Autonomous Systems:
Group Project: Sub-Terrain Challange

Bingkun Huang, Haowen Shi, Siyan Li, Weili Tang, Zhenjiang Li

1 Introduction

This project is part of the Sub-Terrain Challenge in the Autonomous Systems course at TUM. The objective is to develop a system that can autonomously explore a cave environment, detect and locate four objects of interest (lights), and generate a 3D voxel-grid or mesh representation of the environment. The implementation involves a ROS-based framework, integrating perception, path planning, and control using a quadrotor and a Unity-based simulation. This document provides an overview of the system architecture, software components, team contributions, challenges, and results, including a ROS graph, figures.

2 Project Goals

The project aims to develop an autonomous system capable of exploring a cave environment, detecting and locating four objects of interest (lights) as quickly as possible. Additionally, the system must generate a 3D representation of the environment using either a voxel-grid or a mesh-based approach. Key objectives include:

Implementing a perception pipeline to process depth images and generate point clouds. Developing path and trajectory planning algorithms for autonomous navigation. Ensuring seamless ROS integration via a simulation bridge for real-time communication. Designing a state machine for robot operation, handling take-off, navigation, and landing.

3 System Overview

3.1 Eigen Catkin

Eigen catkin is a wrapper for the Eigen linear algebra library in ROS (Robot Operating System), ensuring seamless integration within the catkin build system.

In UAV (Unmanned Aerial Vehicle) navigation and control, many critical data structures rely on Eigen, including:

- Waypoints: Stored as Eigen::Vector3d (x, y, z) or Eigen::MatrixXd for multiple waypoints.
- Current Position: UAV position, velocity, and acceleration are represented using Eigen::Vector3d or Eigen::Affine3d.
- Trajectory Optimization: Uses Eigen::MatrixXd for storing trajectory points and performing optimization computations.

By providing a standardized Eigen version, eigen catkin prevents dependency conflicts and ensures stable, efficient matrix operations in UAV applications within the ROS ecosystem.

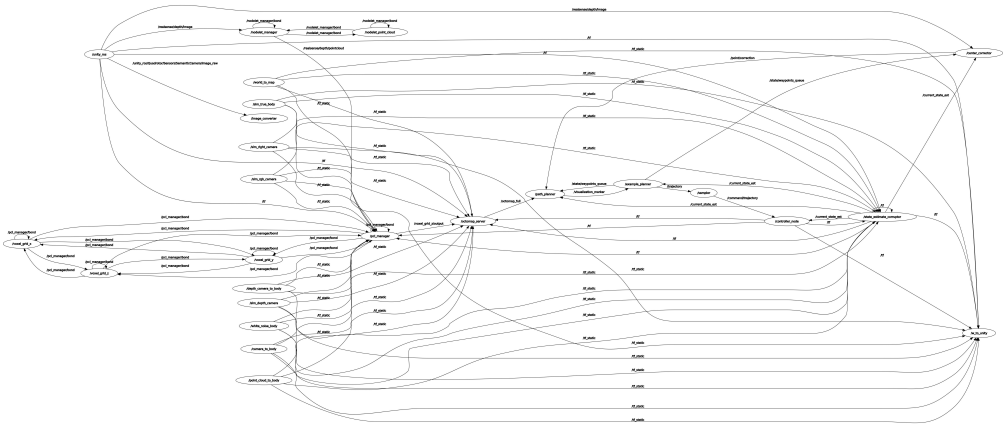

Figure 1: ROS figure of the current project

3.2 Catkin Simple

Catkin simple is a lightweight CMake wrapper designed to simplify the build process for ROS (Robot Operating System) catkin packages. It reduces boilerplate CMake code, making package management easier and more readable. Key Features:

- Simplifies CMakeLists.txt: Reduces the complexity of manually handling dependencies, libraries, and executables.
- Easy Library and Executable Creation: Provides concise functions like cs add library() and cs add executable(), replacing long CMake commands.
- Automatic Dependency Management: Automatically links required ROS and system dependencies, reducing manual setup.
- Better Build System Integration: Works seamlessly with catkin make, catkin build, and catkin tools.

3.3 ROS Noetic OctoMap

ROS Noetic OctoMap is the ROS Noetic integration of OctoMap [1], enabling 3D octree-based mapping and environment representation for robot navigation, UAV obstacle avoidance, and path planning. It builds probabilistic 3D occupancy voixel grids from sensor data (i.e. the depth frame image generated by stereo camera system) and lightweight storage with adaptive resolution. In ROS Noetic, octomap server can be used to publish 3D maps.

3.4 Mav Trajectory Generation

The mav trajectory generation [2] package provides tools for polynomial trajectory generation and optimization, specifically designed for micro aerial vehicles, such as quadrotors. It implements both linear and nonlinear trajectory optimization methods based on minimum snap or minimum jerk approaches, ensuring smooth, dynamically feasible paths for MAV navigation.

The package supports waypoint-based trajectory generation, where users define key positions, and the optimizer calculates continuous polynomial paths. It allows time allocation optimization, ensuring efficient speed profiles. Additionally, it integrates feasibility checks for velocity, acceleration, and thrust constraints to prevent actuator saturation.

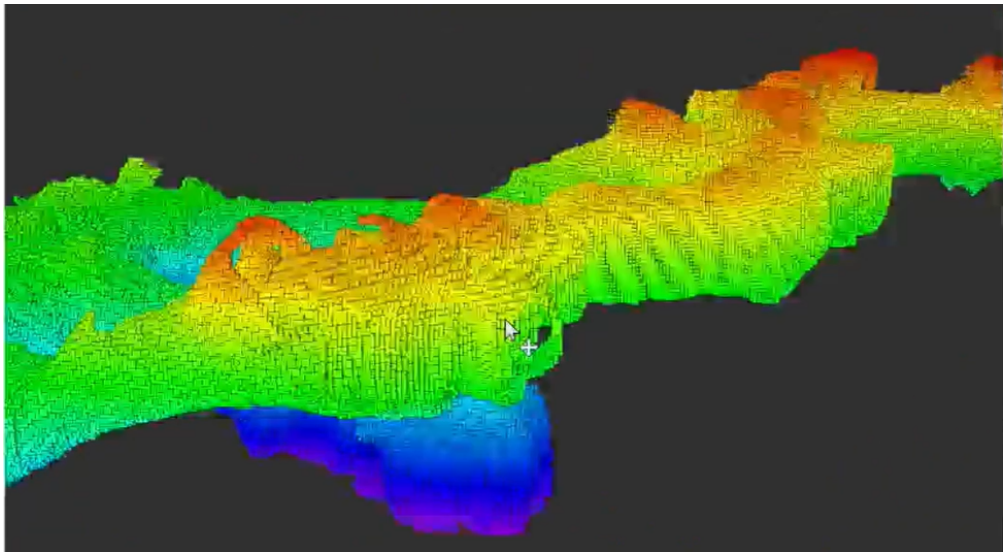

Figure 2: An example of octomap of the current project map

3.5 Flexible Collision Library

The Flexible Collision Library [3] (FCL) is an efficient and versatile collision detection and distance computation library, primarily used in robotics, physics simulation, and computer graphics. It provides a flexible interface supporting rigid body collision detection, nearest point computation, distance queries, and continuous collision detection. FCL utilizes Bounding Volume Hierarchies (BVH) to accelerate collision detection and supports various geometric representations, such as triangle meshes, convex hulls, AABBs, OBBs, and RSS. Widely used in ROS (Robot Operating System) and other robotic simulation frameworks, FCL is capable of handling complex environment modeling and collision detection tasks.

4 Implementation

1. Unity Environment

Unity serves as the simulation platform for the entire environment, responsible for simulating realistic physical conditions and providing sensor data to the UAV, including RGB camera and depth camera images. During the simulation, Unity continuously updates the UAV's attitude, position, and camera perspective in real time, ensuring that the sensor data accurately reflects the UAV's movement within the environment. Additionally, Unity communicates through ROS (Robot Operating System) nodes, allowing external programs to access UAV state information, sensor data, and environmental feedback. This integration enables developers to utilize ROS to subscribe to key sensor data, perform navigation, path planning, and perception tasks, while also sending control commands to Unity for closed-loop control and high-precision simulation. This architecture not only enhances the realism and interactivity of the simulation but also provides an efficient and flexible development environment for research and testing of autonomous UAV systems.

2. Perception

The perception system of the UAV consists of multiple sensors that provide essential data for navigation, mapping, and obstacle avoidance. The system includes an inertial measurement unit (IMU), which measures acceleration and angular velocity to estimate the UAV's attitude and movement. Additionally, the UAV is equipped with two RGB cameras, positioned to capture stereoscopic vision, enabling depth estimation and scene reconstruction. A depth camera further enhances environmental perception by providing precise distance measurements to objects in the surroundings. These sensors work together to enable autonomous

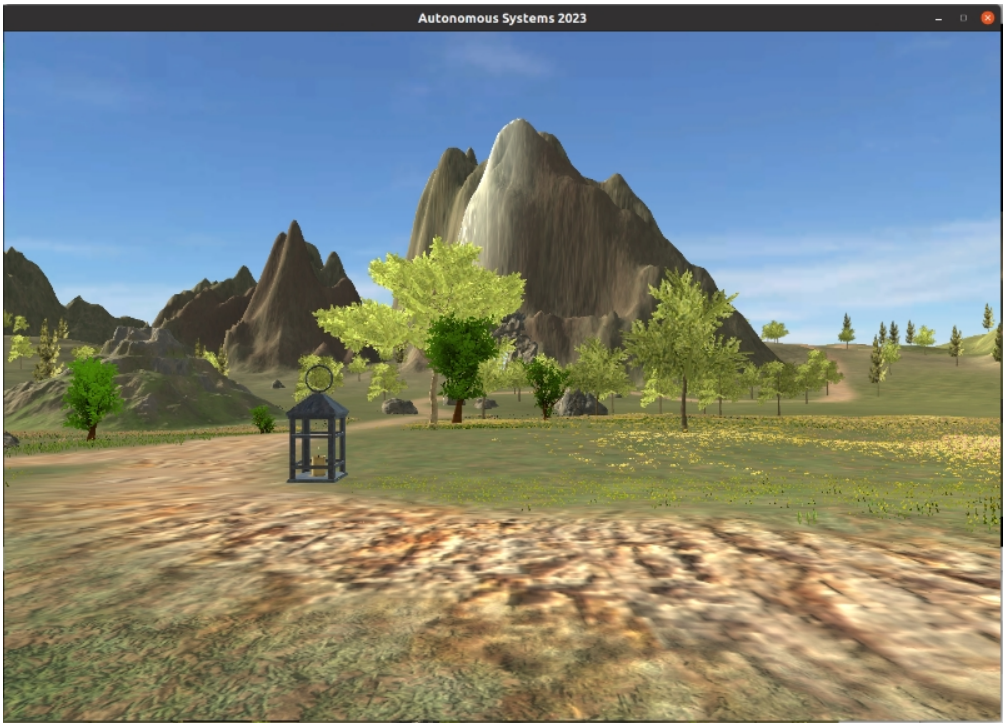

Figure 3: An example of Unity environment of the current project

decision-making and interaction with the simulated environment in Unity, allowing real-time feedback and control through ROS integration.

Here are some example code used in the project, showing how the information from unity is percepted:

```
    ImageConverter()
    : it_(nh_)
{
    // Subscrive to input video feed and publish output video feed
    sem_img_sub_ = it_.subscribe(
        "/unity_ros/Quadrotor/Sensors/SemanticCamera/image_raw", 1,
      &ImageConverter::onSemImg, this);

    depth_img_sub_ = it_.subscribe("/realsense/depth/image", 1,
      &ImageConverter::onDepthImg, this);

    depth_info_sub_ = nh_.subscribe("realsense/depth/camera_info", 1,
    &ImageConverter::onDepthInfo, this);

    point_pub_ = nh_.advertise<geometry_msgs::PointStamped>(
        "point/latern", 10);

}
```

3. From depth image to Octomap

This ROS launch file processes depth data from a RealSense camera to generate a 3D occupancy map (OctoMap). The pipeline begins with two nodelet managers: nodelet_manager, a general-purpose manager, and pcl_manager, specifically for Point Cloud Library (PCL) nodelets. The point cloud generation step uses

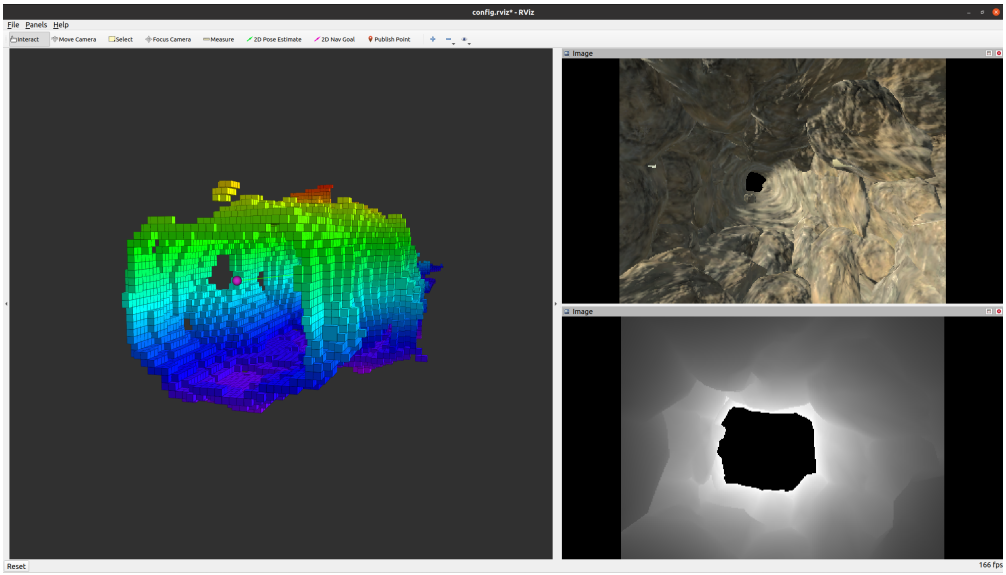

Figure 4: An example of octomap of the current project map

the depth_image_proc/point_cloud_xyz nodelet to convert a depth image into a 3D point cloud, taking as input:

```
/realsense/depth/image  # Depth image input
/realsense/depth/camera_info  # Camera info input
```

and producing:

```
/realsense/depth/pointcloud  # Point cloud output
```

Next, VoxelGrid filtering is applied sequentially along the X, Y, and Z axes to remove NaN values and downsample the cloud to reduce computational load. Each filter is configured with:

```
filter_field_name: X | Y | Z  # Axis to filter
filter_limit_min: <value>  # Minimum valid range
filter_limit_max: <value>  # Maximum valid range
leaf_size: <value>  # Downsampling resolution
```

The filtered point cloud is then processed by the octomap_server node to create the 3D OctoMap, using parameters:

```
resolution: 1.5  # OctoMap resolution in meters
frame_id: map  # Reference frame
sensor_model/max_range: 50  # Maximum sensor range in meters
```

The input to octomap_server is:

```
/voxel_grid_z/output  # Filtered point cloud
```

4. mini-batch A* specialized for the cave

We uses the A* (A-star) search algorithm [4] for path planning and combines it with OctoMap for 3D obstacle avoidance. The whole process can be divided into the following steps:

- Subscribe to sensor data

We use ROS to subscribe to OctoMap (3D occupancy map) and target points, which are used for path finding:

/octomap_full (from octomap_server): Gets the 3D map of the current environment for collision detection. /point/correction (from FrontierDetecter): Gets the target point from the depth camera to guide the UAV forward.

- Preprocess the OctoMap

  When receiving OctoMap data, octomapCallback() parses the 3D map and stores it into the octree variable:

```
void octomapCallback(const octomap_msgs::Octomap::ConstPtr& msg) {
    std::lock_guard<std::mutex> lock(map_mutex);
    delete octree;
    octree = dynamic_cast<octomap::OcTree*>(
        octomap_msgs::fullMsgToMap(*msg));
    resolution = octree->getResolution();
}
```

- Receive target point When a goal point (the location the UAV needs to go to) is received, goalCallback() processes the point:

```
void goalCallback(const geometry
_msgs::PointStamped::ConstPtr& msg) {
    int gx = std::round(msg->point.x / resolution);
    int gy = std::round(msg->point.y / resolution);
    int gz = std::round(msg->point.z / resolution);
    findNearestFree(gx, gy, gz);

    int sx, sy, sz;
    {
        std::lock_guard<std::mutex> lock(map_mutex);
        octomap::point3d start =
        octree->keyToCoord(octree->coordToKey(
            msg->point.x, msg->point.y, msg->point.z));
        sx = std::round(start.x() / resolution);
        sy = std::round(start.y() / resolution);
        sz = std::round(start.z() / resolution);
        findNearestFree(sx, sy, sz);
    }

    auto path = aStarSearch(sx, sy, sz, gx, gy, gz);
    if (path.empty()) {
        ROS_WARN("A* failed to find a path!");
        return;
    }

    nav_msgs::Path path_msg;
    path_msg.header.frame_id = "map";
    path_msg.poses = path;
    path_pub.publish(path_msg);
    vis_pub.publish(path_msg);
    ROS_INFO("Path published with %ld points.", path.size());
}
```

- A* search The A* algorithm is implemented in aStarSearch(), which is used to find the optimal path from the starting point to the goal point:

```
std::vector<geometry_msgs::PoseStamped> aStarSearch(
    int sx, int sy, int sz, int gx, int gy, int gz) {
std::priority_queue<Node*, std::vector<Node*>,
 CompareNode> openList;
std::unordered_map<std::string, Node*> allNodes;

Node* startNode = new Node(sx, sy, sz);
Node* goalNode = new Node(gx, gy, gz);
openList.push(startNode);
allNodes[hashKey(sx, sy, sz)] = startNode;

while (!openList.empty()) {
    Node* current = openList.top();
    openList.pop();

    if (current->isGoal(goalNode))
     return reconstructPath(current);

    for (auto& shift : shifts) {
        int nx = current->x + shift[0];
        int ny = current->y + shift[1];
        int nz = current->z + shift[2];

        if (isCollision(nx, ny, nz)) continue;

        Node* neighbor = new Node(nx, ny, nz, current);
        neighbor->g = current->g + 1;
        neighbor->h = std::sqrt(pow(nx - gx, 2)
         + pow(ny - gy, 2) + pow(nz - gz, 2));

        std::string key = hashKey(nx, ny, nz);
        if (allNodes.find(key) == allNodes.end()
         || neighbor->g < allNodes[key]->g) {
            allNodes[key] = neighbor;
            openList.push(neighbor);
        }
    }
}
return {};
}
```

- Calculate the path and publish When A* finds a path, it calls reconstructPath() to convert the path to the ROS-compatible nav_msgs::Path format:

```
std::vector<geometry_msgs::PoseStamped>
    reconstructPath(Node* node) {
    std::vector<geometry_msgs::PoseStamped> path;
    while (node) {
        geometry_msgs::PoseStamped pose;
        pose.pose.position.x = node->x * resolution;
        pose.pose.position.y = node->y * resolution;
```

```
            pose.pose.position.z = node->z * resolution;
            path.push_back(pose);
            node = node->parent;
        }
        std::reverse(path.begin(), path.end());
        return path;
    }
```

5. Trajectory Generation

The TrajectoryPublisher class initializes with default values for linear and angular scales, setting the trajectory duration to 5 seconds. It sets up publishers and subscribers for the desired state, keyboard input, and current pose. The main loop runs at 50Hz, and if the system is initialized (is_initialized_ == true), it generates a trajectory, publishes the desired state, and broadcasts the TF transform. The keyboard input callback updates goal_pose_ based on user input and resets the trajectory start time:

```
void keyboardCallback(const geometry_msgs::Pose& msg) {
    goal_pose_ = msg;
    trajectory_start_time_ = ros::Time::now();
}
```

The current pose callback initializes desired_pose_ and goal_pose_ with the current pose if the system is not initialized and continuously updates current_pose_ from odometry data:

```
void currentPoseCallback(const nav_msgs::Odometry::ConstPtr& msg) {
if (!is_initialized_) {
    desired_pose_ = msg->pose.pose;
    goal_pose_ = msg->pose.pose;
    is_initialized_ = true;
}
current_pose_ = msg->pose.pose;
}
```

The trajectory generation function computes a smooth 3rd-order polynomial trajectory from current_pose_ to goal_pose_, determining the intermediate desired state (desired_pose_) based on elapsed time:

```
void generateTrajectory() {
double t = (ros::Time::now() - trajectory_start_time_).toSec();
if (t > trajectory_duration_) {
    desired_pose_ = goal_pose_;
} else {
    double alpha = t / trajectory_duration_;
    desired_pose_.position.x = (1 - alpha)
     * current_pose_.position.x + alpha * goal_pose_.position.x;
    desired_pose_.position.y = (1 - alpha)
     * current_pose_.position.y + alpha * goal_pose_.position.y;
    desired_pose_.position.z = (1 - alpha)
     * current_pose_.position.z + alpha * goal_pose_.position.z;
}
}
```

The desired state is then published as a MultiDOFJointTrajectory message:

```
void publishDesiredState() {
    trajectory_msgs::MultiDOFJointTrajectory msg;
    msg.points.resize(1);
```

```
        msg.points[0].transforms[0].translation.x = desired_pose_.position.x;
        msg.points[0].transforms[0].translation.y = desired_pose_.position.y;
        msg.points[0].transforms[0].translation.z = desired_pose_.position.z;
        desired_state_pub_.publish(msg);
    }
```

Finally, the broadcast transform function publishes desired_pose_ as a TF transform with the frame ID "av-desired":

```
    void broadcastTransform() {
    static tf::TransformBroadcaster br;
    tf::Transform transform;
    transform.setOrigin(tf::Vector3(desired_pose_.position.x,
     desired_pose_.position.y,
     desired_pose_.position.z));
    transform.setRotation(tf::Quaternion(desired_pose_.orientation.x,
     desired_pose_.orientation.y, desired_pose_.orientation.z,
      desired_pose_.orientation.w));
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
     "world", "av-desired"));
}
```

5 Results

Figure 5: An example of target rendered in octomap while the UAV is flying

6 Contributions

- Bingkun Huang: path planning, system integration.
- Haowen Shi: UAV control module.
- Siyan Li: trajectory generation, documentation.
- Weili Tang: semantic camera, perception.
- Zhenjiang Li: Octomap, build system, manual control.

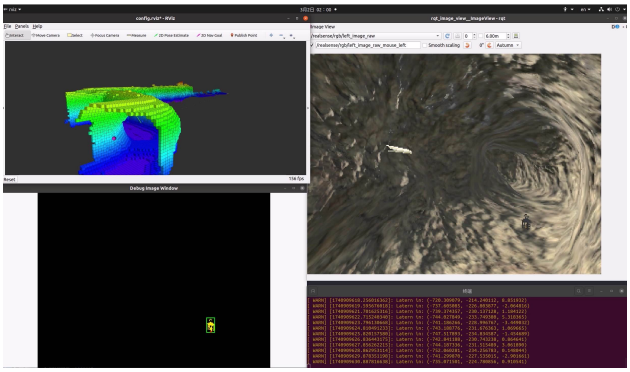

Figure 6: An example of target detected and showed in semantic camera, while the UAV is flying

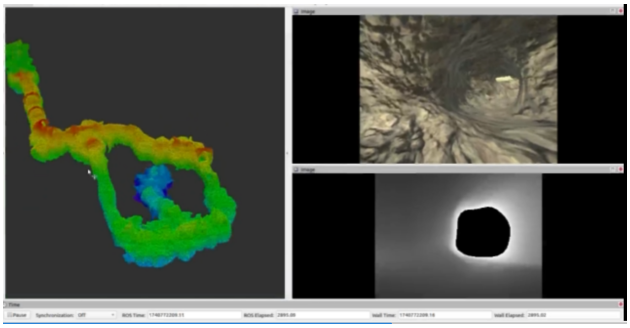

Figure 7: A demo GUI showing the full map is detected and rendered

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] C. Richter, A. Bry, and N. Roy, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in *Robotics Research*. Springer, 2016, pp. 649–666.

[3] 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.

[4] 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.