Sub-Terrain Challenge - Team Division Documentation

Group 3
Akhil Nedumpalli (Matr. No. 03755449
Pablo Revuelto de Miguel (Matr. No. 03800877)
Ahmed Emad Saeed Abuakr (Matr. No. 03801690)
Ammar Saleem Daredia (Matr. No. 03765727)

March 4, 2025

Contents

1 Project Overview

This document outlines our team's approach to the Sub-Terrain Challenge, where we developed an autonomous drone system for cave exploration. The project was divided among four team members, each responsible for a specific component: Perception Pipeline, Path Planning, Drone Control & Navigation, and State Machine & System Integration. This division allowed us to tackle complex subsystems in parallel while ensuring cohesive integration. The following documentation details each component's implementation, the challenges encountered, and our solutions.

2 Akhil: Perception Pipeline

2.1 Perception Pipeline Overview

The perception pipeline is implemented in the package named "perception_pipeline". The goal of the package is to use the raw depth images that are created by depth camera mounted on the drone to produce a point cloud. This is in turn used to build a voxel-grid representation and 3D visualization of the environment in which the drone operates. This information is subsequently used in the "path_planner_pkg" to help the drone identify objects in its path, aiding navigation and collision avoidance.

Within the package, two primary launch files are responsible for the perception pipeline:

- depth_to_pointcloud.launch
- pointcloud_to_octomap.launch

Both launch files are executed by a master launch file, which initiates the perception pipeline process.

2.2 Depth image to Pointcloud

The purpose of the "depth_to_pointcloud.launch" is to convert the raw depth image into a point cloud. This is achieved using the depth_image_proc package [1], which processes depth images for a variety of applications. Since it contains nodelets, it is essential to install the nodelet manager for it to function properly:

```
$ sudo apt-get install ros-noetic-nodelet-core
```

The depth_image_proc/point_cloud_xyz nodelet specifically converts the depth image into an XYZ point cloud. After launching the file through 'roslaunch perception_pipeline depth_to_pointcloud.launch', the pointcloud can be visualized in RViz by setting the global frame to 'Quadrotor/Sensors/DepthCamera' and adding a 'PointCloud2' with the topic '/realsense/depth/pointcloud_output/'.

A key issue encountered was that the depth camera image was published in the 'Quadrotor/DepthCamera' frame, while the point cloud output was published in 'Quadrotor/Sensors/DepthCamera'. To resolve this, a static transformation was applied between the two. However, full integration with the "path_planner_pkg" was not achieved due to differing node implementations.

2.3 Pointcloud to Octomap – Voxel-Grid

Once the point cloud is generated, the 'octomap_server' package is used to create a 3D occupancy grid [2]. This package can be installed using:

```
$ sudo apt-get install ros-noetic-octomap
```

The resolution and maximum range within the launch file can be adjusted to balance visualization detail and computational efficiency. Lower resolutions improve performance, while higher resolutions provide finer detail. The launch file can be executed using 'roslaunch perception_pipeline pointcloud_to_octomap.launch' and visualized in Rviz by setting the global frame once more to 'Quadrotor/Sensors/DepthCamera' and add the topics '/octomap_full' or '/occupied_cells_vis' to display the voxel grid.

3 Pablo: Path Planning

3.1 Path_Planning_Pkg

This package focuses on a general description for 3D path planning in subterranean environments (caves). The objective is to provide the autonomous drone with the ability to: Generate optimal routes between points, Smooth trajectories for fluid movement, visualize critical data in Rviz and Implement exploration strategies.

For the coding and completion of this package, the following documentation was investigated, although just for learning purposes:

- ROS Navigation Stack: navigation/Tutorials/RobotSetup - ROS Wiki
- ROS Local Path Planner: navigation/Tutorials/Writing a Local Path Planner As Plugin in ROS - ROS Wiki
- ROS Global Path Planner: navigation/Tutorials/Writing A Global Path Planner As Plugin in ROS - ROS Wiki

3.1.1 Path_planner_node.cpp

General Description: ROS node implementing the A* algorithm in 3D for route planning in cave environments.

Key Features:

- Input /cave_grid (nav_msgs/OccupancyGrid): 3D map of the cave.
- Output /planned_path (nav_msgs/Path): Calculated optimal route.
- Parameters: grid_resolution (m/cell), inflation_radius (safety radius).
- Algorithm: 3D A* with Euclidean heuristic and 26-directional neighborhood management.

For the implementation of the algorithm and for building purposes, DeepSeek-R1 was used, while for optimizations of the code Chatgpt o3-mini was used.

3.1.2 Trajectory_generator_node.cpp

General Description: ROS node that generates smooth trajectories from discrete waypoints.

Main Function: To convert a planned route (series of points) into a physically viable trajectory with continuous velocity/acceleration profiles.

Looking at the Key Features, we have:

- Input /planned_path (nav_msgs/Path): Route with waypoints from the path planner.
- Output /trajectory (trajectory_msgs/MultiDOFJointTrajectory): Smoothed trajectory Parameters: max_velocity (maximum velocity), max_acceleration (maximum acceleration), spline_order (spline order).
- Algorithm: Eigen cubic splines for spatial interpolation + Trapezoidal velocity profile.

For the implementation of the algorithm, DeepSeek-R1 was used, for coding and optimization of the code.

3.1.3 Path_visualization_node.cpp

General Description: ROS node that generates interactive visual representations of the drone's trajectory in RViz.

Main Purpose: To convert trajectory data into visual markers for: System debugging, Real-time monitoring and Post-mission analysis.

Key Features:

- Input /trajectory (trajectory_msgs/MultiDOFJointTrajectory): Generated trajectory.
- Output /path_markers (visualization_msgs/MarkerArray): RViz markers.

- Visual Elements: Continuous line (trajectory) + Arrows (velocity) Color Encoding Red = High velocity, Green = Low velocity.

For the integration with this node with the other ones of the path_planning_pkg, DeepSeek-R1 was used, for coding, optimization and built of the code.

3.2 Exploration_Pkg

General Description: ROS package for autonomous exploration of unknown environments (caves) using a frontier-based exploration strategy.

3.2.1 Exploration_node.cpp

The exploration_node.cpp is part of the exploration_pkg package and is designed to manage autonomous exploration in a cave environment.

Main Objective: To search for objects of interest (e.g., light sources) using a strategy based on the identification of "frontiers" (boundary areas between the known and the unknown).

Key Features:

- Map Processing (Octomap): The node receives information about the environment in the form of an octomap (3D map) via the /octomap_binary topic.
- Frontier Detection: Traverse the leaf nodes of the octomap to identify those that are frontiers. Subsequently, they are grouped into clusters using Point Cloud Library (PCL) clustering extraction.
- Utility Calculation for Frontiers: Each cluster is evaluated based on various criteria: Cluster size, Distance from the current position and Distance from the entry.
- Exploration Goal Planning: Based on the utility of the frontiers, the node selects the most promising goal and publishes it to the /path_goal topic, after verifying its accessibility through a service call (/check_goal_reachable).
- Detection and Tracking of Objects of Interest: The node subscribes to detections from a semantic camera (/semantic_camera/detections topic). It groups detections based on proximity, and if a detection is repeated several times within a given time interval (detection_time_window), it is marked as confirmed, and the object is then investigated.
- Mission State Management: Several mission states are defined: INITIALIZING, EXPLORING, INVESTIGATING_OBJECT, RETURNING_HOME, COMPLETED, and FAILED.
- Visualization: Markers are published to be visualized in RViz: frontiers and detected objects.

For the coding of this node and its integration with the package.xml and CMakeLists.txt, DeepSeek-R1 was used.

4 Ahmed: Drone Control & Navigation

for this part the task was divided into two packages: 1. Navigation, handling take-off and landing. 2. Trajectory, taking the path from the path planing and converting it to a desired state.

4.1 Navigation_Pkg

This package focuses on high-level navigation commands for the drone, including: waypoint, navigation, take-off and landing procedures.

Mainly contained in two primary nodes: the navigation controller and the landing controller.

4.1.1 Navigation_controller.cpp

General Description: ROS node that handles waypoint-based navigation for the drone.
Key Features:

- Input /current_state_est (nav_msgs/Odometry): Current drone position and orientation.
- Input /planned_path (geometry_msgs/PoseStamped): Target waypoints for navigation.
- Output /desired_state (trajectory_msgs/MultiDOFJointTrajectoryPoint): Commands for the drone.
- Parameters: waypoint_reached_threshold (0.5m), max_velocity (2.0m/s).

Manages waypoints and directs the drone to move toward the current target waypoint. When a waypoint is reached (based on the waypoint_reached_threshold), it commands to the next waypoint in the queue to be reached.

The controller calculates appropriate velocity commands based on the direction and distance to the target waypoint.

4.1.2 Landing_controller.cpp

General Description: ROS node that manages takeoff and landing procedures.
Key Features:

- Input /current_state_est (nav_msgs/Odometry): Current drone altitude and position.
- Input /takeoff_command (std_msgs/Empty): Signal to initiate takeoff.
- Output /desired_state (trajectory_msgs/MultiDOFJointTrajectoryPoint): Vertical movement commands.
- Output /landing_controller/state (std_msgs/String): Current state of landing controller.
- States: IDLE, TAKEOFF, HOVER, LANDING, LANDED.

The landing controller handling the vertical movement of the drone during takeoff and landing. During take-off, it commands the drone to ascend to a specified height (takeoff_height parameter) and then transition to a hovering state. The controller publishes its current state to allow other components of the system to coordinate their actions with the takeoff and landing.

4.2 Trajectory_Tracking_Pkg

This package contains a trajectory tracker node that converts high-level trajectory commands into desired states for the drone controller.

4.2.1 Trajectory_tracker.cpp

General Description: ROS node that tracks and executes multi-DOF trajectories.
Key Features:

- Input /trajectory (trajectory_msgs/MultiDOFJointTrajectory): Trajectory to follow.
- Input /current_state_est (nav_msgs/Odometry): Current drone state.
- Output /desired_state (trajectory_msgs/MultiDOFJointTrajectoryPoint): Desired state for controller.
- Parameters: tracking_frequency (50.0Hz), look_ahead_time (0.2s), timeout_duration (5.0s).

The trajectory tracker implements interpolation between trajectory points. The tracker also includes timeout detection to handle cases where trajectory execution takes longer than expected.

4.3 System Integration

The navigation and trajectory tracking components are integrated with the rest of the system through a navigation.launch file:

- Navigation Controller: Configured with waypoint_reached_threshold and max_velocity parameters.
- Landing Controller: Configured with takeoff_height, hover_height, takeoff/landing velocities, and various thresholds.
- Trajectory Tracker: Configured with tracking_frequency, look_ahead_time, and timeout_duration parameters.

The ROS topic structure allows for seamless communication between these components and the rest of the system:

- /current_state_est → provides drone state to all navigation components
- /planned_path → provides waypoints from the path planner to the navigation controller
- /trajectory → provides trajectory from the path planner to the trajectory tracker
- /desired_state → provides setpoints from navigation components to the drone controller

5 Ammar: State Machine & System Integration

5.1 Responsibilities

- Develop the main state machine for the robot
- Create the main launch file and system configuration
- Set up ROS message interfaces between components
- Handle reporting and recording of object detections
- Ensure proper system startup and shutdown

5.2 Key packages and files

- state_machine_pkg
  - state_machine_node.cpp
  - master_launch.launch

5.3 State Machine Documentation

The state_machine_pkg is responsible for high-level commands for navigating through cave exploration system autonomously. This includes managing the operational states of the drone and ensuring smooth transitions between key flight phases such as takeoff, navigation, and landing. The state machine reacts to real-time feedback from other subsystems (perception, path planning, trajectory tracking, and navigation) and drives the mission forward.

5.3.1 System Flow

IDLE → TAKEOFF → NAVIGATE → LAND → IDLE

1. System Startup: All nodes launch, including state machine, perception pipeline, path planner, trajectory tracker, and controllers.
2. Idle State: State machine waits for /takeoff_complete signal.
3. Takeoff Phase: When the takeoff signal is received, the state transitions.
4. Navigation Phase: Drone navigates using planned paths from the planner.

5. Landing Phase: After reaching its goal or receiving an external land command, the drone transitions to landing.

6. Back to Idle: After successfully landing, the state returns to IDLE, awaiting the next mission.

5.4 Master_launch.launch file

This launch file is the central launch file that launches the state machine node and all other launch files, including those for simulation, perception, path planning and navigation. This is launched using the command:

```
roslaunch state_machine_pkg master_launch.launch
```

References

[1] depth_image_proc: ROS package for processing depth images, including conversion to point clouds. ROS Wiki: `http://wiki.ros.org/depth_image_proc`.

[2] OctoMap: A probabilistic 3D mapping framework using occupancy grids. Official site: `https://octomap.github.io/`.