, `RRT*` or `InformedRRT*` to compute a safe path through the 3D environment, using a simple bounding-box model and FCL for collision check. Finally, these waypoints are transformed into a smooth polynomial trajectory that respects velocity and acceleration limits, enabling the UAV to fly stably while systematically mapping the cave and searching for objects of interest.

VII. Conclusion

This report presented a comprehensive approach to autonomous UAV navigation in challenging cave environments. By integrating color-based occupancy mapping, cluster-driven frontier detection, and advanced sampling-based path planning, our system effectively generates smooth, collision-free trajectories in complex 3D spaces. The use of cubic spline interpolation for trajectory refinement, combined with an LQR controller for precise motor commands, ensures stable and reliable flight even in confined areas.

Overall, the modular design of our perception, navigation, and control subsystems enables robust exploration and mapping of subterranean structures. Future work will focus on optimizing real-time performance and extending the system's adaptability to more dynamic environments, paving the way for even more advanced autonomous exploration applications.

VIII. Task Responsibility

State Machine and Control:	Dian Yu
Perception and Mapping:	Qianru Li, Wenjie Xie
Navigation and Exploration:	Kecheng Zhou, Ziou Hu

IX. Reference

References

[1] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, software available at https://octomap.github.io. [Online]. Available: https://octomap.github.io

[2] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. New York, NY, USA: Cambridge University Press, 2003.

[3] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1–4.

[4] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: an efficient probabilistic 3d mapping framework based on octrees." *Auton. Robots*, vol. 34, no. 3, pp. 189–206, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/arobots/arobots34.htmlHornungWBSB13

[5] S. S. Belavadi, R. Beri, and V. Malik, "Frontier exploration technique for 3d autonomous slam using k-means based divisive clustering," in *2017 Asia Modelling Symposium (AMS)*, 2017, pp. 95–100.

[6] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, 1997, pp. 146–151.

[7] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, https://ompl.kavrakilab.org.

[8] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 3859–3866.

[9] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2520–2525, 2011.

[10] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: ordering points to identify the clustering structure," *SIGMOD Rec.*, vol. 28, no. 2, p. 49–60, Jun. 1999. [Online]. Available: https://doi.org/10.1145/304181.304187

[11] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[12] Q. Zhou and G. Liu, "Uav path planning based on the combination of a-star algorithm and rrt-star algorithm," in *2022 IEEE International Conference on Unmanned Systems (ICUS)*, 2022, pp. 146–151.

[13] J. D. Gammell, T. D. Barfoot, and S. S. Srinivasa, "Informed sampling for asymptotically optimal path planning," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 966–984, 2018.