College of Engineering (COE)

National Chung Cheng University

Report of TEEP@ASIAPLUS2025 Intern Research Program

Student	Ayush Kumar	**Faculty Member**	Dr. Pao-Ann Hsiung
ID Number	113410151	**Gender**	Male
University/Department		Department of Computer Science and Information Engineering	
Research Period		2024/12/16 ~ 2025/06/16	
Date of Report Submission		2025/06/06	
Report Title	AI-Based Smart flight control for UAV Systems		
Highlights of Report	1. Created new **Gazebo-Classic Environment** and performed **UAV Arming** and **Offboard Mode Activation** using **MAVROS.** 2. **Fault-Tolerant Offboard Control (FTC)** Designed a robust FTC system to handle Offboard communication loss and MAV-Link timeouts with fallback behaviours like **AUTO.LAND**/ **RTL**. 3. **Flood Detection with Deep Learning** o **Image classification** using **ResNet-18** achieved **96% accuracy** on a hybrid dataset comprising real-world images and custom Gazebo simulation data. o **Semantic Segmentation** using **U-Net** with **99% accuracy** on a custom Gazebo world dataset. 4. **Real-time GPS Mapping and target localization from Image-Based Flood Detection** 5. **Semantic Segmentation of Real UAV Imagery** Applied **DeepLabV3+ with MobileNetV3** on aerial images from physical UAV, achieving **89.26% accuracy** in real-world flood segmentation.		

Signature of Report	**Ayush Kumar**	**Date**	2025/06/06
Mentor Review Comment	□ Excellent □Good □Average □Poor		
Signature of Mentor		**Date**	

AI-Based Smart Flight Control for UAV Systems

Ayush Kumar
Intern, Research Centre on AI and Sustainability
National Chung Cheng University, Taiwan

ABSTRACT

Floods remain one of the most destructive and frequent natural disasters globally, necessitating rapid, intelligent, and autonomous monitoring solutions. This project presents a comprehensive and modular UAV-based flood detection and navigation framework that integrates artificial intelligence, robotics, and geospatial mapping. The architecture is designed for both simulated and real-world environments, enabling end-to-end autonomous flood analysis, localization, and mission execution using the PX4 flight stack, ROS 2, MAVROS, and AI-based vision models.

In the simulation phase, the UAV operates in the PX4-SITL and Gazebo environments. A robust *Fault-Tolerant Offboard Control (FTC)* system is implemented to ensure mission safety during MAVLink communication failures or Offboard mode drops. The FTC system includes a *Fault Detection Node* that continuously monitors the UAV's heartbeat and mode status, a *Fallback Handler Node* that triggers emergency responses such as Return-to-Launch (RTL) or Auto-Land, and a *Watchdog Node* that detects MAVROS communication loss in real time. These modules provide fail-safe capabilities essential for autonomous UAV operations.

A dedicated flood detection pipeline processes real-time UAV camera feeds using two deep learning models: *ResNet-18*, achieving 96% accuracy for image classification, and *U-Net*, reaching 99% accuracy for semantic segmentation on a custom Gazebo flood-world dataset. The `flood_segmentation_inference.py` ROS 2 node identifies the most flooded area by dividing the segmentation mask into a 4×4 grid and selecting the region with the highest flood pixel density. The center pixel of this region is projected into 3D space using inverse camera intrinsics and extrinsics, and the corresponding ground coordinate is transformed into GPS format using MAVROS localization data and the WGS84 geodesic model. These GPS coordinates are then published to the `/next_gps_waypoint` topic.

To autonomously navigate toward flood-affected areas, the `autonomous_gps_navigation.py` ROS 2 node consumes the GPS waypoints and generates setpoints for the UAV in real time. This node supports autonomous takeoff, maintains consistent mission altitude relative to AMSL, queues and tracks multiple waypoints, and ensures a smooth transition to RTL upon mission completion. The entire framework is validated through successful autonomous missions in Gazebo, demonstrating smooth arming, Offboard activation, navigation, waypoint tracking, and emergency fallbacks.

For real-world validation, the architecture is deployed on a physical drone platform consisting of a Pixhawk flight controller and a Jetson Nano companion computer. All ROS 2 nodes originally developed in simulation are successfully migrated and executed onboard the UAV. A demonstration mission was carried out using *Mission Planner*, where the UAV followed a pre-programmed sequence: `TAKEOFF` $\rightarrow$ multiple `WAYPOINTS` $\rightarrow$ `RETURN TO LAUNCH (RTL)`, controlled via MAVLink, replicating the simulated mission behavior.

Furthermore, to evaluate the real-world performance of the perception system, semantic segmentation was performed on actual UAV-captured aerial imagery using a *DeepLabV3+* model with a *MobileNetV3* backbone, achieving an accuracy of 89.26%. This confirms the effective transferability of the AI models from synthetic simulation environments to noisy, real-world scenarios, thus validating the practical applicability of the system.

This project successfully demonstrates a scalable, intelligent, and robust UAV platform for autonomous flood detection and response missions, bridging the gap between simulation and real-world field deployment.

Contents

List of Figures

List of Tables

1 System Setup Instructions

- **Operating System:** Ubuntu 22.04 or compatible Linux distribution.

1.1 Update Locale

To avoid locale-related issues, run the following commands to set up your locale:

```
sudo apt update
sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

1.2 Add ROS 2 Package Repository

You need to add the ROS 2 package repository to your system. Follow these commands:

```
sudo apt update && sudo apt install -y software-properties-common
sudo add-apt-repository universe
sudo apt update && sudo apt install curl -y
```

1.3 Install ROS 2 Humble Desktop

Install the ROS 2 Humble desktop package by running the following command:

```
sudo apt update && sudo apt install ros-humble-desktop
```

1.4 Source ROS 2 Environment

To source the ROS 2 environment in every terminal, use:

```
source /opt/ros/humble/setup.bash
```

To make this permanent, add it to your `/.bashrc` file:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

1.5 Install Gazebo Simulator

Install the Gazebo simulator using:

```
sudo apt update && sudo apt install gazebo
```

1.6 Install ROS 2 Gazebo Plugins

To interface Gazebo with ROS 2, install the necessary plugins:

```
sudo apt install ros-humble-gazebo-ros-pkgs
```

1.7 Verify Installation

To verify if Gazebo has been installed correctly, run:

```
gazebo
```

This should launch the Gazebo simulator window. If successful, you're ready for the next steps.

1.8 Clone PX4 Autopilot Repository

PX4 Autopilot is essential for controlling your UAV. To clone it, follow these steps:

- **Navigate to your workspace:**

```
cd ~/TEEP/src
```

- **Clone the PX4 Autopilot repository:**

```
git clone --recursive https://github.com/PX4/PX4-Autopilot.git
```

- **Install PX4 dependencies:** After cloning the repository, navigate to the PX4-Autopilot directory and install necessary dependencies:

```
cd PX4-Autopilot
make px4_sitl gazebo
```

- **Build PX4:** Build the PX4 autopilot firmware for simulation:

```
make px4_sitl gazebo
```

This will compile PX4 for simulation using Gazebo. Now you have PX4 set up to control the UAV.
Once PX4 is installed, you can access various models and worlds in the simulation:

`cd ~/TEEP/src/PX4-Autopilot/Tools/simulation/gazebo-classic`
`/sitl_gazebo-classic`

We can find or add new models/worlds in the `models` and `worlds` directories within this path.

2 Integration of Camera Sensors within the model

To add a camera sensor to your UAV model in Gazebo, you can modify the "iris" UAV model and change it according to your requirements. The UAV model will be configured for either a vertical downward camera view or a horizontal camera view.

2.1 Steps to Integrate Camera

1. Modify UAV SDF Description

To integrate the camera into the UAV model, open the SDF file for your UAV model and locate the `<model>` section that defines your drone.

Navigate to the PX4 source directory:

```
cd  /TEEP/src/PX4-Autopilot
```

2.1.1 2. Select Camera View Type

Depending on your application, you can select either a vertical downward camera view or a horizontal camera view.

For the downward-facing camera, compile the model by running:

```
make px4_sitl gazebo-classic_iris_downward_depth_camera
```

For the horizontal camera view, run:

```
make px4_sitl gazebo-classic_iris_depth_camera
```

2.1.2 3. Modify the SDF File

After selecting the correct camera view and compiling it, modify the UAV's SDF file to add the camera sensor.

Locate the `<model>` section for your UAV inside the SDF file. Define a camera sensor within the `<link>` section of the model. Below is an example of how to add a camera sensor:

```
<sensor name="camera" type="depth">
    <camera>
        <horizontal_fov>1.396</horizontal_fov>
        <image>
            <width>640</width>
            <height>480</height>
            <format>R8G8B8</format>
        </image>
        <clip>
            <near>0.1</near>
            <far>100.0</far>
        </clip>
        <pose>0 0 0 0 0 0</pose>
    </camera>
</sensor>
```

In the `<pose>` tag, adjust the camera position and orientation. For a downward view, ensure the camera is positioned and oriented properly relative to the UAV body.

2.1.3 4. Launch the Simulation

Once the model has been rebuilt, launch the Gazebo environment to see the UAV with the integrated camera.

Start MAVROS (for PX4 communication):

```
ros2 run mavros mavros_node
```

Launch PX4 SITL (Software in the Loop simulation):

```
make px4 sitl gazebo-classic iris downward depth camera
```

2.1.4 5. Check Camera Output

To verify that the camera is working properly, list the available topics:

```
ros2 topic list
```

To inspect the messages being published on this camera topic:

```
ros2 topic echo /camera/image_raw
```

3 Launching the Simulation with our Desired World and Model

3.1 World Configuration

For simulating flood conditions, a custom world file named `flood_detection.world` was created to reflect the desired environment. By default, PX4 SITL Gazebo uses the `empty.world` file, which does not include any specific simulation elements.

To ensure our custom flood simulation environment loads by default, we manually replaced the content of `empty.world` with that of `flood_detection.world`. As a result, upon launching the simulation, the system loads the flood simulation environment, though the file name remains `empty.world`. This allows us to maintain compatibility while incorporating the desired features.

3.2 Model Configuration

The default UAV model in PX4 SITL is the `iris`. To enable first-person view (FPV) capabilities for flood monitoring, we replaced the SDF content of the default `iris` model with that from `iris_fpv_cam.sdf`.

This change was made in the models directory:

```
~/TEEP/src/PX4-Autopilot/Tools/simulation/gazebo-classic/sitl_gazebo-
classic/models
```

3.3 Launching the Simulation

With the modified world and model configurations in place, navigate to the PX4 root directory and launch the simulation using the following commands:

```
cd ~/TEEP/src/PX4-Autopilot
make px4_sitl gazebo-classic
```

This will now launch the desired world with the desired model, enabling a complete simulation setup for flood detection using a UAV equipped with an FPV camera.

4 MAVROS and PX4 Connection

4.1 Connecting MAVROS to PX4

Launch the MAVROS node with the following command to establish the connection:

```
ros2 launch mavros px4.launch fcu_url:=
"udp://:14540@127.0.0.1:14557"
```

The MAVROS state should indicate `connected: true`.

4.2 Sending Position Setpoints

Before setting the UAV to Offboard mode, it is important to continuously publish position setpoints. This ensures the flight controller accepts the mode transition. Use the following command to start publishing setpoints at 10 Hz:

To send position setpoints, use the `setpoint_position/local` topic. For example, to move the drone to coordinates (5, 5, 10), use the following command:

```
ros2 topic pub -r 10 /mavros/setpoint_position/local
geometry_msgs/msg/PoseStamped "{header: {frame_id: ''},
pose: {position: {x: 5.0, y: 5.0, z: 10.0}, orientation:
{x: 0.0, y: 0.0, z: 0.0, w: 1.0}}}"
```

4.3 Arming the UAV and Setting Offboard Mode

Arm the drone and set it to Offboard mode.

4.3.1 Arming the UAV

To arm the UAV, use the following command:

```
ros2 service call /mavros/cmd/arming mavros_msgs/srv/CommandBool
"{value: true}"
```

4.3.2 Setting Offboard Mode

To set the drone to Offboard mode, use the following command:

```
ros2 service call /mavros/set_mode mavros_msgs/srv/SetMode
"{base_mode: 0, custom_mode: 'OFFBOARD'}"
```

4.4 Monitoring UAV's Position

To monitor the position of the drone, subscribe to the `/mavros/local_position/pose` topic using the following command:

```
ros2 topic echo /mavros/local_position/pose
```

4.5 Landing and Return-to-Launch Commands

To land the UAV at its current location:

```
ros2 service call /mavros/set_mode mavros_msgs/srv/SetMode
"{base_mode: 0, custom_mode: 'AUTO.LAND'}"
```

To make the UAV return to its launch (home) position:

```
ros2 service call /mavros/set_mode mavros_msgs/srv/SetMode
"{base_mode: 0, custom_mode: 'AUTO.RTL'}"
```

5 Safe Landing in PX4 SITL within Gazebo-Classic

In PX4 SITL simulations using Gazebo-Classic, drones may exhibit unsafe landing behavior such as free-falling or bouncing upon touchdown when executing the `commander land` command. Additionally, during takeoff, the drone may fail to lift off and instead auto-disarm immediately after issuing the `commander takeoff` command. These issues often stem from improper parameter settings or missing sensor simulations, and addressing them is essential for achieving smooth, controlled landings.

5.1 Common Causes

- Descent velocity parameters too low.
- Missing simulated downward rangefinder.
- Takeoff altitude below threshold.
- Faulty preflight checks or bad parameter settings.

5.2 Recommended PX4 Parameters

Table 1: Recommended PX4 Parameters for Safe Landings

Parameter	Recommended Value / Description
`MPC_LAND_SPEED`	Landing descent speed: `0.5 m/s`
`MPC_Z_VEL_MAX_DN`	Max downward velocity: `1.5 m/s`
`LNDMC_Z_VEL_MAX`	Landing velocity cap: `1.5 m/s`
`MIS_TAKEOFF_ALT`	Minimum takeoff altitude: `5.0 m`

Commands:

```
param set MPC_LAND_SPEED 0.5
param set MPC_Z_VEL_MAX_DN 1.5
param set LNDMC_Z_VEL_MAX 1.5
param set MIS_TAKEOFF_ALT 5.0
```

5.3 Verification Steps

Run checks and commands in pxh shell to validate:

System Check:

```
commander check
```

Flight Test:

```
commander takeoff
commander land
```

Ensure controlled ascent and gentle landing.

6 Fault-Tolerant Offboard Control (FTC) for Offboard Communication Loss and MAVLink Timeouts

This system ensures safe operation of UAVs during critical failures, such as loss of Offboard mode or MAVLink communication timeouts.

6.1 Features

- Monitors MAVROS heartbeat
- Detects Offboard mode failure or MAVLink timeout
- Automatically switches to RTL (Return to Launch) for safety

6.2 System Architecture

ROS 2 Nodes

- **Fault Detection Node**: Continuously monitors heartbeat and `set_mode` status.
- **Fallback Handler Node**: Sends fallback commands such as RTL or AUTO.LAND to PX4.

PX4 Autopilot

- Handles mode management, `set_mode` commands, and interprets MAVROS heartbeat.
- Executes fallback behavior (e.g., RTL) if triggered by the ROS 2 system.

6.3 Modifying Fallback Behavior

To change the fallback response from AUTO.LAND to AUTO.RTL, modify the following in your FTC node script:

File: `~/TEEP/src/ftc_node/ftc_node/ftc_node.py`

Locate inside `execute_fallback()`**:**

```
req.custom_mode = 'AUTO.LAND'
```

Change it to:

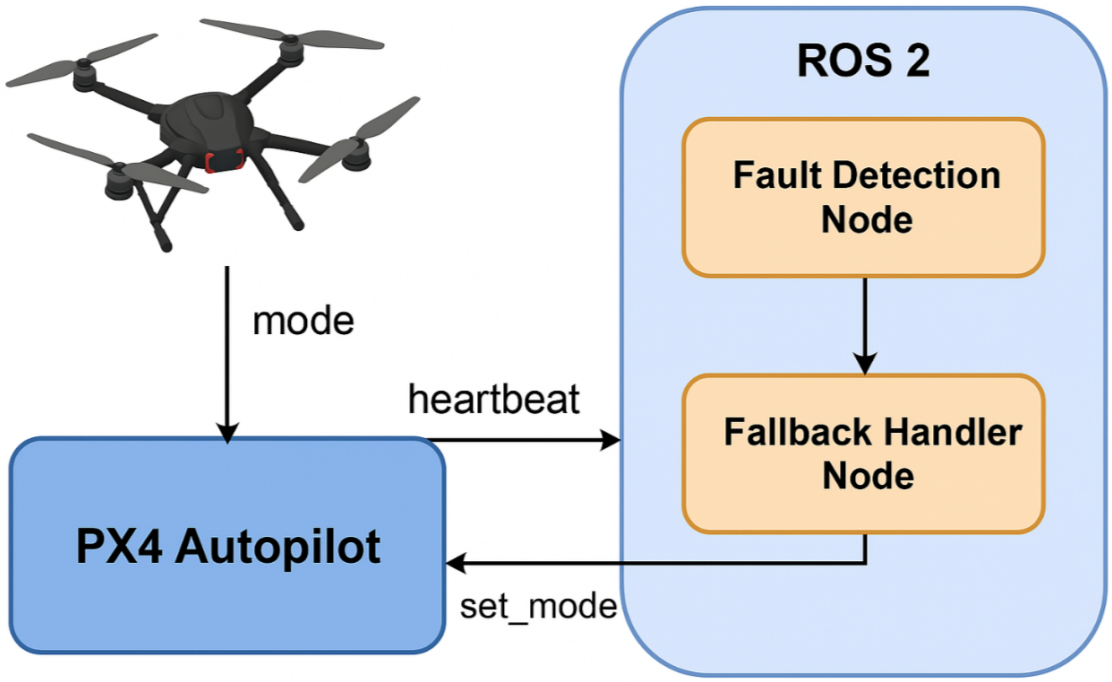


Figure 1: Control Architecture

```
req.custom_mode = 'AUTO.RTL'
```

Rebuild the ROS 2 workspace:

```
cd ~/TEEP
colcon build --packages-select ftc_node
```

6.4 Running the FTC Node

After building, run the FTC node with:

```
ros2 run ftc_node ftc_node
```

6.5 Verifying the System

1. Inject a Fault by Killing MAVROS:

```
pkill -9 ros2
```

2. Manually Change Flight Mode from OFFBOARD to POSCTL:

```
ros2 service call /mavros/set_mode mavros_msgs/srv/SetMode
"{base_mode: 0, custom_mode: 'POSCTL'}"
```

Upon detection, the fallback handler will issue an `AUTO.RTL` command, and the UAV will safely return to its launch point.

6.6 Watchdog Node for MAVROS Communication Loss

To maintain a stable communication link between PX4 and ROS 2, especially during simulations involving Gazebo, a watchdog mechanism is deployed. The Watchdog Node (‘watchdog_node‘) is a ROS 2 utility designed to ensure that the ‘mavros‘ process remains active throughout the simulation.

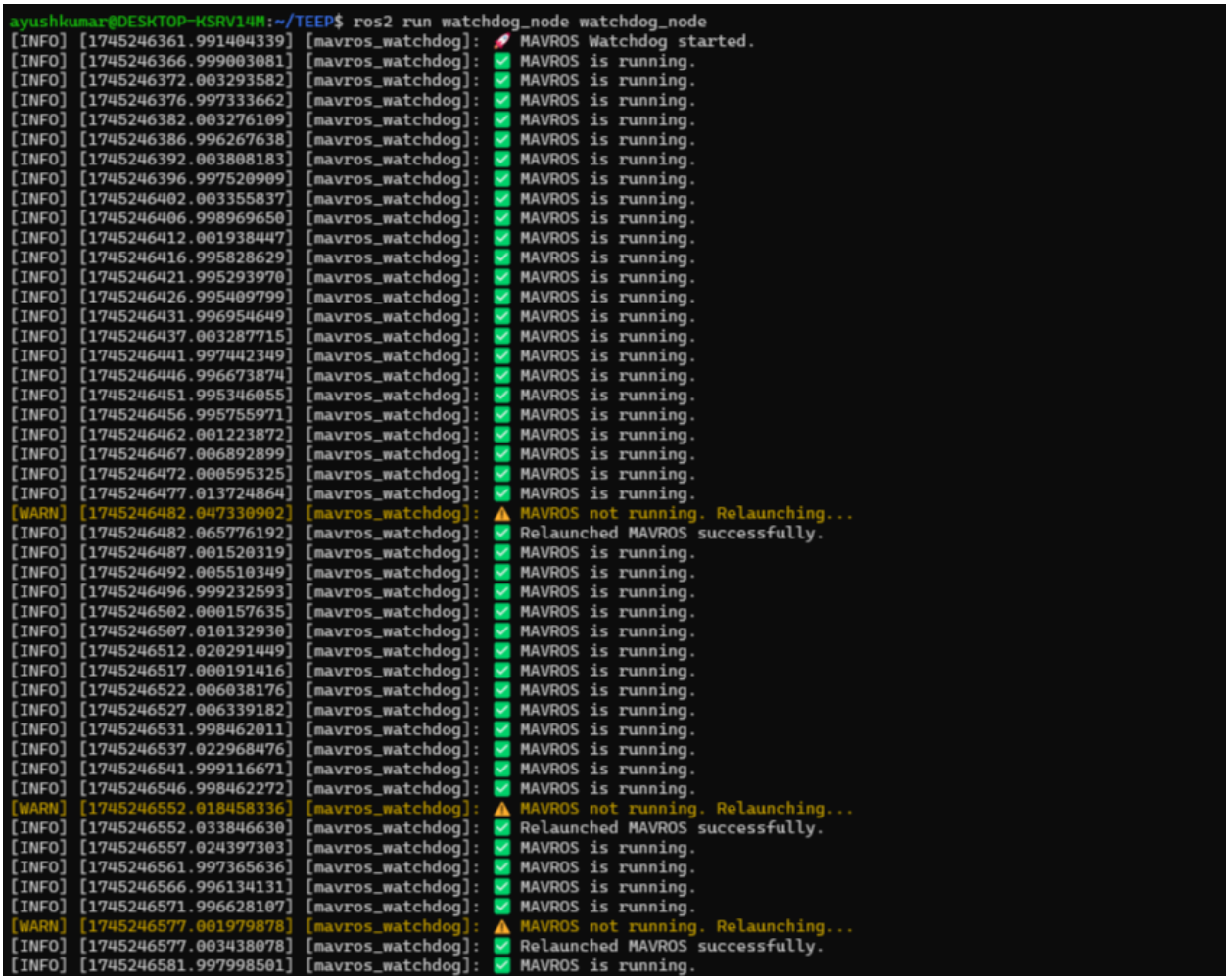


Figure 2: Watchdog Node

- It performs periodic health checks every 1 second.
- If the ‘mavros‘ node is found to be terminated or unresponsive, the watchdog automatically relaunches it.

6.6.1 Launching the Watchdog Node

```
ros2 run watchdog_node watchdog_node
```

This command should be executed before launching ‘mavros‘ to ensure monitoring begins early.

6.6.2 Handling Simulation Clock Synchronization

To prevent simulation clock mismatches that can lead to warnings such as “Time jumped backwards“, ensure the watchdog uses Gazebo’s simulation time:

```
ros2 param set /mavros_watchdog use_sim_time true
```

This sets the ‘use_sim_time‘ parameter to ‘true‘, enabling synchronization with Gazebo’s simulated clock and avoiding timing-related issues.

7 Viewing Camera Feed by OpenCV and Rviz2

This section describes how to visualize the camera feed from the UAV for flood detection in the Gazebo simulation.

7.1 ROS 2 Image Publisher Node

To view the camera feed, a ROS 2 node is required to subscribe to the Gazebo camera topic and potentially process the images for viewing with OpenCV or for further tasks such as flood detection.

7.1.1 OpenCV

To use OpenCV with ROS 2, follow these steps:

- **Install OpenCV:** Run the following command to install OpenCV and its dependencies:

```
sudo apt install libopencv-dev python3-opencv
```

- **Install CvBridge:** CvBridge is used to convert ROS 2 image messages to OpenCV images. Install it by running:

```
sudo apt install ros-humble-cv-bridge
```

- **Create a Python Node:** The Python node subscribes to the `/camera/image_raw` topic, converts the ROS image messages to OpenCV format, and displays the image.

7.1.2 Rviz2

Rviz2 allows visualization of the camera feed in a GUI format. To view the feed in Rviz2, follow these steps:

- **Launch Rviz2:** Start Rviz2 with the following command:

```
rviz2
```

- **Add the Camera Feed:** In the Rviz2 window:
  1. Click the "Add" button in the left panel.
  2. Select the "Image" display type.
  3. Set the "Image Topic" property to `/camera/image_raw`.
  4. You should now see the live camera feed from the UAV in the Rviz2 window.

8 Flood Detection with Deep Learning

Once the camera feed is visible, we process the images and classify them as "Flooded" or "Non-Flooded" using a deep learning model (ResNet18). This is done by creating a Python script that subscribes to the camera feed, preprocesses the images, and passes them through the trained model for inference.

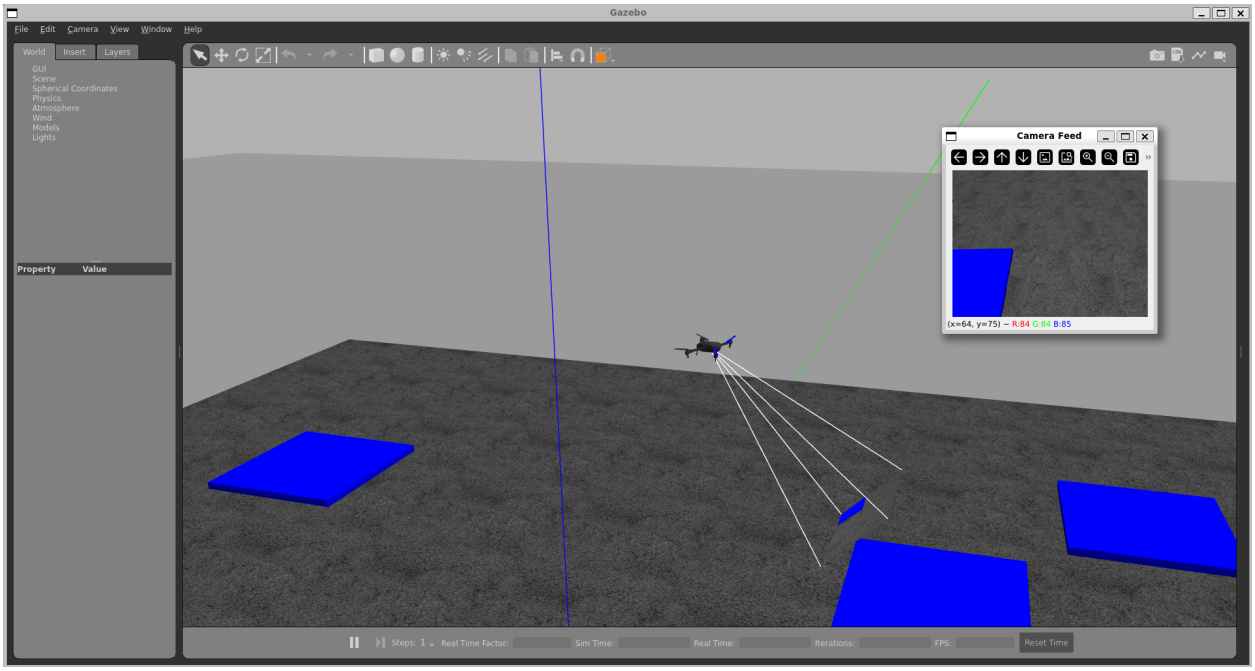


Figure 3: Gazebo Simulation

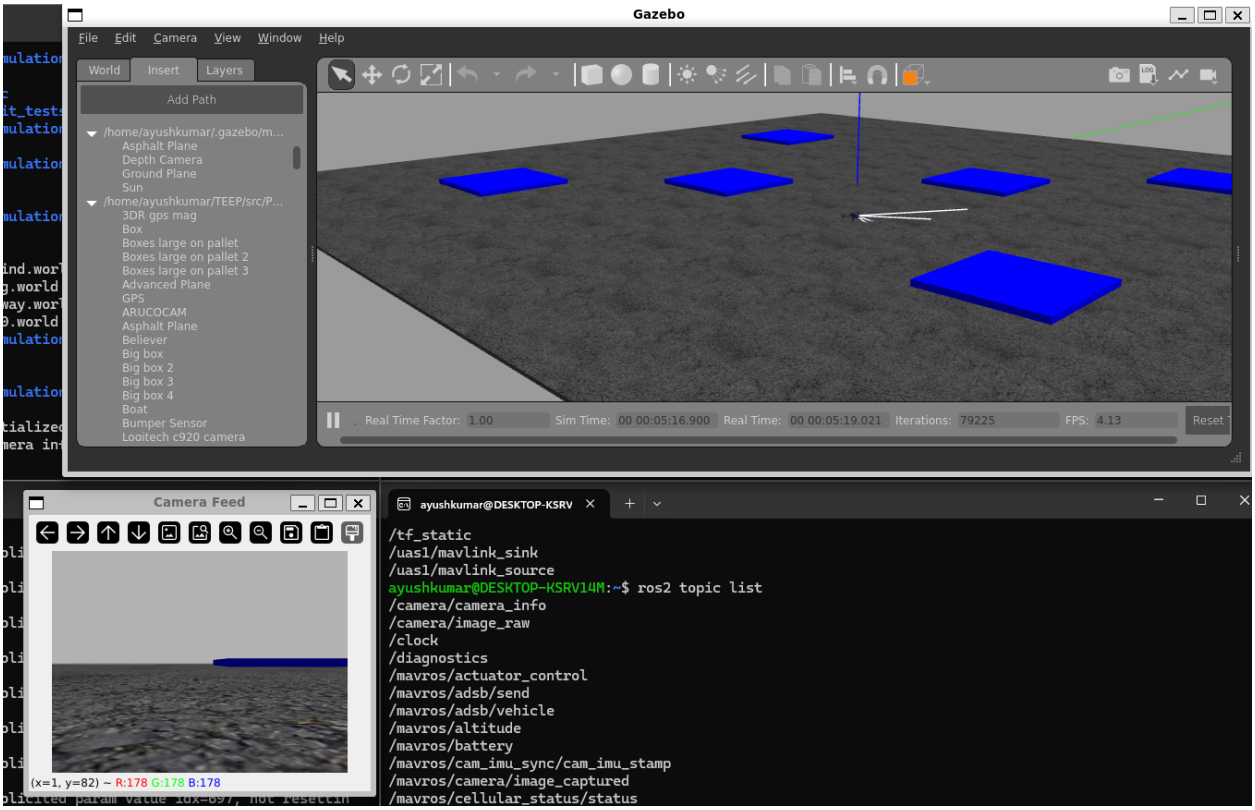


Figure 4: Camera View by OpenCV

8.1 Inference Python Script (camera_feed_inference.py)

Created a Python script that subscribes to the `/camera/image_raw` topic, preprocesses the images, and performs inference with the model.

To run the flood detection and display the result on the camera feed, execute the following command:

```
python3 ~/TEEP/src/camera_feed_inference.py
```

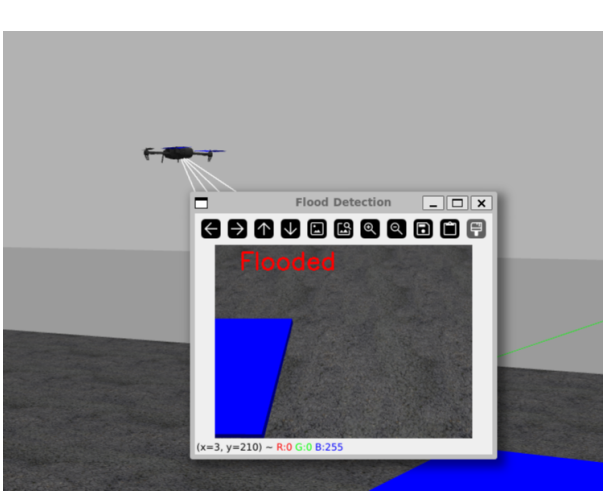
8.2 Real-Time Flood Detection

The flood detection model will process the images in real-time. As new images are received from the camera, they will be classified as "Flooded" or "Non-Flooded," and the result will be displayed on the camera feed in real-time.

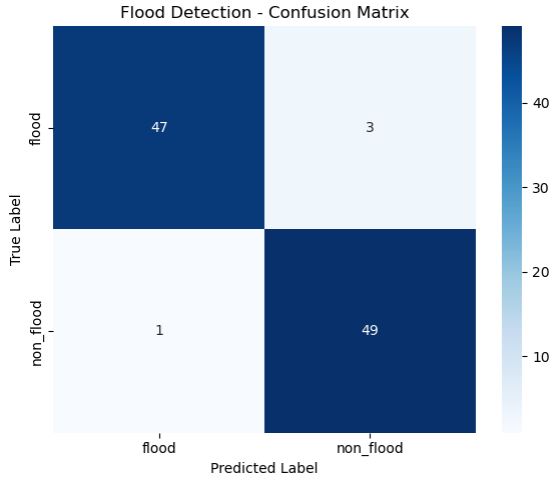
8.3 Final Setup

- Ensure that the camera is publishing the feed to `/camera/image_raw` in ROS 2.
- The `camera_viewer.py` node will display the camera feed.
- The `camera_feed_inference.py` node will classify the image and overlay the result.

By following these steps, we will have a real-time flood detection system for the UAV using Gazebo.



(a) Flood Detection



(b) Confusion Matrix for ResNet-18 Flood Detection Model

Figure 5: Results of Flood Detection and Model Evaluation

8.4 Model Evaluation and Results

The performance of the trained ResNet-18 model was evaluated on a balanced and representative test dataset. The training dataset consisted of **120 flooded** and **80 non-flooded** images, while the testing dataset comprised **50 flooded** and **50 non-flooded** images. Both datasets were a mix of real-world flood imagery and synthetic data generated from a custom Gazebo simulation environment, ensuring diversity and generalization capability.

The model was evaluated using standard classification metrics such as accuracy, precision, recall, and F1-score. The confusion matrix in Figure 27b provides a visual overview of the model's classification

performance. The ResNet-18 model achieved an overall accuracy of **96%** on the test set, demonstrating strong performance on both classes. Specifically:

- **Precision:** 98% for flooded, 94% for non-flooded
- **Recall:** 94% for flooded, 98% for non-flooded
- **F1-Score:** 0.96 for both classes

These results highlight the model's robustness and suitability for deployment in UAV-based flood monitoring systems, where real-time and accurate detection is critical.

9 Image-Based Flood Detection to GPS Mapping Pipeline

This pipeline integrates real-time image classification with GPS data logging to monitor flooded areas using a UAV.

9.1 Pipeline Overview

The system performs the following tasks:

- Subscribes to image data from the onboard UAV camera via the ROS topic:

  `/camera/image_raw`

- Subscribes to GPS data from MAVROS:

  `/mavros/global_position/raw/fix`

- Applies a trained ResNet-18 classification model to the image stream.
- Outputs the following:
  - Classification result: `Flooded` or `Non-Flooded`
  - Real-time GPS coordinates corresponding to each classification

9.2 Running the Mapping Script

To start the real-time classification and GPS logging, run:

```
python3 ~/TEEP/src/gps_mapper.py
```

Alternatively:

```
cd ~/TEEP/src
python3 gps_mapper.py
```

9.3 Output

The script prints the GPS coordinates received from the topic
`/mavros/global_position/raw/fix`
and the ResNet-18 classification output based on the camera feed. All results are logged in a CSV file:

- `flood_map.csv`

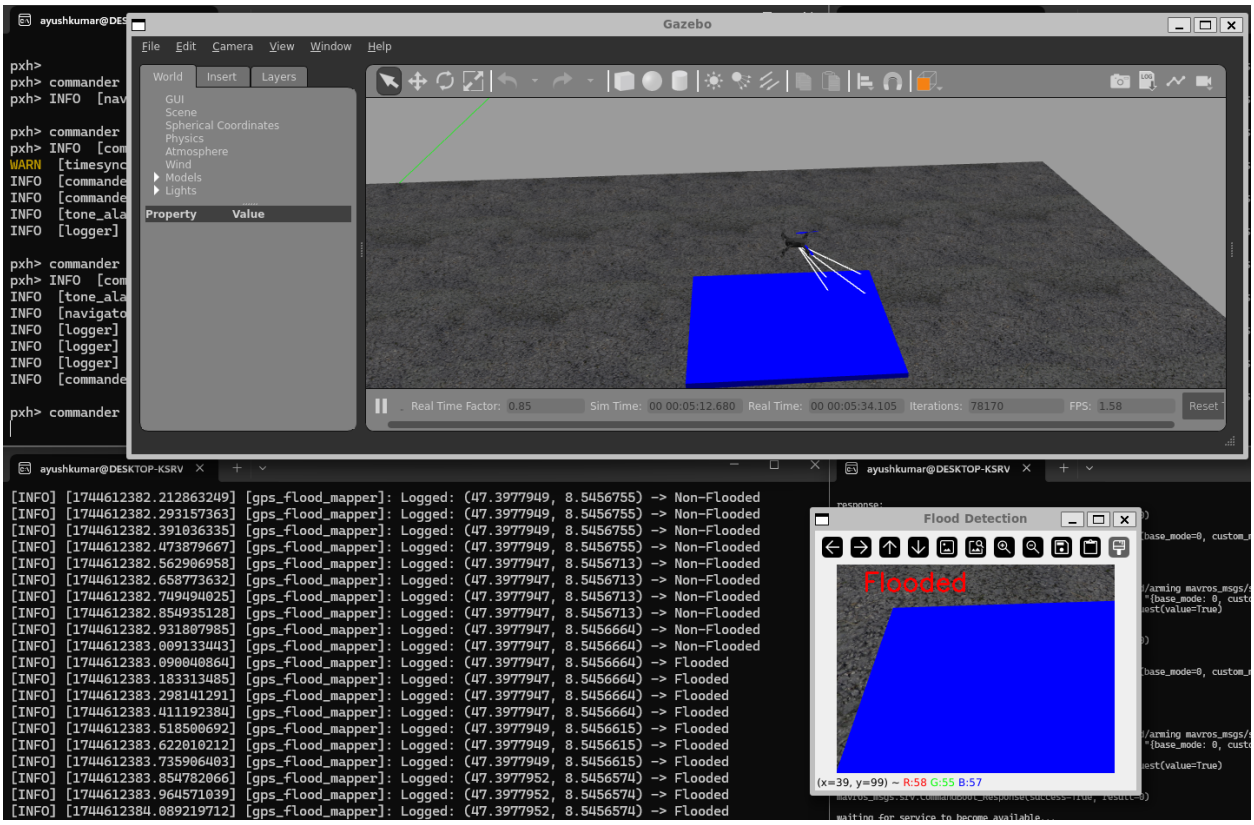


Figure 6: GPS Mapping

9.4 Model Details

The current implementation uses a ResNet-18 model for binary classification of camera images into:

- **Flooded**
- **Non-Flooded**

This classification is then paired with the drone's current GPS coordinates to create a geotagged flood detection map in real time.

10 Flood Detection by UNet Segmentation Model

Flood detection is performed using a deep learning-based UNet segmentation model that classifies each pixel of the image captured by an onboard drone camera as either `Flooded` or `Non-Flooded`.

10.1 Training Procedure for Flood Segmentation using Synthetic Gazebo Images

To accurately segment flooded regions from aerial images, we developed and trained a deep learning model based on the U-Net architecture. The training was conducted using a fully synthetic dataset generated from custom-designed Gazebo simulation environments. These environments emulate various flood scenarios in urban and semi-urban settings and provide precise pixel-level ground truth masks.

10.1.1 Dataset Preparation

The dataset consists of paired RGB images and binary segmentation masks. These images were generated by simulating UAV flights in Gazebo over custom flood-prone worlds, where water levels, terrain textures, lighting conditions, and camera angles were varied to introduce realistic diversity. Each RGB image corresponds to a binary mask that labels pixels as either flooded (value 1) or non-flooded (value 0). The dataset was organized into two directories:

- `images/` – RGB input images from Gazebo camera
- `masks/` – Ground truth segmentation masks

All images and masks were resized to 256×256 pixels and normalized using torchvision transforms. A custom PyTorch dataset class was implemented to handle image-mask pairing and data loading.

10.1.2 Model Architecture

We employed the U-Net architecture, which is widely used for biomedical and environmental image segmentation tasks. It features an encoder-decoder structure with skip connections, enabling the model to capture both spatial context and fine-grained details. The model was initialized with:

- 3 input channels (RGB)
- 1 output channel (sigmoid-activated binary mask)

10.1.3 Training Setup

The model was trained using the following hyperparameters:

- Epochs: 20
- Batch size: 8
- Learning rate: 1×10^{-4}
- Optimizer: Adam
- Loss Function: Binary Cross Entropy with Logits (BCEWithLogitsLoss)

10.1.4 Training Results

The model converged steadily over 20 epochs, minimizing the loss function and learning to accurately distinguish between flooded and non-flooded regions. The use of synthetic data from Gazebo ensured clean, noise-free supervision, enabling the model to generalize well on varied flood scenarios.

10.1.5 Conclusion

By leveraging custom Gazebo simulations, we generated a scalable and controllable dataset for supervised training of a U-Net based segmentation model. The approach eliminates the need for manual annotation and ensures high-quality supervision, which is particularly useful for applications like UAV-based disaster monitoring and response.

10.2 Flood Segmentation Evaluation

The performance of the flood segmentation model was evaluated using a test dataset of paired images and masks. A pre-trained U-Net model was used to generate binary masks indicating flooded and non-flooded regions. The following evaluation metrics were computed:

Table 2: Segmentation Model Performance Metrics

Metric	Value
IoU Score	0.9324
Dice Coefficient	0.9565
Pixel Accuracy	0.9963
Precision	0.9862
Recall	0.9873
F1 Score	0.9867

The confusion matrix, shown in Figure 7, summarizes the prediction results:

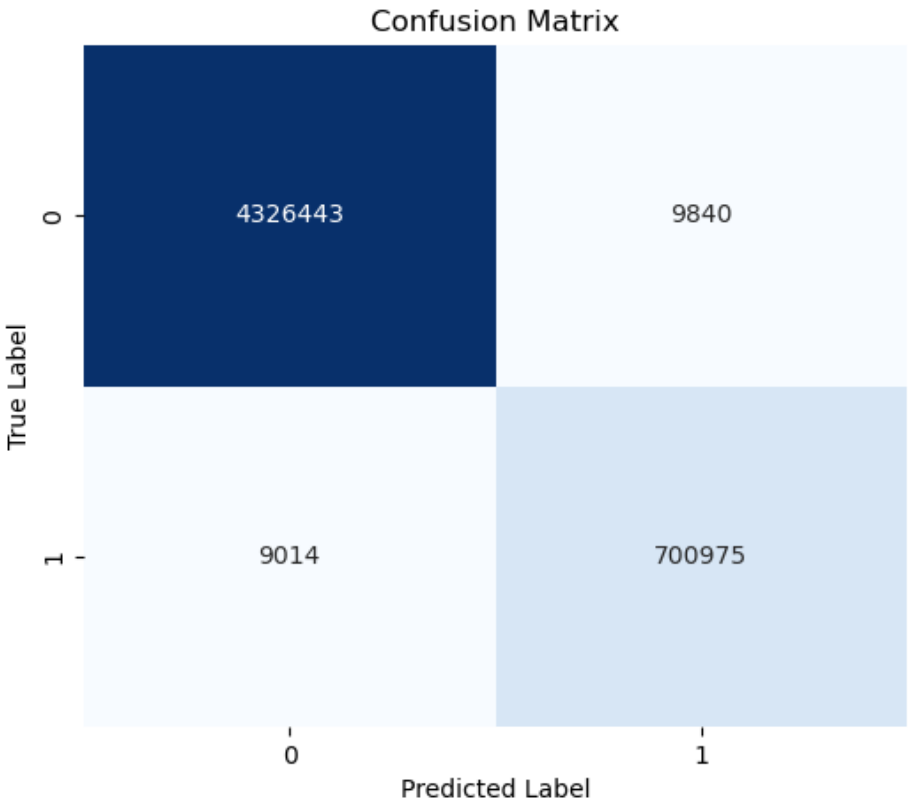


Figure 7: Confusion Matrix of Flood Segmentation

The model achieves a high segmentation accuracy and exhibits strong performance across all major metrics, indicating effective flood region detection from aerial or satellite imagery.

10.3 Execution

Navigate to PX4 Autopilot source and build SITL with Gazebo Classic

```
cd ~/TEEP/src/PX4-Autopilot
make px4_sitl gazebo-classic
```

Launch MAVROS with PX4 SITL using UDP communication

```
ros2 launch mavros px4.launch fcu_url:="udp://:
14540@127.0.0.1:14557"
```

In PX4 shell prompt, set EKF origin to sync drone's height

```
pxh> commander set_ekf_origin 47.397751 8.545607 488.255
```

Verify EKF origin by monitoring local position

```
ros2 topic echo /mavros/local_position/pose
```

```
cd ~/TEEP/src/Flood_Detection_Segmentation
python3 flood_segmentation_inference.py
```

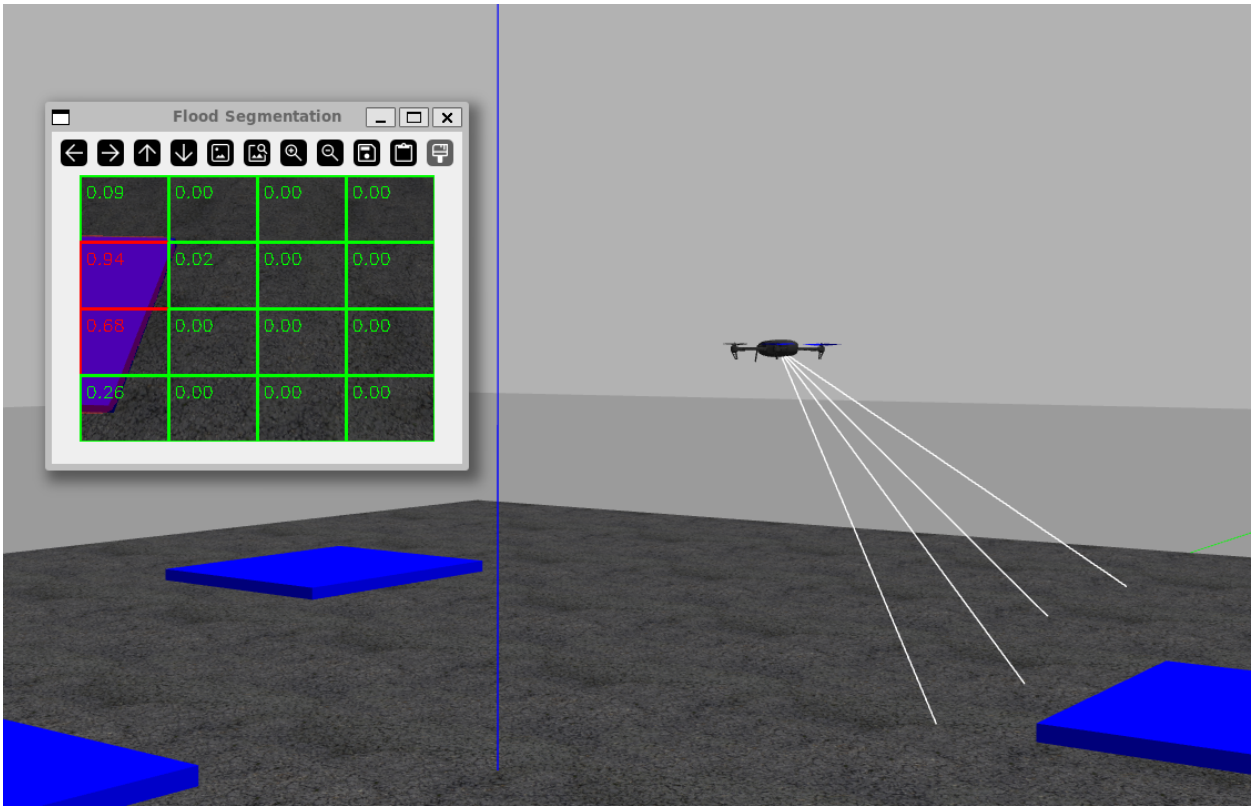


Figure 8: Grid-Based Segmentation Model Simulation

10.4 Drone and Camera Description

The flood detection platform is a quadrotor UAV equipped with a high-resolution RGB camera rigidly mounted at a known pitch angle. The camera's intrinsic parameters include focal lengths f_x, f_y and

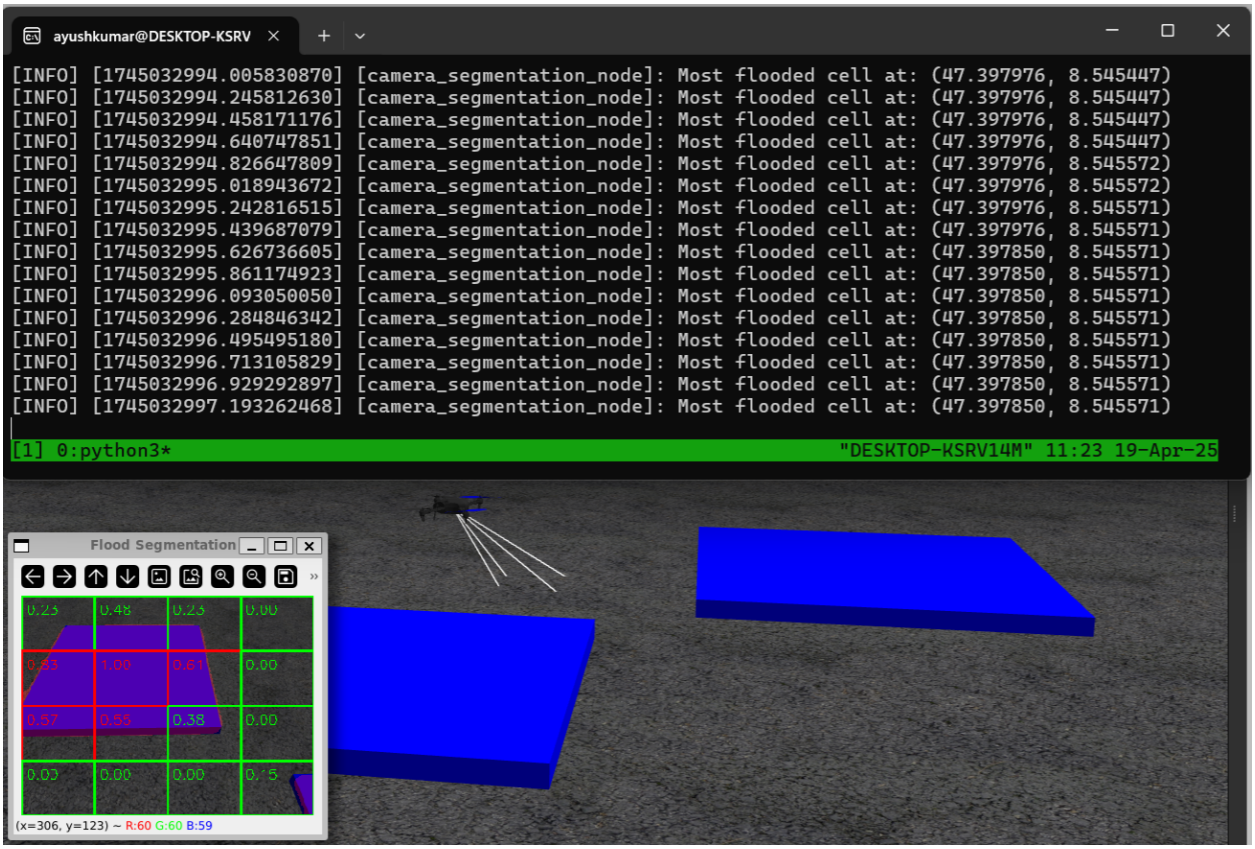


Figure 9: GPS Indication of the Most Flooded Region

principal point coordinates (c_x, c_y), and these are used to construct the intrinsic matrix K. The drone captures real-time images of size $W \times H$ pixels via the topic `/camera/image_raw`. The system also retrieves the drone's current GPS position from `/mavros/global_position/global` (message type: `sensor_msgs/msg/NavSatFix`) and its altitude from the local pose topic `/mavros/local_position/pose` (message type: `geometry_msgs/msg/PoseStamped`).

10.5 Mathematical Model and Coordinate Transformations

- **Image Segmentation:** Each incoming image $I \in \mathbb{R}^{W \times H \times 3}$ is processed through a pretrained UNet segmentation model. The model outputs a binary mask $M \in \{0,1\}^{W \times H}$ where 1 denotes flooded pixels.
- **Grid-based Flood Intensity:** The mask is partitioned into a 4×4 grid. For each cell $C_{m,n}$, the flood ratio is computed as:

$$R_{m,n} = \frac{1}{|C_{m,n}|} \sum_{(i,j) \in C_{m,n}} M_{i,j}$$

- **Most Flooded Cell:** The grid cell with the highest flood intensity is selected:

$$(m^*, n^*) = \arg\max_{m,n} R_{m,n}$$

- **Pixel Ray Projection:** Let (x_c, y_c) be the center of the most flooded cell in pixel coordinates.

Define the homogeneous pixel vector:

$$\mathbf{p} = \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix}$$

The corresponding ray direction in camera coordinates is:

$$\mathbf{r}_c = K^{-1}\mathbf{p}$$

Normalize the ray:

$$\mathbf{r}_c \leftarrow \frac{\mathbf{r}_c}{\|\mathbf{r}_c\|}$$

– **Camera to World Ray:** The camera is assumed to be pitched downward at 45°, represented by a fixed rotation matrix R_{pitch}. The world-frame ray is:

$$\mathbf{r}_w = R_{\text{pitch}} \cdot \mathbf{r}_c$$

Normalize:

$$\mathbf{r}_w \leftarrow \frac{\mathbf{r}_w}{\mathbf{r}_w^z}$$

– **Ground Plane Intersection:** Given the drone's altitude h (from local pose), the ground intersection point in the local ENU (East-North-Up) frame is:

$$\mathbf{p}_{\text{ground}} = -h \cdot \mathbf{r}_w$$

– **Local Offset to GPS:** Let (ϕ, λ) be the current GPS position. Define local offsets:

$$\Delta x = -\mathbf{p}_{\text{ground}}^x, \quad \Delta y = -\mathbf{p}_{\text{ground}}^y$$

Compute the azimuth and distance:

$$\text{azimuth} = \arctan 2(\Delta x, \Delta y), \quad \text{distance} = \sqrt{\Delta x^2 + \Delta y^2}$$

Use the WGS84 geodesic library to get the projected GPS target:

$$(\phi', \lambda') = \texttt{Geodesic.WGS84.Direct}(\phi, \lambda, \text{azimuth}, \text{distance})$$

– **Waypoint Publication:** The new GPS coordinate (ϕ', λ') corresponding to the most flooded grid cell is published to `/next_gps_waypoint` as a `geographic_msgs/msg/GeoPoseStamped` message.

10.6 Functional Description

– The node subscribes to `/camera/image_raw` to receive real-time RGB frames from the UAV's downward-facing camera.

– Each frame is processed through a pretrained UNet model, which outputs a binary segmentation mask distinguishing flooded and non-flooded pixels.

– The segmentation mask is divided into a 4×4 grid, and the flood intensity in each cell is computed as the fraction of pixels labeled as flooded.

– The cell with the highest flood ratio is selected as the region of interest.

– The center pixel of this most flooded cell is identified to serve as a representative image location.

– The pixel coordinate is transformed into a 3D direction vector in the camera frame using the inverse of the intrinsic camera matrix.

– This direction vector is then rotated to the world frame using a fixed extrinsic rotation matrix corresponding to the camera's pitch orientation.

- The direction vector is projected onto the ground plane using the UAV's current altitude (from `/mavros/local_position/pose`).
- The ground intersection point is expressed in the UAV's local ENU frame as an offset from the drone's current position.
- The drone's current GPS location is obtained from `/mavros/global_position/global`, and the local offset is converted into global coordinates using the Geodesic WGS84 projection.
- The resulting GPS coordinates of the detected flood region are published to `/next_gps_waypoint` as a `geographic_msgs/msg/GeoPoseStamped` message.

10.7 Camera Intrinsics

Camera intrinsics are internal parameters that define the mapping between 3D points in the camera coordinate frame and their corresponding 2D projections on the image plane. In our flood detection pipeline, they are essential for reconstructing viewing rays from pixel centroids, enabling us to project segmented image regions onto the ground plane in the world frame.

10.7.1 Intrinsic Matrix

The camera intrinsics are encapsulated by the intrinsic matrix $\mathbf{K}$:

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

where:

- f_x, f_y denote the focal lengths (in pixels) along the horizontal and vertical image axes, respectively.
- c_x, c_y represent the coordinates of the principal point (optical center) in the image frame.

This matrix is used to back-project a 2D pixel location into a normalized 3D direction vector in the camera frame using:

$$\mathbf{r}_c = \mathbf{K}^{-1} \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix}$$

where (x_c, y_c) denotes the centroid of flooded pixels in image coordinates.

10.7.2 Parameter Values

In our system, the intrinsic parameters are obtained from the `/camera/camera_info` topic. The calibrated values used in this pipeline are:

$$f_x = 277.19, \quad f_y = 277.19, \quad c_x = 160.5, \quad c_y = 120.5$$

These values correspond to a camera with square pixels and a principal point located near the center of a 320×240 image frame.

10.7.3 Obtaining Camera Intrinsics

Intrinsic parameters are typically derived through a calibration process, which involves:

1. Capturing images of a known calibration target (e.g., a checkerboard pattern).
2. Detecting feature points (e.g., corners) across multiple images.
3. Estimating the intrinsic matrix and lens distortion coefficients using calibration algorithms such as OpenCV's `calibrateCamera()`.

In ROS, if the camera is factory-calibrated, these parameters are published on the `/camera/camera_info` topic as part of the camera's sensor message.

Note: In our implementation, the pixel-derived direction vector $\mathbf{r}_c$ is later transformed into the world frame and intersected with the ground plane to compute geospatial flood coordinates.

10.8 Example: Computing GPS Coordinate for the Most Flooded Cell (Bottom-Left)

Given the drone's current position and altitude:

- Latitude: $\phi_0 = 47.3977504°$
- Longitude: $\lambda_0 = 8.5456074°$
- Altitude above ground: $h = 5.54$ m

The objective is to compute the GPS coordinate corresponding to the **center of the bottom-left cell** of the segmentation mask, assumed to be the most flooded region detected by the UAV's camera.

Step 1: Identify the pixel coordinates of the cell center

The segmentation mask is assumed to be a 320×240 pixel image divided into a 4×4 grid. Each cell therefore has size:

$$\text{cell width} = \frac{320}{4} = 80 \text{ pixels}, \quad \text{cell height} = \frac{240}{4} = 60 \text{ pixels}.$$

The center of the bottom-left cell (first column, last row) in pixel coordinates is:

$$\mathbf{p} = (x, y) = (40,\ 210)$$

Step 2: Compute the direction vector (ray) in the camera frame

Using the camera intrinsic parameters:

$$K = \begin{bmatrix} 277.19 & 0 & 160.5 \\ 0 & 277.19 & 120.5 \\ 0 & 0 & 1 \end{bmatrix}$$

Convert the pixel to homogeneous coordinates:

$$\tilde{\mathbf{p}} = \begin{bmatrix} 40 \\ 210 \\ 1 \end{bmatrix}$$

Compute the ray direction in the camera frame:

$$\mathbf{r}_{\text{cam}} = K^{-1}\tilde{\mathbf{p}}$$

Step 3: Rotate the ray to align with the world frame

Assuming the camera is pitched downward by 45°, the rotation matrix about the Y-axis is:

$$R_{\text{pitch}} = \begin{bmatrix} \cos 45^\circ & 0 & \sin 45^\circ \\ 0 & 1 & 0 \\ -\sin 45^\circ & 0 & \cos 45^\circ \end{bmatrix}$$

Then:

$$\mathbf{r}_{\text{world}} = R_{\text{pitch}} \cdot \mathbf{r}_{\text{cam}}$$

Step 4: Project the ray onto the ground plane ($z = 0$)

To find the intersection of the ray with the ground, compute the scale factor:

$$\lambda = -\frac{h}{r_{\text{world},z}}$$

The 2D local coordinates on the ground are then:

$$(x_{\text{ground}}, y_{\text{ground}}) = \lambda \cdot (r_x, r_y)$$

Step 5: Convert local ground coordinates to GPS coordinates

Compute the azimuth:

$$\alpha = \arctan 2(x_{\text{ground}}, y_{\text{ground}}) \quad \Rightarrow \quad \alpha_{\text{deg}} = (\text{degrees}(\alpha) + 360) \bmod 360$$

Compute the horizontal ground distance:

$$d = \sqrt{x_{\text{ground}}^2 + y_{\text{ground}}^2}$$

Use the geodesic projection:

$$(\phi_t, \lambda_t) = \texttt{Geodesic.WGS84.Direct}(\phi_0, \lambda_0, \alpha_{\text{deg}}, d)$$

```
from geographiclib.geodesic import Geodesic
geod = Geodesic.WGS84
result = geod.Direct(47.3977504, 8.5456074, azimuth_deg, distance)
target_lat = result['lat2']
target_lon = result['lon2']
```

Result:

For the given inputs (altitude $h = 5.54$ m), the computed GPS coordinate for the most flooded bottom-left cell is approximately:

$$\boxed{\phi_t = 47.397770^\circ, \quad \lambda_t = 8.545630^\circ}$$

This point indicates where the UAV should navigate for closer inspection or flood response.

The most flooded cell was identified as the bottom-left cell. The GPS coordinates for its center were computed and the drone was successfully navigated to that location.

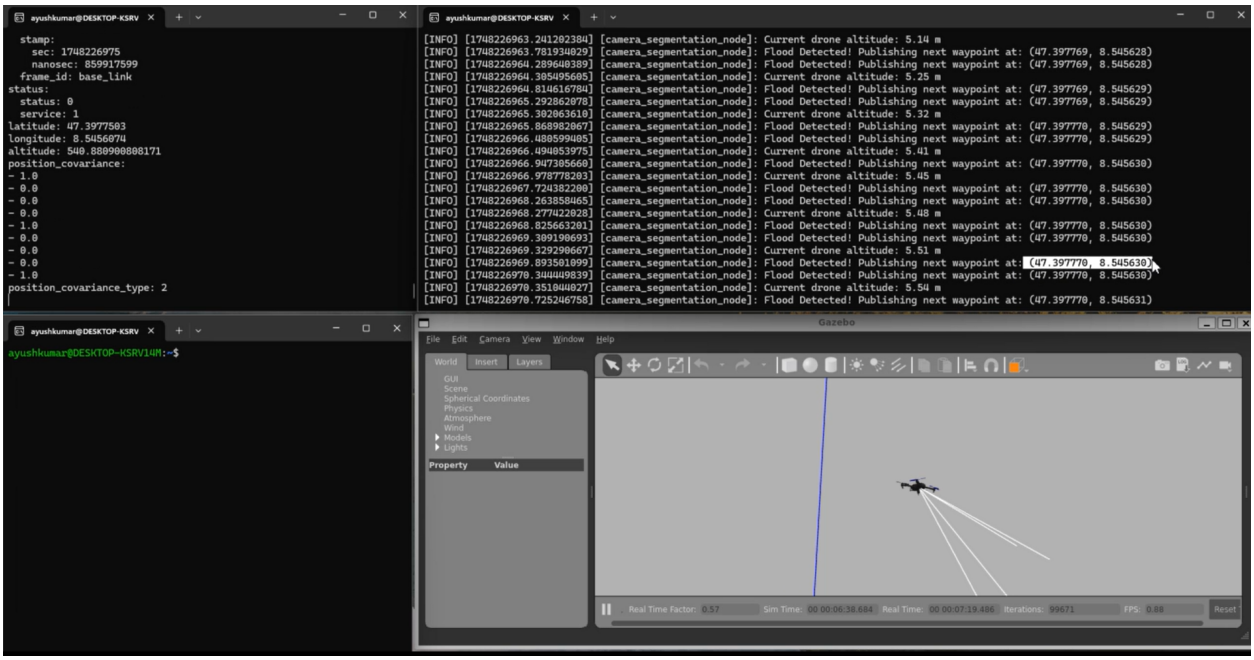


Figure 10: Drone at current GPS location (47.3977504°N, 8.5456074°E)

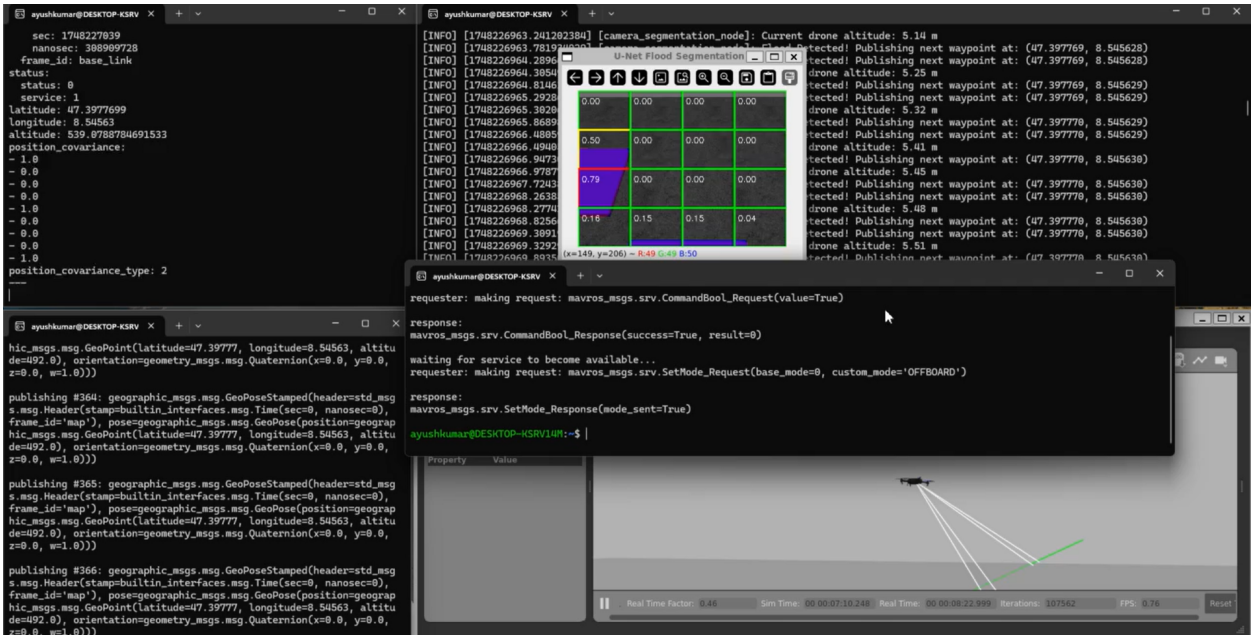


Figure 11: Drone at targeted GPS location (47.397770°N, 8.545630°E) over most flooded region

11 Autonomous Flight Movement

This ROS 2 node enables autonomous UAV navigation toward the most flooded region, as identified by the UNet segmentation model. It listens for target GPS coordinates and publishes real-time global position setpoints to guide the drone in flight.

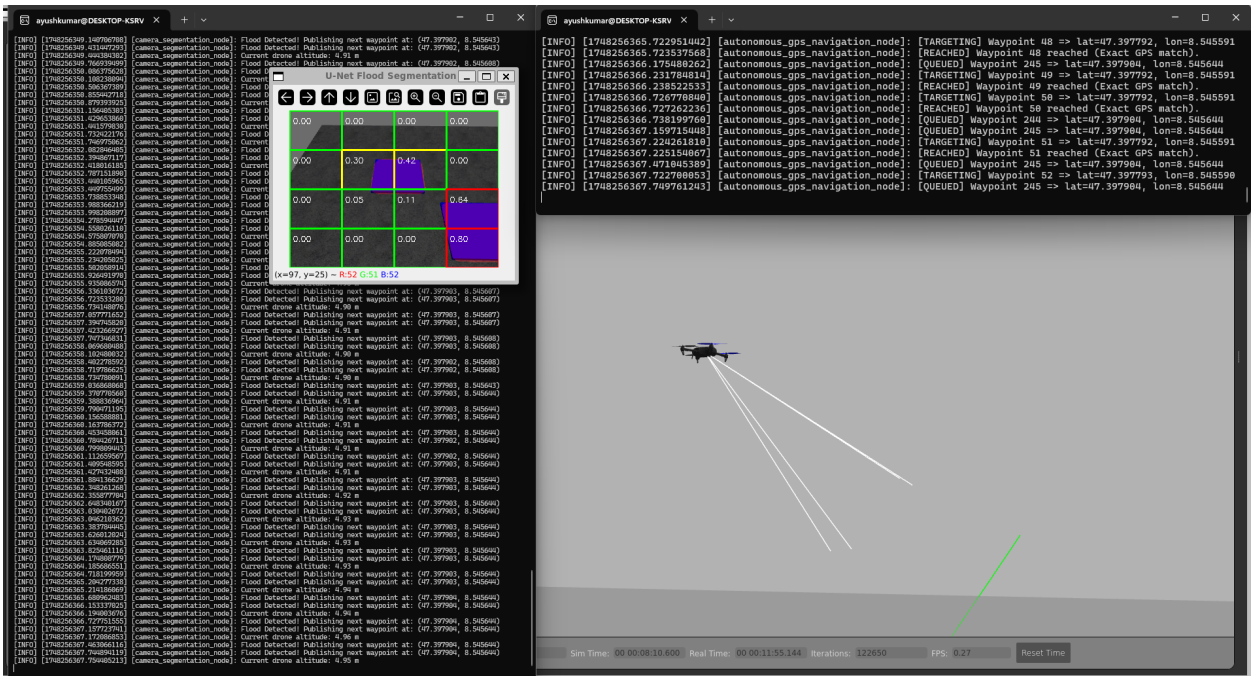


Figure 12: Autonomous UAV Navigation Toward Flooded Region

11.1 Execution

We can launch the autonomous navigation node using either of the following methods:

```
# Using ros2 run
ros2 run flood_detection_segmentation
autonomous_gps_navigation_node

# Or manually
cd ~/TEEP/src/Flood_Detection_Segmentation
python3 autonomous_gps_navigation_node.py
```

11.2 Functional Description

- Subscribes to `/next_gps_waypoint`, a topic of type `geographic_msgs/ msg/GeoPoseStamped`, which provides the GPS coordinates of a flood-affected region, typically generated by a segmentation or detection node.
- Subscribes to `/mavros/global_position/global` (`sensor_msgs/msg/ NavSatFix`) to access the drone's real-time global GPS coordinates.
- Subscribes to `/mavros/altitude` (`mavros_msgs/msg/Altitude`) to obtain AMSL (Above Mean Sea Level) altitude for accurate altitude-based control during takeoff and waypoint navigation.
- Subscribes to `/mavros/local_position/pose` (`geometry_msgs/msg/ PoseStamped`) to track local Z-coordinate for takeoff monitoring.
- Publishes GPS navigation setpoints to `/mavros/setpoint_position/ global` using `geographic_msgs /msg/GeoPoseStamped` to direct the drone toward either the takeoff altitude or a flood-affected GPS waypoint.
- Implements a user-prompted startup routine with two options:
  1. Take off to a specified relative altitude and hover.

2. Directly set and fly to a fixed GPS waypoint.

- Maintains a queue of GPS waypoints for sequential autonomous navigation and logs each new target as it is reached.
- Monitors takeoff progress using local altitude and switches to waypoint navigation mode once the target altitude is reached.
- Dynamically adjusts mission altitude relative to AMSL and maintains consistent altitude flight throughout the mission.
- When all waypoints are reached, it triggers a Return-to-Launch (RTL) fallback behavior by publishing the drone's current GPS position as a hold command.

12 Sending Target Waypoints by GPS in SITL using ROS 2 and MAVROS

In the context of PX4 SITL simulation, real GPS data is not available. Instead, PX4 provides simulated GPS through Gazebo. This section explains how to send GPS-based waypoints to a UAV in simulation, using `MAVROS` and ROS 2, while ensuring proper initialization with a fake GPS origin.

12.1 Simulated GPS Data in PX4 SITL

When SITL is launched using:

```
cd ~/TEEP/src/PX4-Autopilot
make px4_sitl gazebo-classic
```

PX4 automatically starts publishing fake GPS data over the topic:

```
/mavros/global_position/raw/fix
```

Example:

```
latitude: 47.3977508
longitude: 8.5456075
altitude: 535.3
```

12.2 Setting Fake GPS Origin

Despite having simulated GPS, PX4's EKF2 module needs an initial global origin to compute local position estimates. This must be manually set in the PX4 shell:

```
commander set_ekf_origin 47.3977508 8.5456075 535.3
```

Additionally, enable the use of fake GPS:

```
param set MAV_USEHILGPS 1
```

12.3 Preparing the Drone for Offboard GPS Navigation

Before sending waypoints, the drone must be:

- Armed
- Set to `OFFBOARD` mode
- Continuously receiving setpoints (at least 2 Hz)

12.4 Sending Waypoint using `GeoPoseStamped`

To send a GPS waypoint using high-level pose information:

```
ros2 topic pub -r 10 /mavros/setpoint_position/global
geographic_msgs/msg/GeoPoseStamped "{
  header: {frame_id: 'map'},
  pose: {
    position: {
      latitude: 47.397751,
      longitude: 8.545607,
      altitude: 545.0
    },
    orientation: {
      w: 1.0
    }
  }
}"
```

12.5 Alternative: Using `GlobalPositionTarget` (Optional)

For finer control (e.g., yaw, velocity), use:

```
ros2 topic pub -r 10 /mavros/setpoint_raw/global
mavros_msgs/msg/GlobalPositionTarget "{
  coordinate_frame: 6,
  type_mask: 4088,
  latitude: 47.3977508,
  longitude: 8.5456075,
  altitude: 535.3
}"
```

However, this is not required if using `GeoPoseStamped` for simple waypoint navigation.

12.6 Summary

To enable GPS-based waypoint movement in Gazebo SITL:

- Use the simulated GPS feed
- Set EKF origin with `commander set_ekf_origin`
- Arm and switch to `OFFBOARD` mode
- Publish GPS goal using `GeoPoseStamped`

This process supports autonomous UAV navigation over global waypoints within the simulated flood environment.

12.7 Extracting Current Drone GPS with Classification Output

To retrieve the UAV's current GPS coordinates and simultaneously obtain the real-time flood classification result, we use a custom ROS 2 Python script: `gps_mapper.py`. This script listens to both the camera feed and the GPS topic.

12.7.1 Usage

Navigate to the project directory and run:

```
cd ~/TEEP/src
python3 gps_mapper.py
```

12.7.2 Functionality

The script performs the following:

- Subscribes to `/camera/image_raw` to receive image frames.
- Subscribes to `/mavros/global_position/raw/fix` to get the latest simulated GPS data.
- Runs ResNet-18 model inference on the camera image to classify the scene as `Flooded` or `Non-Flooded`.
- Prints the latest GPS coordinates (latitude, longitude) alongside the classification result.

12.7.3 Example Output

```
Image received.
Classification: Flooded
Latitude: 47.397751
Longitude: 8.545607
```

12.7.4 Verification

To independently verify the GPS output, we can echo the MAVROS GPS topic:

```
ros2 topic echo /mavros/global_position/raw/fix
```

This workflow enables tight coupling between visual flood detection and geolocation, allowing the UAV to tag flooded zones with global coordinates for downstream path planning or map logging.

12.8 Geospatial Boundaries of the Simulated Gazebo World

For accurate waypoint setting and map tagging, it is crucial to understand the GPS coordinate range covered by the Gazebo simulation world. Since Gazebo uses a local Cartesian frame (in meters), but the drone navigation and classification results operate in global GPS (WGS84) coordinates, we convert the local positional range into geographic latitude and longitude.

12.8.1 Simulated World Scale

Our simulated environment contains models distributed within approximately ± 6 meters in both the x and y directions from the drone's origin point.

12.8.2 Conversion from Meters to Degrees (WGS84)

Using the WGS84 standard, the approximate degree-per-meter conversion at a latitude of 47.4°N is:

- 1 meter in **latitude** $\approx 0.00000899°$
- 1 meter in **longitude** $\approx 0.00001341°$

These conversion values are dependent on the curvature of the Earth and vary slightly by latitude.

12.8.3 GPS Range Calculation

Let the fake GPS origin (i.e., the reference point for the simulated world) be:

- **Latitude:** 47.397751°
- **Longitude:** 8.545607°

Then, the drone can move within a range of:

- **Latitude range:**

$$\begin{aligned} &47.397751 \pm (6 \times 0.00000899) \\ &= 47.397751 \pm 0.00005394 \\ &= \textbf{47.397697 to 47.397805} \end{aligned}$$

- **Longitude range:**

$$\begin{aligned} &8.545607 \pm (6 \times 0.00001341) \\ &= 8.545607 \pm 0.00008046 \\ &= \textbf{8.545526 to 8.545688} \end{aligned}$$

12.8.4 Use Case

Any GPS-based classification result or waypoint should fall within this bounding box to be considered inside the simulation world. Values outside this range may indicate a faulty EKF origin setup or invalid movement commands.

This geospatial mapping is especially important when overlaying flood detection results onto real-world maps or interfacing with external GIS tools.

13 Waypoint Monitoring and Autonomous RTL using MAVROS

This section outlines the implementation of an autonomous waypoint monitoring system using a custom ROS 2 node, `publish_waypoints.py`, which publishes a series of GPS waypoints to the drone in simulation. After traversing the waypoints, the drone automatically switches to **Return to Launch (RTL)** mode and navigates back to its home position.

13.1 1. System Setup

Before running the waypoint publishing node, the PX4 SITL environment and MAVROS must be correctly configured and launched.

1. Launch PX4 SITL with Gazebo Classic:

```
cd ~/TEEP/src/PX4-Autopilot
make px4_sitl gazebo-classic
```

2. Launch MAVROS and connect to PX4 via UDP:

```
ros2 launch mavros px4.launch fcu_url:="udp://:14540@127.0.0.1:14557"
```

13.2 2. Geographic Boundaries for Waypoints

All waypoints published by the node are bounded within:

- Latitude: `47.3977506` – `47.3978000`
- Longitude: `8.5452600` – `8.5456880`

These values represent a tight geographic region near the simulated environment's origin, ensuring realistic yet controlled path traversal in the Gazebo simulation.

13.3 3. Set Home Position

Before publishing waypoints, it is essential to set the drone's home position to its current GPS coordinates. This ensures accurate return during RTL (Return to Launch).

```
ros2 service call /mavros/cmd/set_home mavros_msgs/srv/CommandHome "{
  current_gps: true,
  latitude: 0.0,
  longitude: 0.0,
  altitude: 0.0
}"
```

You can verify the current GPS fix and home position using:

```
ros2 topic echo /mavros/global_position/raw/fix
ros2 topic echo /mavros/home_position/home
```

13.4 4. Launching the Waypoint Publisher Node

The custom node `publish_waypoints.py` is used to send predefined GPS waypoints to the topic `/mavros/setpoint_position/global` at a frequency of 10 Hz, holding each point for 10 seconds.

```
cd ~/TEEP/src
python3 publish_waypoints.py
```

Each waypoint is sent using the `GeoPoseStamped` message type from the `geographic_msgs` package.

Prerequisites:

- Ensure `geographic_msgs` is installed and sourced.
- MAVROS is connected to PX4 SITL.
- The drone is flying in **OFFBOARD** or **GUIDED** mode.

13.5 5. Arming the Drone and Switching to OFFBOARD Mode

The drone must be armed and placed into **OFFBOARD** mode before it can follow waypoints:

```
ros2 service call /mavros/cmd/arming mavros_msgs/srv/
CommandBool "{value: true}"
ros2 service call /mavros/set_mode mavros_msgs/srv/
SetMode "{base_mode: 0, custom_mode: 'OFFBOARD'}"
```

13.6 6. Return to Launch (RTL) After Final Waypoint

After the final waypoint has been reached, the UAV will switch to **RTL** mode using the following function inside the node:

```
from mavros_msgs.srv import SetMode

def trigger_rtl(self):
    cli = self.create_client(SetMode, '/mavros/set_mode')
    while not cli.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('Waiting for /mavros/set_mode service...')
    req = SetMode.Request()
    req.custom_mode = 'RTL'
    future = cli.call_async(req)
    rclpy.spin_until_future_complete(self, future)
    if future.result().mode_sent:
        self.get_logger().info('RTL mode set.')
    else:
        self.get_logger().warn('Failed to set RTL mode.')
```

Important: For RTL to function properly, the home position must be valid and configured using the service call shown earlier. The drone will return to the `/mavros/home_position/home`, not the current GPS fix.

13.7 7. Summary

This node provides an autonomous flight routine for UAVs in PX4 SITL simulation with the following key features:

- Periodic publishing of `GeoPoseStamped` messages for global waypoint navigation.
- Holding each waypoint for a specified duration (e.g., 10 seconds).
- Autonomous switch to RTL mode upon completion of waypoint list.
- Uses MAVROS services for mode switching and arming.

This method is effective for waypoint tracking, path following, and testing mission termination behavior such as RTL in a fully controlled simulation environment.

14 Connect PX4 SITL with Mission Planner

To connect the PX4 SITL (Software-In-The-Loop) simulation running on Linux to Mission Planner on Windows using MAVLink.

Step 1: Find Windows PC's IP Address

- On your Windows PC, open `Command Prompt`.
- Run the following command to get your local IP address:

```
ipconfig
```

- Look for the `IPv4 Address` (e.g., `192.168.16.1`), which will be used for the connection.

14.1 Step 2: Run PX4 SITL with MAVLink Configuration

- On your **Linux system** (where PX4 SITL is running), you need to specify the IP address of your Windows PC so that PX4 can send MAVLink data to Mission Planner.
- Run the following command in the **PX4 shell** to start the simulation with the correct MAVLink address and port:

```
make px4_sitl gazebo-classic MAVLINK_ADDR=192.168.16.1
MAVLINK_PORT=14550
```

- Replace `192.168.16.1` with the IP address obtained in **Step 1**.

14.2 Step 3: Set Up Mission Planner on Windows

- Open **Mission Planner** on **Windows**.
- Go to **Connect** in the top-right corner of Mission Planner.
- Select **UDP** as the connection type.
- Set the **Port** to `14550` (the same port specified in the PX4 SITL command).
- Click **Connect**.

Mission Planner

14.3 Step 4: Verify MAVLink Communication

If the connection does not work, follow these additional steps:

- **Check MAVLink Status** on PX4 SITL (pxh shell):

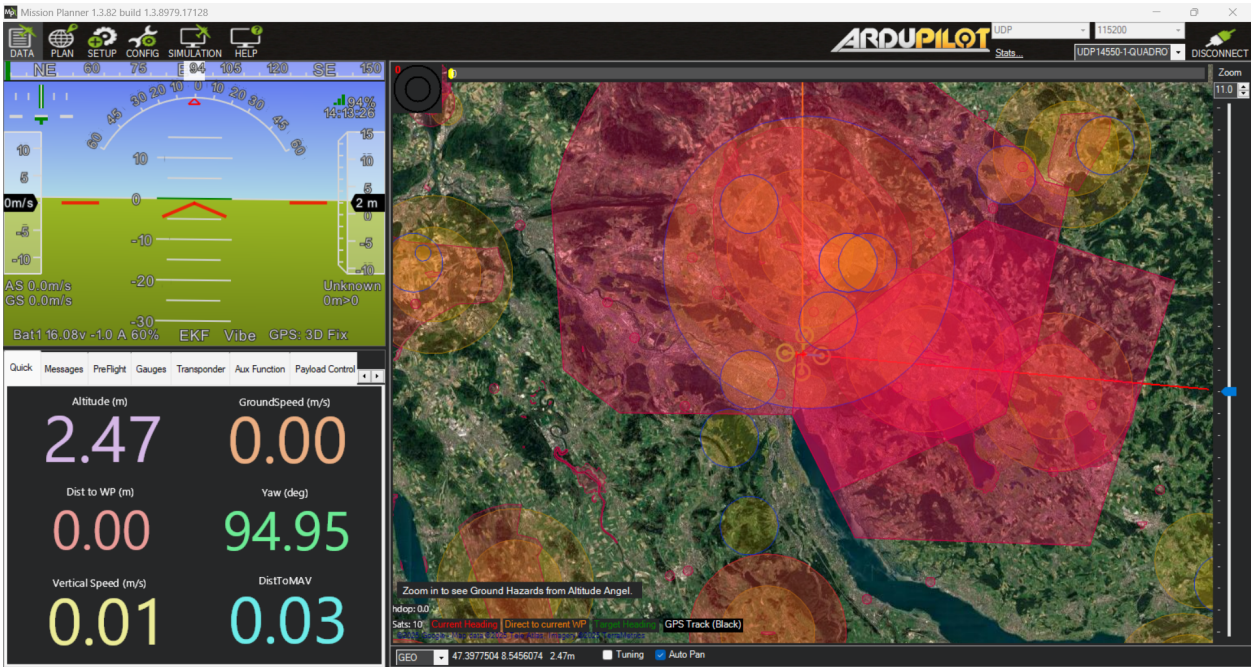
```
mavlink status
```

- This will show if the broadcast is enabled and if data is being transmitted.
- **Restart MAVLink Communication** if necessary:

```
mavlink stop-all
mavlink start -m onboard -t 192.168.16.1 -o 14550 -u 14556 -p
```

```
pxh> mavlink start -m onboard -t 192.168.16.1 -o 14550 -u 14556 -p
INFO  [mavlink] mode: Onboard, data rate: 2880 B/s on udp port 14556 remote port 14550
```

(a) Mavlink Connection



(b) Mission Planner Interface

Figure 13: Mavlink Connection and Mission Planner

– This will configure MAVLink to broadcast to the Windows PC's IP (`192.168.16.1`) on port `14550`.

– **Check MAVLink Status Again**:

```
mavlink status
```

– After restarting MAVLink, verify if `Broadcast enabled` is now YES and data is flowing properly in the `rx` section.

15 DroneKit Setup and Usage

DroneKit is a Python library for interacting with MAVLink-compatible drones and is commonly used for controlling drones in simulation environments like SITL (Software In The Loop). This section details the installation, setup, and usage of DroneKit along with MAVProxy to control a drone simulation.

15.1 Installation of DroneKit and Dependencies

To use DroneKit for controlling drones in simulation, the following packages must be installed:

– **dronekit-sitl**: The SITL (Software In The Loop) package that simulates the drone.

– **pymavlink**: A Python MAVLink library to interact with the drone.

Run the following commands to install these packages:

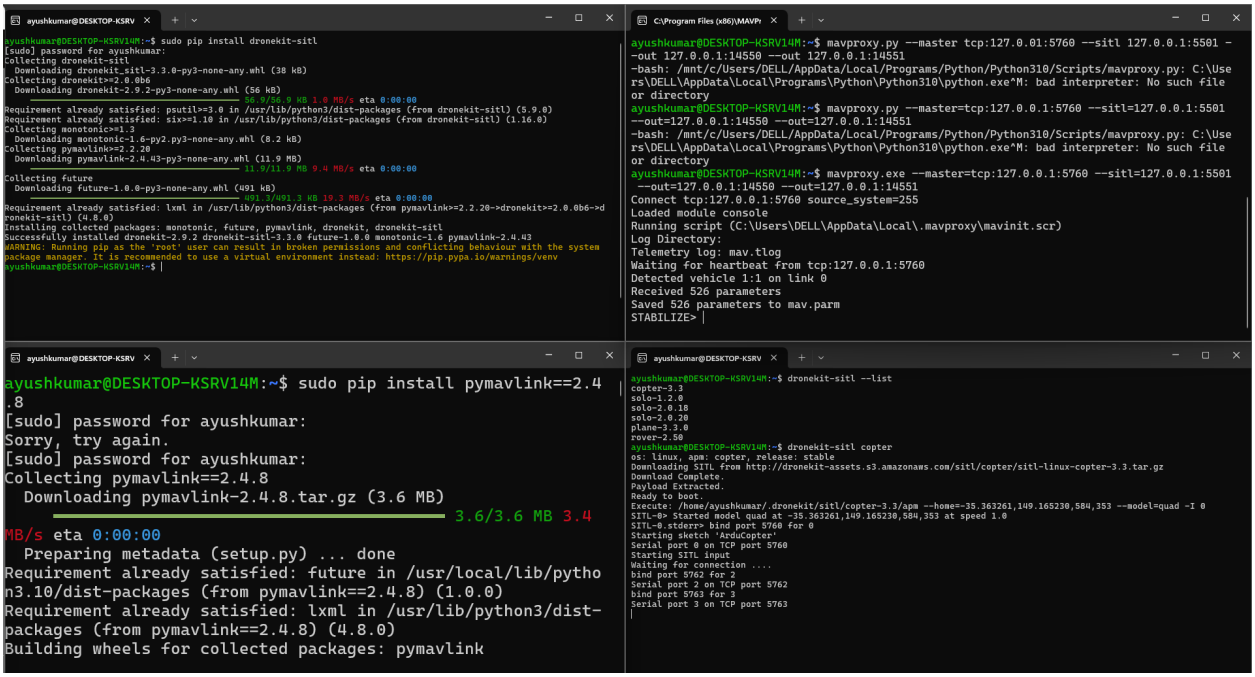


Figure 14: DroneKit Setup

```
sudo pip install dronekit-sitl
sudo pip install pymavlink==2.4.8
```

15.2 List Available Drone Models

Once DroneKit and dependencies are installed, we can list the available drone models for simulation. Run the following command:

```
dronekit-sitl --list
```

This will show a list of available drone models, such as:

- copter-3.3
- solo-1.2.0
- solo-2.0.18
- solo-2.0.20
- plane-3.3.0
- rover-2.50

15.3 Starting a Drone Simulation

To start a drone simulation using DroneKit, use the following command:

```
dronekit-sitl copter
```

This will start the copter simulation and wait for a connection. The terminal will display the message:

```
Waiting for connection ....
```

15.4 Connecting MAVProxy to DroneKit SITL

Once the DroneKit SITL instance is running, we can use MAVProxy to connect to the simulation using the following command to connect MAVProxy to the DroneKit SITL:

```
mavproxy.exe --master=tcp:127.0.0.1:5760 --sitl=127.0.0.1:5501 --
out=127.0.0.1:14550 --out=127.0.0.1:14551
```

Alternatively, we can add the '–map' option to visualize the drone's position on the map:

```
mavproxy.exe --master=tcp:127.0.0.1:5760 --sitl=127.0.0.1:5501 --
out=127.0.0.1:14550 --out=127.0.0.1:14551 --map
```

15.5 Conclusion

Using DroneKit and MAVProxy, we can simulate and control UAVs in a virtual environment. The setup allows us to test different flight commands, mission plans, and scenarios in a safe and controlled environment before real-world deployment.

16 Connection of SITL with Different GCSs

A detailed guide for connecting the Software In The Loop (SITL) simulation with various Ground Control Stations (GCSs) using different tools such as Mission Planner, MAVProxy, and PX4 SITL in Gazebo. This setup is crucial for performing autonomous flight simulations, testing, and validation in a controlled virtual environment before deploying real-world UAVs.

16.1 Prerequisites

Before connecting SITL with the GCS, ensure that the following tools are installed and configured:

- **PX4 SITL** - PX4 Software In The Loop setup in Gazebo.
- **MAVProxy** - A command-line ground station that facilitates communication between SITL and GCS.
- **Mission Planner (MP)** - A graphical user interface GCS for flight control and monitoring.
- **ROS 2** - For integration with MAVROS for communication between PX4 and other systems.

16.2 Connection of Mission Planner with MAVProxy

This section explains how to connect Mission Planner to MAVProxy. MAVProxy is a command-line-based Ground Control Station (GCS) that communicates with MAVLink-compatible autopilots like PX4 or ArduPilot. By connecting MAVProxy to Mission Planner, we can use both GCSs to control and monitor the same UAV, enabling more robust testing and control of UAV operations.

16.2.1 1. Open Mission Planner

To begin the connection process, open Mission Planner on your system.

1. Launch Mission Planner and navigate to the simulation mode.

2. In the Mission Planner interface, select **Simulation** as the connection option.

3. Ensure that the internal SITL (Software In The Loop) simulation is selected within Mission Planner, which will allow the simulator to communicate with MAVProxy.

16.2.2 2. Start MAVProxy in Windows Command Prompt

MAVProxy will handle communication between the GCS and the SITL simulator. To connect MAVProxy to Mission Planner, follow these steps:

1. Open the Windows command prompt.

2. Run the following command to start MAVProxy and configure the communication with Mission Planner:

```
mavproxy.exe --master=tcp:127.0.0.1:5762
```

This command tells MAVProxy to connect to Mission Planner over TCP/IP using the specified IP address and port. The `--master` flag specifies the connection settings to communicate with the SITL simulator in Mission Planner.

16.2.3 3. Connecting MAVProxy to Mission Planner

After starting MAVProxy, the next step is to establish the connection to Mission Planner.

1. In the Mission Planner interface, click `Connect`.

2. Enter the following connection string in the Mission Planner connection settings:

```
tcp:127.0.0.1:5762
```

This connects Mission Planner to MAVProxy using the TCP protocol on the local machine at the specified IP address (127.0.0.1) and port (5762).

3. Set the source system ID as `255` to establish a proper connection. In the Mission Planner window, ensure the source system is set to the same system ID used by MAVProxy.

```
Connect tcp:127.0.0.1:5762 source_system=255
```

After entering this information, click `Connect`, and Mission Planner should now be connected to MAVProxy, receiving telemetry data and allowing for control of the UAV in the simulation.

16.2.4 4. Verifying the Connection

Once the connection is successfully established, you will see the following in Mission Planner:

- The system status and telemetry data will be displayed in real-time on the Mission Planner interface.
- The map view will show the UAV's position in the simulation environment.
- The flight mode (e.g., **LOITER**) will be reflected in the Mission Planner display.

In the MAVProxy console, you should also see the connection status, which will indicate that it has successfully connected to Mission Planner over TCP and is receiving data from the simulated UAV.

16.3 Connection of Mission Planner with PX4 SITL in Gazebo

This section explains how to connect Mission Planner (MP) with a PX4 Software In The Loop (SITL) instance running in Gazebo. This configuration allows real-time monitoring and control of the simulated UAV from Mission Planner, providing a GUI-based ground control station for managing the flight simulation.

16.3.1 1. Building PX4 SITL for Gazebo

The first step is to build the PX4 SITL in Gazebo, which simulates the UAV in a virtual environment. To do this:

1. Navigate to the PX4-Autopilot directory in your terminal:

```
cd ~/TEEP/src/PX4-Autopilot
```

2. Build the PX4 SITL for Gazebo by running the following command:

```
make px4_sitl gazebo-classic
```

This will compile the necessary code for PX4 and launch the SITL simulation in Gazebo.

16.3.2 2. Launching MAVROS for PX4 Communication

Next, you need to launch MAVROS to facilitate communication between PX4 and the Mission Planner. MAVROS acts as a bridge between ROS 2 and PX4 autopilot.

Run the following command to start MAVROS with the correct settings for PX4 SITL communication:

```
ros2 launch mavros px4.launch fcu_url:="udp://:14540@127.0.0.1:14557"
```

This command launches the MAVROS node and sets up communication between PX4 SITL and ROS 2 over UDP. The FCU URL specifies the communication port for MAVLink communication.

16.3.3 3. Starting MAVLink Communication in PX4 SITL

After starting MAVROS, the next step is to initiate the MAVLink communication in PX4. To do this:

1. Open the PX4 shell (pxh¿) and stop any existing MAVLink connections by running:

```
pxh> mavlink stop-all
```

2. Start the MAVLink communication in the onboard mode:

```
pxh> mavlink start -m onboard -t 192.168.16.1 -o 14550 -u 14556
```

This sets up the MAVLink communication between PX4 and the GCS. The flags used are: - `-m onboard` specifies that MAVLink will run in onboard mode. - `-t 192.168.16.1` defines the target IP address for MAVLink communication. - `-o 14550 -u 14556` set the output and input UDP ports.

16.3.4 4. Connecting Mission Planner to PX4 SITL

Once MAVLink communication has been established, the final step is to connect Mission Planner to the PX4 SITL. Open Mission Planner and connect to the PX4 SITL instance using the following connection string:

```
connect udp:127.0.0.1:14550
```

This command connects Mission Planner to the UDP stream provided by PX4 SITL, allowing real-time monitoring and control of the simulated UAV. After connecting, Mission Planner will display telemetry and flight status information.

16.3.5 5. Verifying the Connection in Mission Planner

Once the connection is established, Mission Planner should be able to display the flight mode, system status, and other telemetry data from the simulated UAV. You can also interact with the UAV by sending commands from Mission Planner.

- The connection status should show that Mission Planner is receiving data from PX4 SITL.
- The flight mode should reflect the current mode set in PX4 SITL (e.g., **LOITER**).
- The map in Mission Planner will show the current position of the UAV in the simulation.

16.4 Connection of MAVProxy with PX4 SITL

This section outlines the process of connecting MAVProxy to a PX4 Software In The Loop (SITL) simulation running in Gazebo. MAVProxy is a command-line-based ground control station (GCS) that can communicate with PX4 SITL and be used for controlling and monitoring UAVs in a simulated environment.

16.4.1 1. Building PX4 SITL for Gazebo

To begin, you need to build the PX4 SITL (Software In The Loop) for Gazebo. This simulates the drone and creates the virtual environment for testing.

1. Navigate to the PX4-Autopilot directory in your terminal:

```
cd ~/TEEP/src/PX4-Autopilot
```

2. Build the PX4 SITL for Gazebo by running the following command:

```
make px4_sitl gazebo-classic
```

This command compiles the necessary code for the PX4 autopilot and launches the SITL simulation within Gazebo.

16.4.2 2. Starting MAVLink Communication for PX4 SITL

After building the PX4 SITL, the next step is to start MAVLink communication to allow MAVProxy to communicate with the PX4 autopilot in the simulation.

1. Open the PX4 shell (pxh¿) and start the MAVLink communication with the following command:

```
pxh> mavlink start -m onboard -t 192.168.16.1 -o 14550 -u 14556
```

- `-m onboard` specifies that MAVLink will operate in onboard mode.

- `-t 192.168.16.1` specifies the target IP address for MAVLink communication.

- `-o 14550 -u 14556` defines the output and input UDP ports.

This will set up the MAVLink communication between PX4 SITL and the GCS.

16.4.3 3. Connecting MAVProxy to PX4 SITL

Once MAVLink communication has been started, the next step is to connect MAVProxy to PX4 SITL. MAVProxy will serve as the command-line interface for monitoring and controlling the UAV in the simulation.

Run the following command in the Windows command prompt to connect MAVProxy to PX4 SITL:

```
mavproxy.exe --master=udp:0.0.0.0:14550 --console --map
```

- `--master=udp:0.0.0.0:14550` specifies that MAVProxy will connect to the SITL over the UDP protocol on port 14550. - `--console` enables the MAVProxy console for input and output commands. - `--map` enables the map module to visualize the UAV's position.

Once the command is executed, you will see the following output in the MAVProxy console:

```
Connect udp:0.0.0.0:14550 source_system=255
Loaded module console
Loaded module map
Running script (C:\Users\DELL\AppData\Local\.mavproxy\mavinit.scr)
Log Directory:
Telemetry log: mav.tlog
Waiting for heartbeat from 0.0.0.0:14550
Detected vehicle 1:1 on link 0
Received 892 parameters
Saved 893 parameters to mav.parm
LOITER>
```

- `Waiting for heartbeat from 0.0.0.0:14550` indicates that MAVProxy is waiting to receive data from the PX4 SITL instance. - `Detected vehicle 1:1 on link 0` confirms that MAVProxy has successfully detected the vehicle. - `LOITER` indicates the UAV is currently in **LOITER** mode.

16.4.4 4. Monitoring the PX4 SITL in MAVProxy

After connecting MAVProxy to the PX4 SITL, you can monitor and control the UAV from the MAVProxy console. The console will display real-time information, such as the mode of the UAV, system status, and telemetry data.

For example, after the UAV is connected, you may see:

```
Running script C:\Users\DELL\AppData\Local\.mavproxy\mavinit.scr
-> set moddebug 2
online system 1
Mode LOITER
fence breach
GPS lock at 488 meters
AP: GCS connection regained
```

- `Mode LOITER` shows that the UAV is in the *LOITER* mode. - `GPS lock at 488 meters` confirms that the UAV has acquired GPS data with a lock at 488 meters above ground. - `fence breach` indicates that the UAV has crossed the set boundaries for its operation area.

16.5 Connection of Mission Planner and MAVProxy with PX4 SITL

This section describes the configuration required to connect two Ground Control Stations (GCSs), namely MAVProxy and Mission Planner, simultaneously to the PX4 SITL running in Gazebo Classic. This multi-GCS setup allows monitoring and controlling the UAV from two interfaces, offering flexibility in mission management and telemetry analysis.

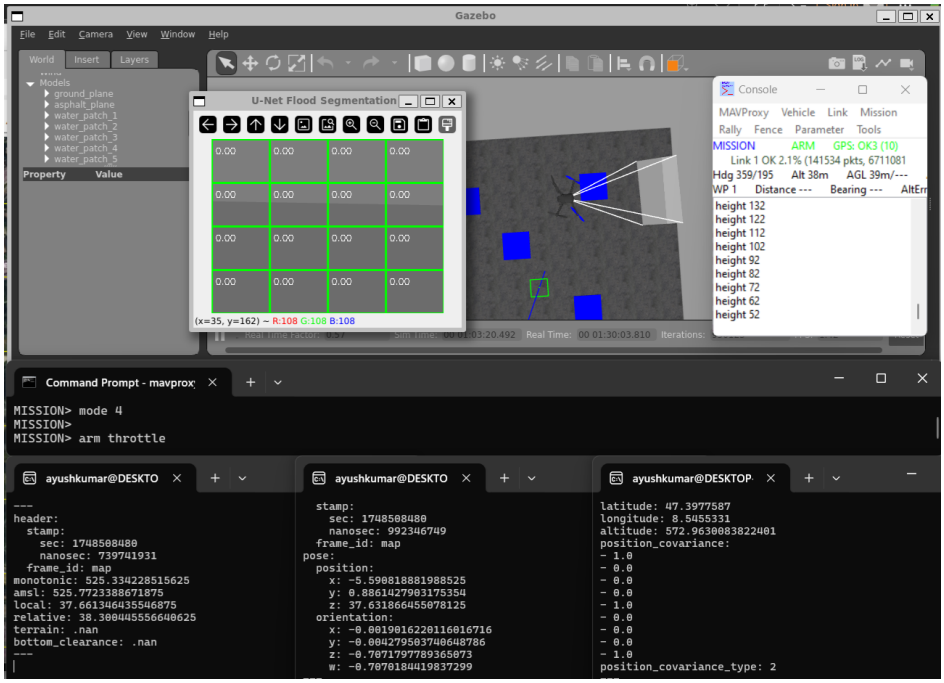


Figure 15: Mavproxy with PX4 SITL

MAVLink Communication Framework

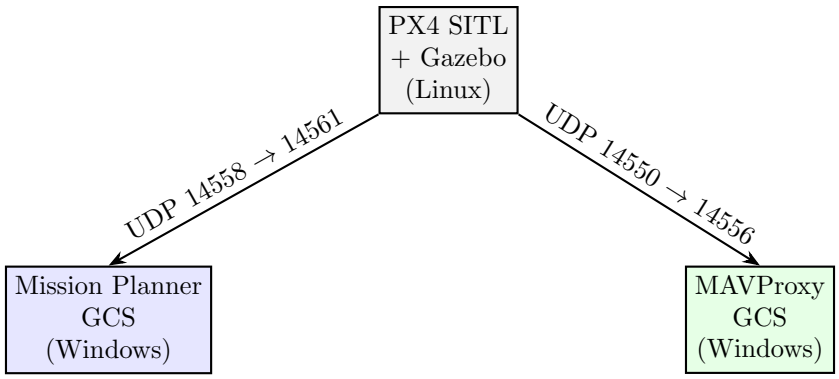


Figure 16: PX4 SITL connected to Mission Planner and MAVProxy GCSs over UDP (Compact View)

16.5.1 Step 1: Launch PX4 SITL in Gazebo

Begin by launching the PX4 SITL simulation with Gazebo:

```
cd ~/TEEP/src/PX4-Autopilot
make px4_sitl gazebo-classic
```

16.5.2 Step 2: Connect MAVProxy to PX4 SITL

In the PX4 shell (pxh), start the MAVLink communication for MAVProxy on port `14550`:

```
mavlink start -m onboard -t 192.168.16.1 -o 14550 -u 14556
```

Then, on your Windows host, launch MAVProxy using the following command:

```
mavproxy.exe --master=udp:0.0.0.0:14550 --console --map
```

This initiates the MAVProxy interface, allowing full command-line control and real-time mapping. On successful connection, you should observe:

- Detection of the vehicle via heartbeat.
- Loading of parameters (e.g., "Received 892 parameters").
- Drone entering `LOITER` mode.
- Console logs such as "GCS connection regained" and "fence breach" (useful for debugging).

16.5.3 Step 3: Connect Mission Planner to PX4 SITL

To connect Mission Planner as a second GCS, start another MAVLink stream on a different port (e.g., 14558):

```
mavlink start -m onboard -t 192.168.16.1 -o 14558 -u 14561
```

Next, open Mission Planner:

- Click on "Simulation".
- Choose "UDP" and enter port `14558`.
- Click "Connect".

Once connected, Mission Planner will begin receiving live telemetry and mission data from the drone. It can be used simultaneously with MAVProxy without any communication conflicts.

16.5.4 Outcome

This configuration enables a powerful dual-GCS setup where:

- MAVProxy is used for command-line control, scripting, and debugging.
- Mission Planner is used for graphical mission planning, real-time monitoring, and visualization.

Note: Both GCSs must be assigned distinct outbound and inbound UDP ports to avoid packet collisions or connection failures.

16.5.5 Ports Used

GCS	Outgoing Port (PX4)	Incoming Port (GCS)
MAVProxy	14550	14556
Mission Planner	14558	14561

This dual GCS architecture provides enhanced observability and control over UAV missions in simulated environments and can be extended for real hardware-in-the-loop setups as well.

16.6 Conclusion

The SITL setup with Mission Planner, MAVProxy, and PX4 SITL in Gazebo enables the simulation of UAV flight using multiple Ground Control Stations. The process involves setting up MAVProxy, connecting it to the PX4 SITL in Gazebo, and configuring Mission Planner as an additional GCS. The ability to connect multiple GCSs to the same SITL instance is useful for real-time monitoring and control during autonomous flight simulations.

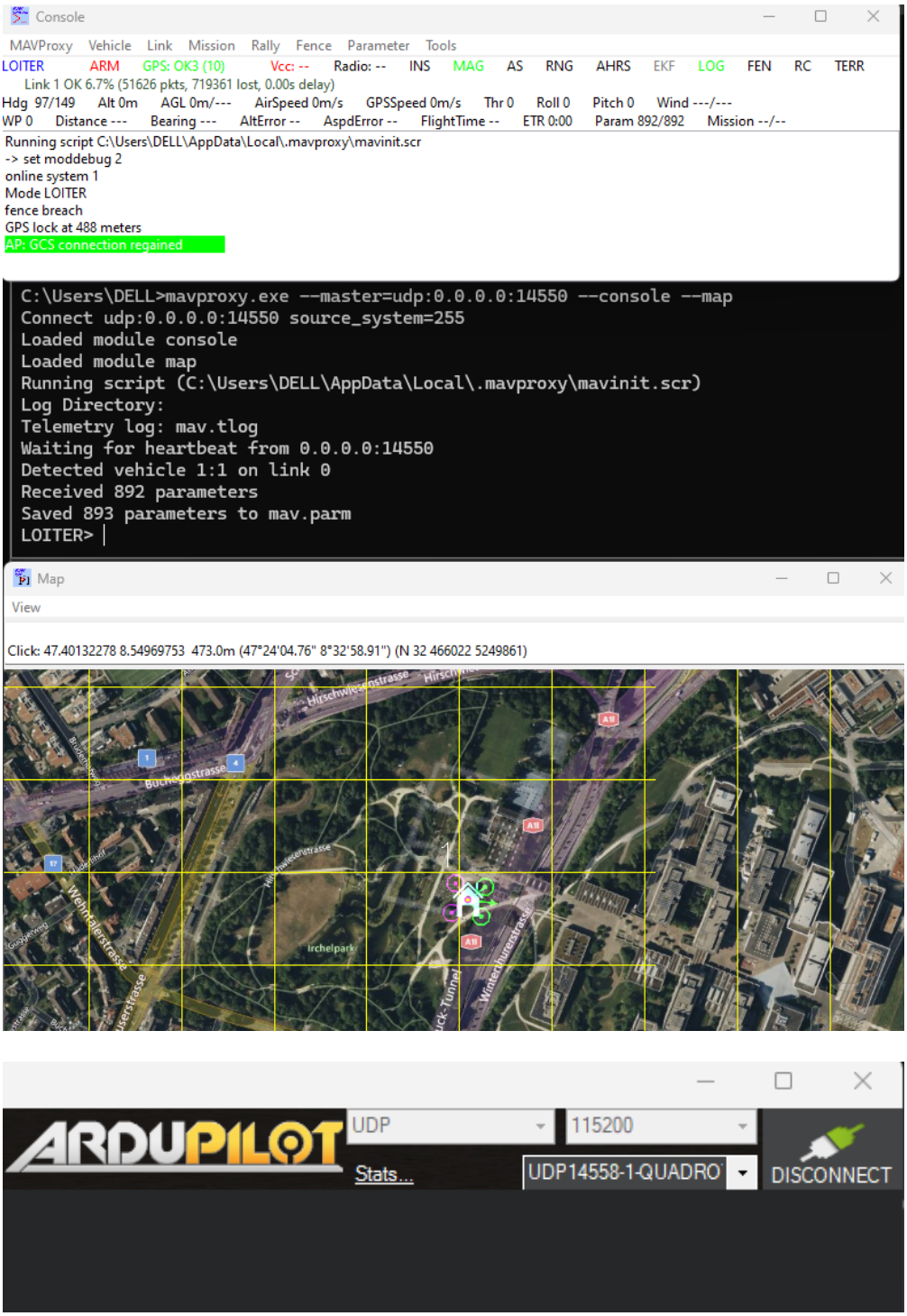


Figure 17: Mavproxy and Mission Planner Connections

```
wp list
LOITER> Requesting 4 waypoints t=Tue May  6 14:50:39 2025 now=Tue May  6 14:50:39 2025
16 6 47.3985816000 8.5468911000 100.000000 p1=0.0 p2=0.0 p3=0.0 p4=0.0 cur=1 auto=1
16 6 47.3985816000 8.5468911000 100.000000 p1=0.0 p2=0.0 p3=0.0 p4=0.0 cur=0 auto=1
16 6 47.3976084000 8.5473633000 100.000000 p1=0.0 p2=0.0 p3=0.0 p4=0.0 cur=0 auto=1
16 6 47.3965045000 8.5463333000 100.000000 p1=0.0 p2=0.0 p3=0.0 p4=0.0 cur=0 auto=1
Saved 4 waypoints to way.txt
Saved waypoints to way.txt
```

Figure 18: way.txt : Created by "Plan" in Mission Planner

16.7 Create or Upload a Mission File

We can manually create a waypoint mission using the Plan tab in Mission Planner and click "Write". The corresponding '.txt' file is saved in our home directory.

Alternatively, we can load a predefined mission file directly in MAVProxy:

```
wp load mission.txt
wp list
```

16.8 7. Valid GPS Range in Gazebo World

All waypoints should fall within the bounds computed from WGS84 projection:

- Latitude: `47.3977506 -- 47.3978000`
- Longitude: `8.545260 -- 8.545688`

Ensure altitude values are consistent with your simulated terrain and desired flight profile.

16.9 8. Executing the Mission

After loading the mission:

```
wp list
arm throttle
mode auto
```

This arms the UAV and starts the mission in Auto mode. The last command (code 20) is Return to Launch (RTL).

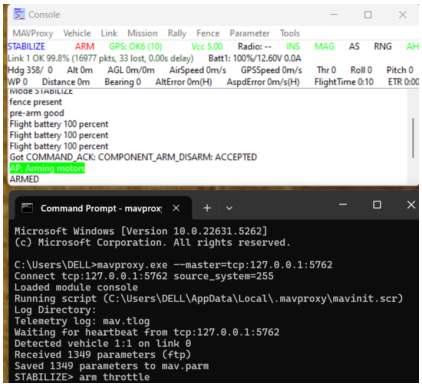


Figure 19: Mavproxy Console

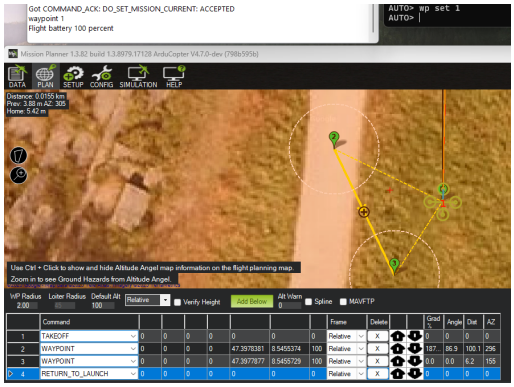


Figure 20: Waypoints

Figure 21: Mavproxy CLT and Mission Planning

17 Physical Drone Setup for Autonomous Flood Detection Mission

17.1 Overview

This section outlines the real-world deployment of an autonomous flood detection system using a UAV. The setup involves integration of physical hardware components (Pixhawk, Jetson Nano) with software modules (ROS 2 nodes, MAVROS, UNet model), configured to detect flooded regions and autonomously navigate to those areas.

17.2 Hardware Components

- **Pixhawk Flight Controller:** Controls low-level flight dynamics, executes waypoints, receives MAVROS commands.
- **Jetson Nano (NVIDIA Board):** Onboard edge device responsible for running ROS 2 nodes, flood segmentation using UNet, and GPS waypoint publishing.
- **Camera:** Mounted on the UAV, streams real-time images to the Jetson Nano.
- **Telemetry Module:** Enables communication between Jetson Nano and Ground Station (Mission Planner).

17.3 Software Nodes

17.3.1 1. Flood Segmentation Node

- **Subscribed Topic:** `/camera/image_raw`
- **Task:** Performs semantic segmentation on input images using a pretrained UNet model to detect flooded regions.
- **Output:** Segmentation mask is analyzed using a 4x4 grid to identify the cell with the highest flood density.

17.3.2 Execution

Navigate to ~/ayush/TEEP/src/flood_segmentation/flood_segmentation

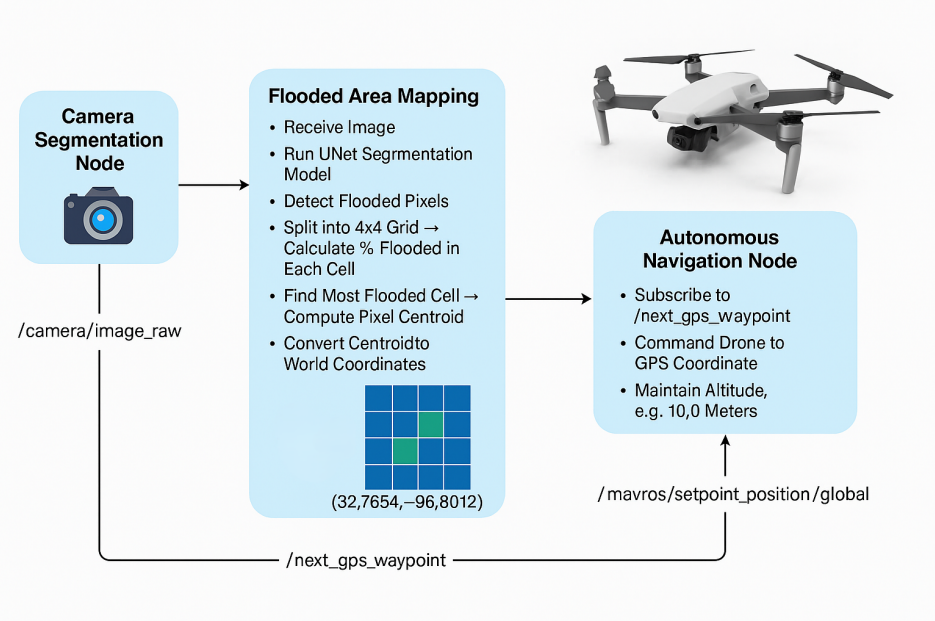


Figure 22: Setup of Jetson NX

```
cd ~/ayush/TEEP/src/flood_segmentation/flood_segmentation
python3 flood_seg_node.py
```

17.3.3 2. Autonomous GPS Navigation Node (Real Drone Setup)

- **Input:** Pixel centroid of the most flooded cell from segmentation mask.
- **Coordinate Transformations:**
  1. **Camera Intrinsics:** Convert pixel coordinates to normalized 3D ray in the camera frame using the inverse of the intrinsic matrix:

$$\mathbf{r}_{\text{cam}} = K^{-1} \cdot \begin{bmatrix} u_c \\ v_c \\ 1 \end{bmatrix}, \quad \text{where } K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

     The result is a unit vector pointing from the camera center toward the image plane.
  2. **Camera to World (Rotation):** Since the drone camera is pitched downward (typically $\theta = 45°$), apply the camera's rotation matrix to transform the ray into the world frame:

$$\mathbf{r}_{\text{world}} = R_{\text{pitch}} \cdot \mathbf{r}_{\text{cam}}, \quad R_{\text{pitch}} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

  3. **Ray-Ground Intersection:** Assuming a flat ground plane at $Z = 0$, and drone flying at altitude h (relative to the ground), scale the direction vector:

$$s = -\frac{h}{r_z}, \quad \mathbf{P}_{\text{ground}} = s \cdot \mathbf{r}_{\text{world}} = \begin{bmatrix} \Delta x \\ \Delta y \\ 0 \end{bmatrix}$$

     Here, $(\Delta x, \Delta y)$ are displacements in meters (East, North) from the drone's current GPS position.

4. **World to GPS:** Use GeographicLib to convert the local Cartesian displacement to global GPS coordinates. Given current drone GPS (ϕ_0, λ_0):

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}, \quad \alpha = \tan^{-1}\left(\frac{\Delta x}{\Delta y}\right)$$

$$(\phi, \lambda) = \text{Geodesic.Direct}(\phi_0, \lambda_0, \alpha, d)$$

This gives the precise GPS waypoint for the drone to autonomously navigate to the flooded region.

– **Waypoint Publication:** The computed target GPS coordinate is published to the MAVROS topic:

`/mavros/setpoint_position/global`

which allows the PX4 flight stack to initiate autonomous navigation to the detected flooded area.

17.4 Mission Planner Integration

– Mission Planner is used to:

1. Monitor telemetry and GPS status.
2. Visualize dynamic waypoint updates.
3. Ensure safety checks before arming.
4. Log the flight path for post-mission analysis.

17.5 Control Sequence

1. **Initial Random Waypoint:** A random GPS location is first published to initiate Offboard mode readiness.
2. **Set to OFFBOARD Mode:**

```
ros2 topic pub /mavros/set_mode mavros_msgs/msg/SetMode "
{custom_mode: 'OFFBOARD'}"
```

3. **Arm the Drone:**

```
ros2 service call /mavros/cmd/arming std_srvs/srv/SetBool "
{data: true}"
```

4. **Live Flood Detection:** Camera stream is analyzed frame-by-frame and a new GPS target is computed dynamically upon detection.
5. **Waypoint Update:** GPS of the flooded area is published to the UAV.

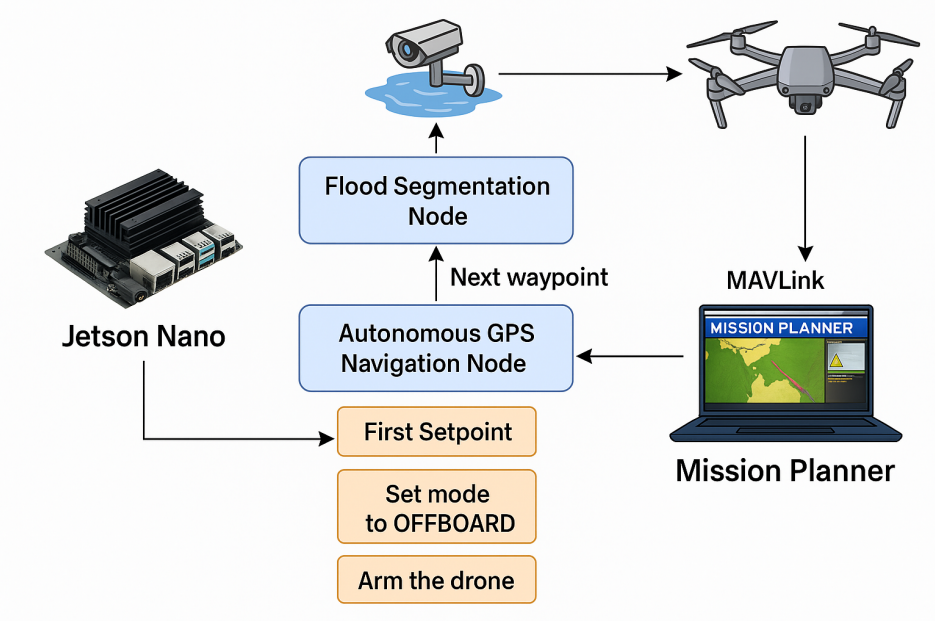


Figure 23: Communication Flow

18 Connection of Mission Planner and MAVProxy with Pixhawk (PX4 firmware)

In UAV systems using the PX4 firmware and Pixhawk flight controller, it is often beneficial to connect both Mission Planner and MAVProxy simultaneously for enhanced ground control and debugging capabilities. Mission Planner provides a user-friendly graphical interface for flight planning, sensor calibration, and real-time telemetry, making it ideal for operational oversight. MAVProxy, on the other hand, is a powerful command-line ground control station (GCS) that supports scripting, automation, and advanced diagnostics. Connecting both tools to the drone enables a flexible workflow where Mission Planner handles mission management while MAVProxy can run real-time commands or Python modules for autonomous behavior testing, logging, or fault handling. However, due to COM port limitations, this dual connection requires UDP-based MAVLink forwarding setup on the Pixhawk.

18.1 One-time Setup: Forward MAVLink via UDP

1. **Remove the microSD card from the Pixhawk** and insert it into your PC using a card reader.
2. **Create directory structure if not present:**
   - Navigate to the root of the SD card and create the file:

     `/etc/extras.txt`
   - On Windows, this may appear as:

     `D:\etc\extras.txt`
3. **Edit the `extras.txt` file** and add the following line:

   ```
   mavlink start -m onboard -t 192.168.24.28
   -u 14555 -o 14550
   ```

   - Replace `192.168.24.28` with your GCS (laptop) IP address.
4. **Safely eject the SD card and reinsert it into the Pixhawk.**
5. **Reboot the Pixhawk.**

18.2 Connect using MAVProxy (via UDP)

On your PC, start MAVProxy using:

```
mavproxy.exe --master=udp:0.0.0.0:14555 --console --map
```

18.3 Alternative: Connect MAVProxy via USB COM Port

MAVProxy can also connect directly via USB, just like Mission Planner:

```
mavproxy.exe --master=COM17 --baudrate=57600
--console --map
```

- Note: Only one application can access a COM port at a time.
- If you attempt to run both Mission Planner and MAVProxy simultaneously using USB, one will fail with an "Access Denied" error.
- For simultaneous use, use the UDP forwarding setup described above.

```
C:\Users\DELL>mavproxy.exe --master=COM17 --baudrate=57600 --console --map
Connect COM17 source_system=255
Failed to connect to COM17 : could not open port 'COM17': PermissionError(13, 'Access is denied.', None, 5)
```

Figure 24: Problem when multiple applications try to access the COM port simultaneously

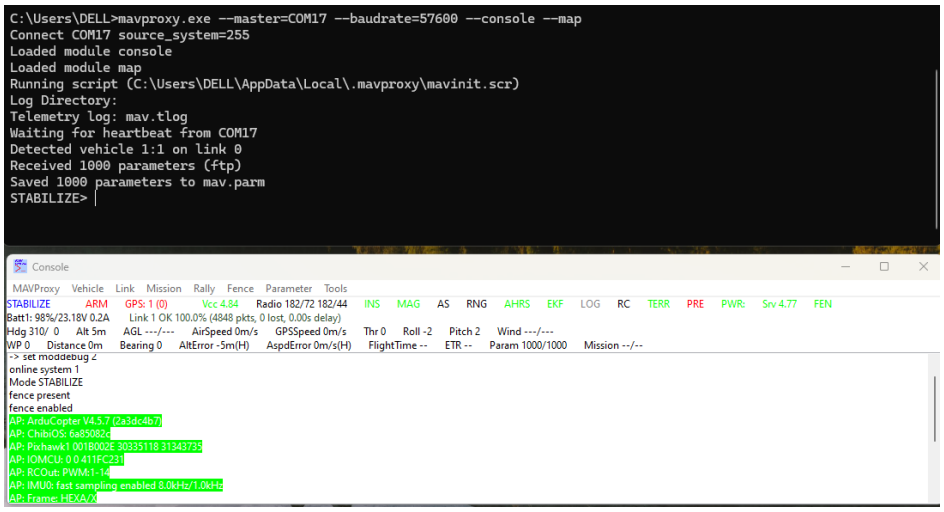


Figure 25: Successful MAVProxy connection via COM Port

19 Demonstration of Physical Drone with Mission Planner

A physical flight test of the UAV was successfully carried out on the university ground using the Mission Planner ground control station (GCS). The drone was programmed with a predefined mission consisting of a sequence of waypoints, which included a `TAKEOFF` command, multiple `WAYPOINT` commands, and a `RETURN_TO_LAUNCH` command. The mission plan is illustrated in Figure 26, where the waypoints and flight path are marked.

The drone was connected to the Mission Planner software via a telemetry radio module through a designated COM port. The telemetry connection parameters were:

– **COM Port:** COM5 (or as detected by the system)
– **Baud Rate:** 57600 bps

Upon establishing the connection, Mission Planner successfully uploaded the mission commands to the drone's flight controller. The flight was initiated with the `ARM` and `TAKEOFF` commands, and the drone autonomously navigated through each waypoint at a relative altitude of 5 meters.

The home location, where the drone was initially armed and launched, is shown below:

– **Latitude:** 23.5627971
– **Longitude:** 120.4776742
– **Altitude (ASL):** 106.29 meters

Prior to the flight, a complete PreFlight setup was carried out (Figure 27a). The checklist included:

– GPS lock with 15 satellites
– Battery voltage at 23.67 V
– 99% telemetry signal strength
– GPS type: 3D fix
– Mode set to `RTL` (Return to Launch)
– All safety and control checks verified

The mission concluded with the drone safely returning to the launch position, demonstrating effective autonomous control in real-world outdoor conditions.

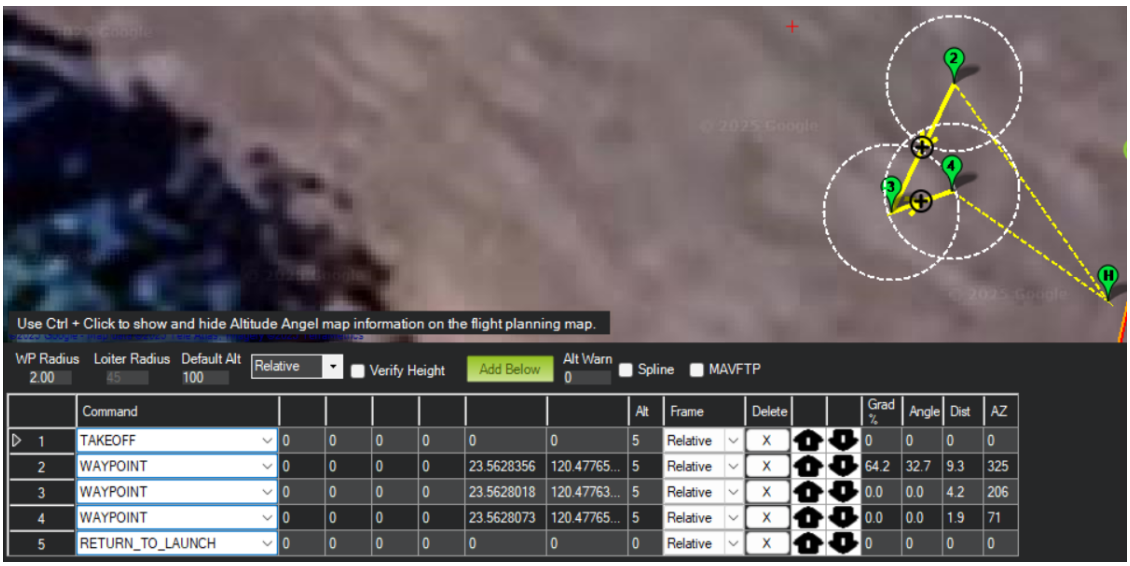
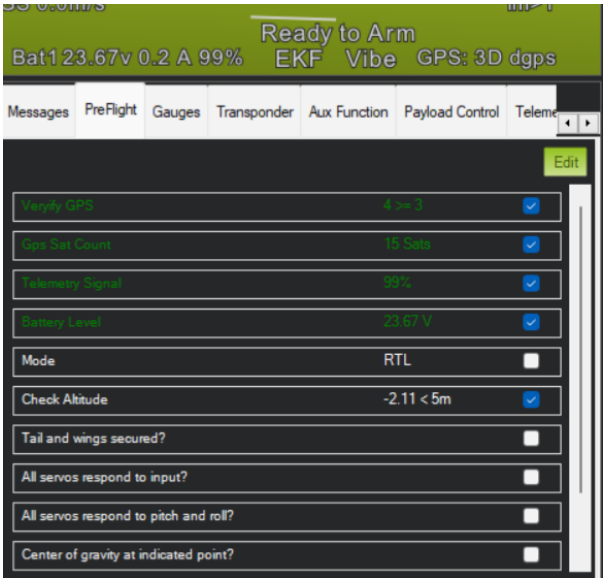


Figure 26: Waypoint mission plan executed using Mission Planner



(a) PreFlight Checklist and Setup Verification in Mission Planner

(b) Fly view

Figure 27: Preflight setup and mission execution

20 Semantic Segmentation of Real-World Flooded Regions from UAV Imagery Using DeepLabV3+ with MobileNetV3

Overview

This project demonstrates the segmentation of flood-affected regions using a real-world drone-view dataset. A semantic segmentation pipeline was built using DeepLabV3+ with MobileNetV3-Large as the backbone in PyTorch. The model is trained, evaluated, and tested on aerial images to identify water-inundated areas.

The workflow comprises the following stages:

- Dataset preparation and preprocessing
- Model architecture and configuration
- Training procedure
- Evaluation and metric analysis
- Testing on individual drone-view images
- Visual result generation

20.1 Dataset Details

- **Source:** Custom drone-view flood dataset from Kaggle (`https://www.kaggle.com/datasets/armaanoajay/flooded-images`)
- **Annotation Tool:** Annotated manually using the open-source tool CVAT (Computer Vision Annotation Tool) for semantic segmentation.
- **Image folder:** `JPEGImages/`
- **Mask folder:** `SegmentationClass/`

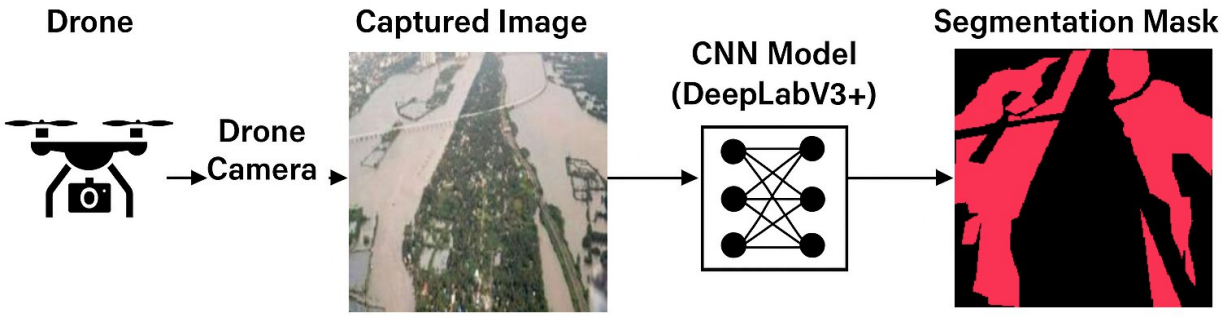


Figure 28: Framework

- **Image Resolution:** Resized to 256×256
- **Classes:**
  - Class 0 – Non-Flooded Region
  - Class 1 – Flooded Region

20.2 Preprocessing Pipeline

- Resized all images and masks to 256×256
- Converted to tensor format using PyTorch's 'transforms'
- Mask resizing used nearest-neighbor interpolation
- Augmentations applied during training (horizontal and vertical flips)

20.3 Model Architecture

- **Segmentation Model:** DeepLabV3+
- **Backbone:** MobileNetV3-Large (lightweight, efficient)
- **Framework:** PyTorch (torchvision library)
- **Output:** Per-pixel binary classification mask (Flood vs Non-Flood)

20.4 Training Details

- **Loss Function:** CrossEntropyLoss
- **Optimizer:** Adam
- **Learning Rate:** 1×10^{-4}
- **Epochs:** 10
- **Batch Size:** 4
- **Split:** 80% training and 20% testing using 'random_split()'

20.5 Evaluation Procedure

- Model tested on 30 unseen drone-view images.
- All predictions compared to ground truth masks.
- Metrics calculated:
  * Intersection over Union (IoU)
  * Dice Coefficient
  * Pixel-wise Accuracy
  * Precision and Recall
- Confusion matrix generated and saved as a heatmap.

20.6 Evaluation Metrics

Table 3: Final Test Results on Drone-View Dataset

Metric	**Value**
IoU (Jaccard Index)	0.7327
Dice Coefficient	0.8457
Accuracy	0.8926
Precision	0.8123
Recall	0.8821

20.7 Confusion Matrix

20.8 Confusion Matrix Analysis

The confusion matrix provides a pixel-wise comparison between the predicted segmentation mask and the ground truth mask, measuring how well the model distinguishes between flooded and non-flooded regions.

The confusion matrix is as follows:

	Predicted Non-Flood	**Predicted Flood**
Actual Non-Flood	1,176,318 (TN)	133,755 (FP)
Actual Flood	77,351 (FN)	578,656 (TP)

Where:

- **True Positive (TP)**: Pixels correctly predicted as flooded — 578,656
- **True Negative (TN)**: Pixels correctly predicted as non-flooded — 1,176,318
- **False Positive (FP)**: Pixels incorrectly predicted as flooded — 133,755
- **False Negative (FN)**: Pixels incorrectly predicted as non-flooded — 77,351

20.8.1 Metric Derivations

From the confusion matrix, we derive several key performance metrics to evaluate the quality of segmentation:

– **Precision (Flood Class)**
Precision measures how many of the pixels predicted as flooded are actually flooded. It helps assess false alarm rate.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} = 0.8123$$

Interpretation: Among all pixels predicted to be flooded, approximately 81.23% were correctly identified. The remaining 18.77% were false alarms.

– **Recall (Flood Class)**
Recall measures how many of the actual flooded pixels were successfully identified. It captures the model's ability to detect floods.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = 0.8821$$

Interpretation: Out of all actual flooded pixels, about 88.21% were correctly identified, indicating a strong ability to detect inundated regions.

– **Accuracy**
Accuracy measures the overall correctness of classification, combining both flooded and non-flooded predictions.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Pixels}} = \frac{1,754,974}{1,966,080} = 0.8926$$

Interpretation: Overall, the model correctly labeled about 89.26% of all pixels, reflecting strong performance across both classes.

– **Intersection over Union (IoU)**
IoU is a core metric in segmentation tasks. It measures the overlap between the predicted flood region and the actual ground truth flood region.

$$\text{IoU} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives} + \text{False Negatives}} = \frac{578,656}{789,762} = 0.7327$$

Interpretation: The predicted flood masks overlap with the ground truth by approximately 73.27%, a strong indication of accurate segmentation.

– **Dice Coefficient (F1-Score in Segmentation)**
Dice Score emphasizes overlap between predicted and ground truth regions and is especially useful for imbalanced data.

$$\text{Dice} = \frac{2 \times \text{True Positives}}{2 \times \text{True Positives} + \text{False Positives} + \text{False Negatives}} = 0.8457$$

Interpretation: Dice Coefficient of 84.57% confirms high-quality overlap between predicted and actual flood regions.

These derived metrics confirm the robustness of the trained DeepLabV3+ model with MobileNetV3 backbone. The model performs well in accurately segmenting flood-affected regions in real drone imagery, achieving a balance between detection sensitivity (recall) and prediction confidence (precision).

20.8.2 Interpretation

- The model shows a strong ability to correctly detect flooded areas (high TP and Recall).
- A moderate number of false positives (FP) indicates that a few dry regions were misclassified as flooded.
- High Dice and IoU scores confirm that the overlap between prediction and ground truth is substantial.
- The balance between Precision and Recall shows the model is robust and suitable for deployment in safety-critical drone-based applications.

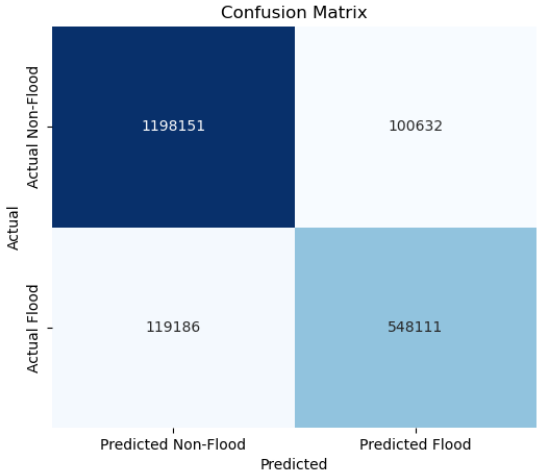


Figure 29: Pixel-wise Confusion Matrix

20.9 Single Image Testing

- The file `test_image.py` allows isolated testing of new drone images.
- Input: `test1.jpg`
- Output visualizations:
  - Input Image
  - Ground Truth (if available)
  - Predicted Mask
  - Overlay Visualization

The 'test_image.py' script enables single-image evaluation of the segmentation model on newly captured drone images. For each test image (e.g., 'test1.jpg'), it generates a set of visual outputs including the original input image, the predicted segmentation mask, and an overlay visualization that highlights the detected flooded regions. Below are examples for three test images, each showing the input, predicted mask, and overlay side by side to illustrate the model's performance in isolating flood-affected areas.

Left: Input Image **Center:** Predicted Mask **Right:** Overlay

Left: Input Image **Center:** Predicted Mask **Right:** Overlay

Left: Input Image **Center:** Predicted Mask **Right:** Overlay

20.10 Directory Structure

```
TEEP/
|-- src/
|   +-- Flood_Detect/
|       |-- JPEGImages/              # Input images
|       |-- SegmentationClass/      # Ground truth masks
|       +-- flood_segmentation/
|           |-- train.py             # Training script
|           |-- test.py              # Evaluation script
|           |-- test_image.py        # Single image testing
|           +-- outputs/             # Saved predictions and metrics
```

20.11 Conclusion

- This project achieved high segmentation performance on drone-view images of flood-affected regions using DeepLabV3+ with MobileNetV3 backbone.
- The model accurately distinguished between flooded and non-flooded areas, as demonstrated by high IoU, Dice, and Recall scores.
- The pipeline is modular and optimized for real-time applications.
- **Ready to deploy the system on a real UAV platform for autonomous disaster response.**