ENAE788M - Final Project

Piyush Goenka Yi-Hsuan Chen

1 Introduction

This project aims to detect AprilTag positions and use a planner to generate the drone's trajectory. We use multiple AprilTags to provide waypoints for the drone, leveraging the tracking camera's video feed for detection. The detected poses are transformed into the drone's body and inertial frames, generating accurate waypoints. The Fast-Planner, developed by the HKUST Aerial Robotics Group, is used to plan a safe, kinodynamically feasible trajectory based on these waypoints. We integrate Fast-Planner with PX4 offboard control using ROS1 bridge, providing a method to utilize this ROS1 project for efficient and robust drone replanning in complex environments under the ROS2 framework.

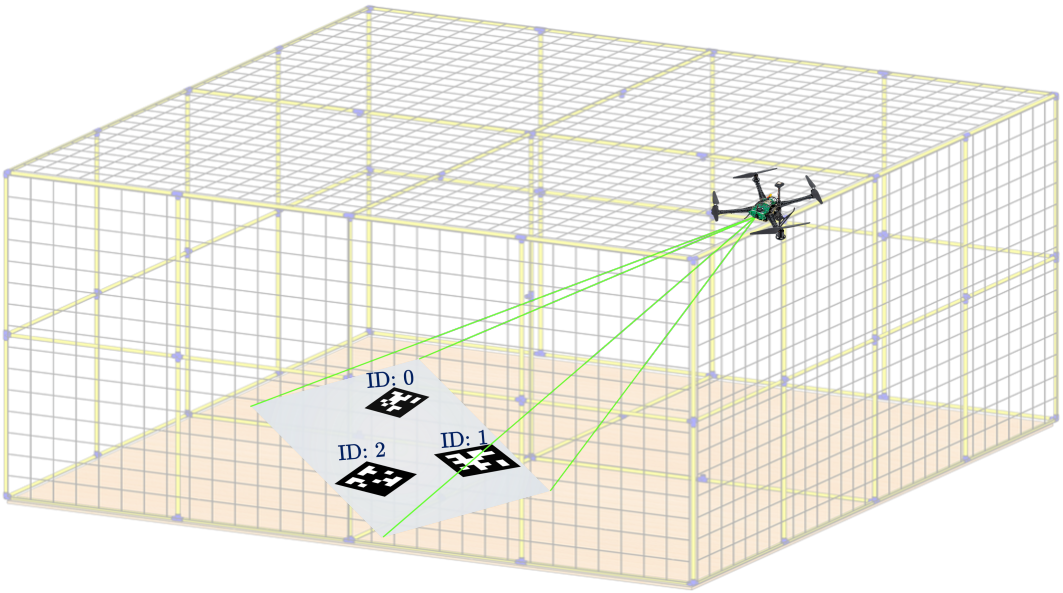

Figure 1: Illustration of multiple AprilTags detection.

2 Visual Detection - Multiple AprilTags Localization

The goal of the Visual Detection module is to provide waypoints to the path planner to move the drone to specific positions. In our case, we employ the use of three apriltags with id 0, 1 and 2 placed on the ground to provide the x and y positions for our drone's waypoints.

2.1 Apriltag Detection

In order to detect apriltags on the ground, we use the tracking camera's video feed as seen in Fig. 2. A ROS2 node `voxl_mpa_to_ros2_node` from the package `voxl_mpa_to_ros2` performs the April-

Tag detection and publishes their pose w.r.t the camera frame in the topic `/tag_detectionss/tagpose` with the AprilTag id being populated in the header's frame_id field of the `PoseStamped` message.

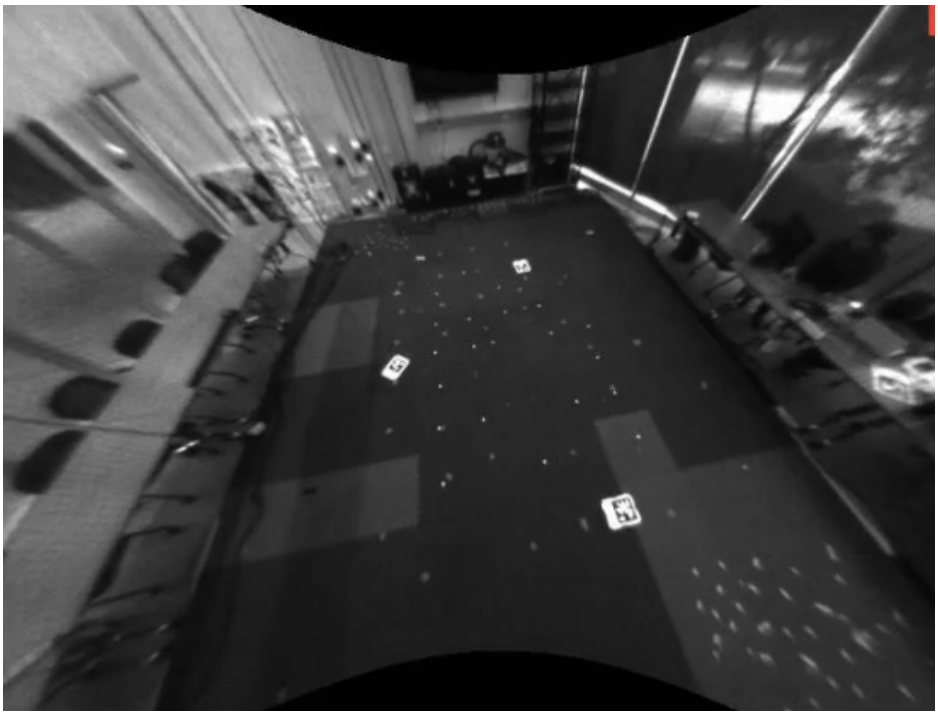

Figure 2: Tracking camera video feed

2.2 Apriltag Localization

To obtain the AprilTag poses in the inertial frame, we run the ros2 node `apriltag_localize_node` from the ROS2 package `apriltag_localize`. This node transforms the AprilTag poses from the tracking camera's frame to the drone's body frame and the inertial frame. The ROS2 topic `tag_detections/tagpose_inertial` gives us the AprilTag poses w.r.t the inertial frame with the AprilTag id being populated in the header's `frame_id` field of the`PoseStamped` message.

2.3 Waypoint Generation

During the flight, we first manually hover over each AprilTag to record their positions and then publish the waypoints one by one in the topic `/waypoint`. We use a ROS2 node `generate_waypoints` which subscribes to the `tag_detections/tagpose_inertial` topic and saves the position of each detected AprilTag during manual control. Once we obtain the positions of all three AprilTags, the node publishes a waypoint, in our case it's the position information of `apriltag_0`. At this time, we switch the drone to offboard mode. The node continuously monitors the position of the drone by subscribing to the topic `/fmu/out/vehicle_local_position`. Once the drone's x-y position is within 0.2m of the waypoint's x-y position, the node then publishes the next waypoint. This continues indefinitely and the drone moves over each waypoint in a continuous loop.

3 Path Planning — Fast-Planner Implementation

The objective of path planning part is to find a safe, kinodynamic feasible and minimum-time initial trajectory based on the detected AprilTags locations as waypoints. The planner we are using here is the Fast-Planner, developed by the HKUST Aerial Robotics Group, aiming to enable quadrotor fast flight in complex unknown environments [1]. The original Fast-Planner project contains three planning algorithms: Kinodynamic path searching, B-spline-based trajectory optimization, Topological path searching, and path-guided optimization. In this project, we only use the feature of **kinodynamic path searching** and **B-spline Optimization**, which can be run by `roslaunch plan_manage kino_replan.launch`.

The Fast-Planner is developed and tested on Ubuntu 16.04 (ROS1 Kinetic) and 18.04 (ROS1 Melodic), yet our visual detection part is running on ROS2 Foxy. Thus, instead of migrating the source code of an existing package from ROS1 to ROS2, we use the ROS2 package `ros1_bridge` [2] to bridge topics between ROS1 and ROS2. We started by setting up the Fast-Planner in ROS2 and integrating it with PX4 offboard control in the Gazebo simulation.

3.1 Framework of Fast-Planner

We first used the rqt_graph tool to examine the workflow of Fast-Planner. As seen in Fig. 3, the original project contains perception, estimation, planning, control, simulation, and visualization components.

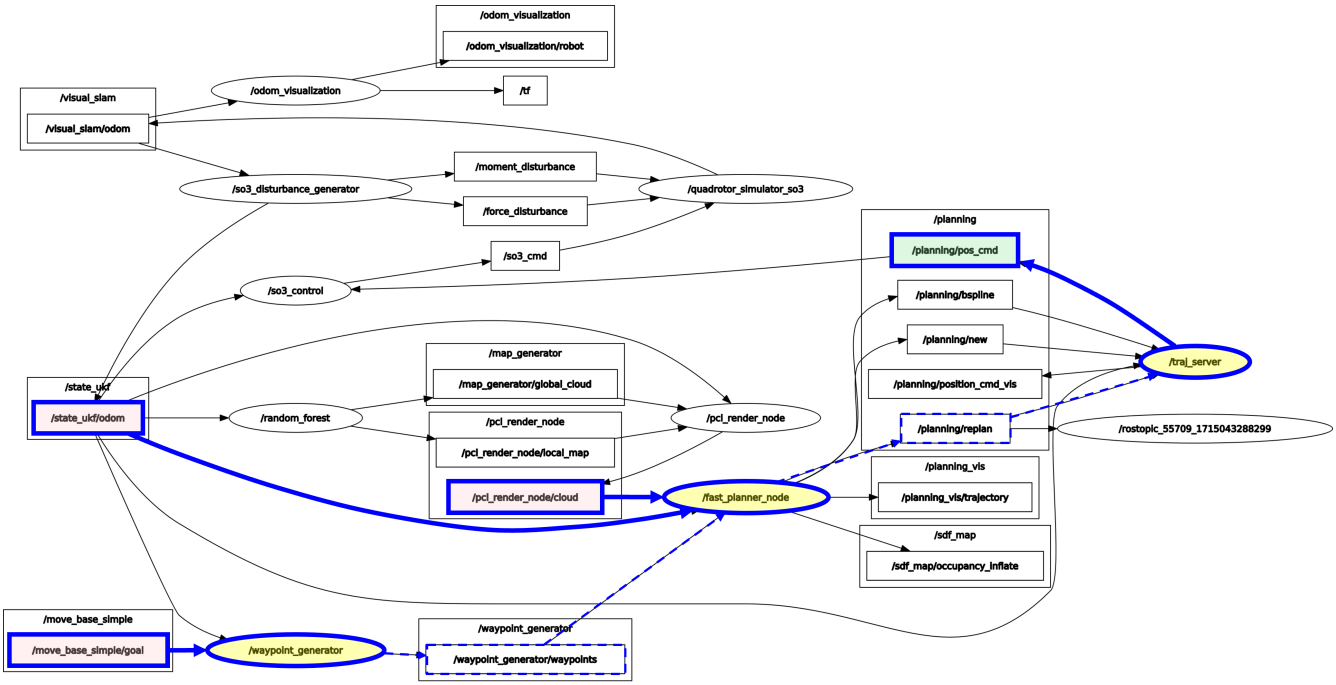

Figure 3: The framework of Fast-Planner, integrated with PX4 offboard control in a Gazebo simulation in ROS2. Planning-related nodes are shaded in yellow, topics to be replaced are shaded in pink, and the output topic is shaded in green.

However, in our case, PX4 flight stack will handle the perception, estimation, and control, and Gazebo will be used for simulation and visualization. Therefore, we only need to transmit data

(vehicle odometry, goal points, and point cloud or depth image) for the `/fast_planner_node` to engage the path planning and trajectory generation node `/traj_server` to publish the generated trajectory to the topic `'/planning/pos_cmd'`.

3.2 Prerequisites - Environment Setup and Configuration

In order to bridge ROS1 and ROS2, we need to install both of them on the same operating system, and here we use Docker to run Ubuntu 20.04 in the docker container, and the steps to build `ros1_bridge` can be found in Appendix 1. One issue we faced when building the bridge was that some messages could not be matched, such as `'ActuatorArmed'`, `'ActuatorOutput'`, `'EstimatorStatus'`, `'Airspeed'`, `'EstimatorStatusFlags'` etc, and delete these problematic messages on ROS2 will resolve this. Additionally, since Fast-Planner is officially developed only for ROS Kinetic and Melodic, we need to make a few modifications to migrate it to Ubuntu 20.04 with ROS Noetic, refer to Appendix 2.

3.3 Integration of Fast-Planner, PX4, and Gazebo in ROS2

After setting up the environment and building the ROS1 bridge and Fast-Planner packages in the docker container, we need a few Python scripts to properly transmit data between Fast-Planner in ROS1 and PX4 and Gazebo in ROS2. The integration framework is shown in Fig. 4. In this setup, we feed odometry and point cloud data from Gazebo, and the goal point to the Fast-Planner. The Fast-Planner will then publish the topic `'planning/pos_cmd'`, which we need to transform from ENU (Fast-Planner) to NED (PX4) and remap the message type for a new topic `'fast_planner/pos_cmd'` to pass through the ROS1 bridge.

For PX4 control part, we use a ROS2 Python package `px4_offboard_py` to subscribe the topic `'/fast_planner/pos_cmd'` and publish the topics of `'/fmu/in/trajectory_setpoint'` and `'/fmu/in/offboard_control_mode'` to enable the PX4 offboard control. The message types of the above mentioned topics are listed in 3.3.1.

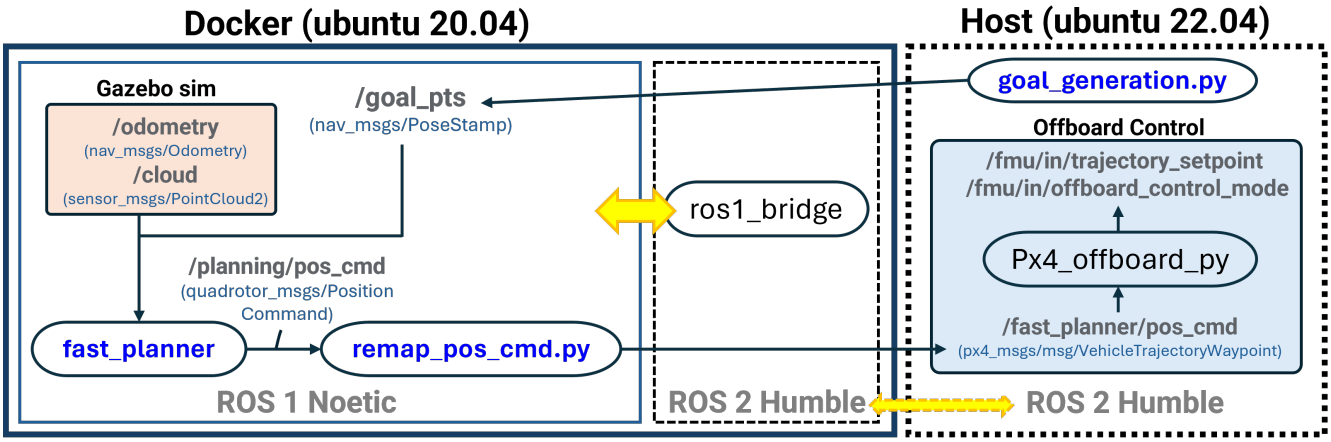

Figure 4: The framework of Fast-Planner with PX4 flight control in Gazebo Simulation in ROS2.

3.3.1 List of Data Transmission

The Python scripts that remap the message type and publish the new topics are listed here:

- `remap_odometry.py`: Remaps `'/fmu/out/vehicle_odometry'` (`px4_msgs/VehicleOdometry`) from NED in PX4 (ROS2) to `'/odometry'` (`nav_msgs/Odometry`) for ENU in Fast-Planner (ROS1).
- `remap_pos_cmd.py`: Remaps `'planning/pos_cmd'` (`quadrotor_msgs/PositionCommand`) from ENU in Fast-Planner (ROS1) to `'/fast_planner/pos_cmd'` (`px4_msgs/VehicleWaypoint`) for NED in PX4 (ROS2).
- `px4_offboard_py`: Remaps `'/fast_planner/pos_cmd'` (`px4_msgs/VehicleWaypoint`) to `'/fmu/in/trajectory_setpoint'` (`px4_msgs/TrajectorySetpoint`). Only message type transform is involved since they are both defined in PX4 NED coordinates.

We also need to modify the `bridg.yaml` file used by the `ros_gz` package [3] to bridge the point cloud data from Gazebo to ROS2.

```
- topic_name: "depth_camera/points"
  ros_type_name: "sensor_msgs/msg/PointCloud2"
  gz_type_name: "gz.msgs.PointCloudPacked"
```

3.3.2 Modification of `launch` file

The last step for running the Fast-Planner in Gazebo simulation is to change the subscribing topics:

```
<arg name="odom_topic" value="/odometry" />
```

```
<arg name="cloud_topic" value="depth_camera/points"/>
```

```
<remap from="~goal" to="/goal_pts"/>
```

3.4 Gazebo Simulation

We use a ROS2 node `waypoint_publisher` to publish the waypoint to the topic (`'/goal_pts'`), and the subsequent waypoint is only published when the drone reaches the previous one. The simulation result shows that the drone can execute the trajectory generated by Fast-Planner and has hit 10 waypoints successfully, see Fig. 5.

3.5 Voxl2 Drone Flight Test

For the Voxl2 drone setup, we followed the same framework of using a docker container to bridge ROS1 and ROS2 topics. The only difference between Gazebo simulation and real drone implementation is odometry and point cloud topics. In our case, we use `'/qvio'` and `'/voa_pc_out'`.

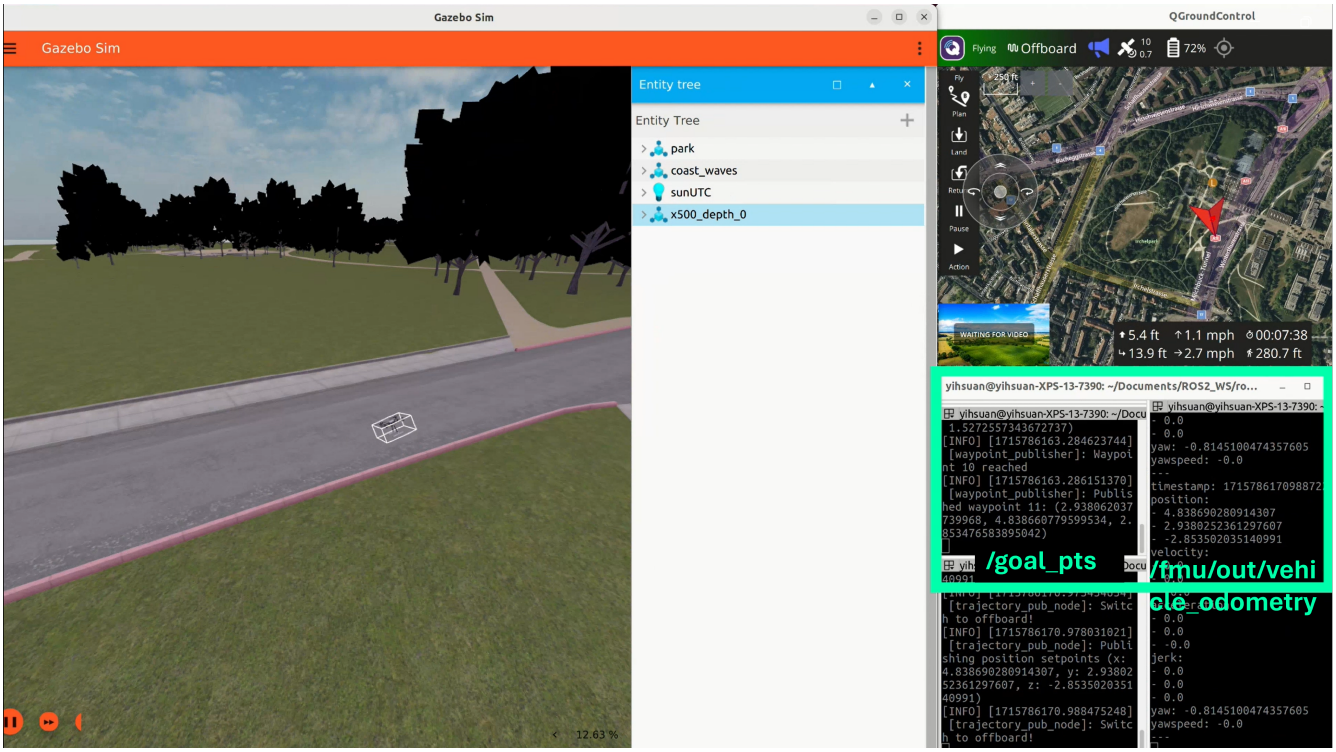

Figure 5: Gazebo Simulation of Fast-Planner with PX4 flight control in ROS 2. Note that the `'/goal_pts'` is defined in ENU, while `'/fmu/out/vehicle_odometry' is in NED`.

3.5.1 Terminal Setup Using Shell scripts

To run the Fast-Planner-PX4-Offboard framework on the drone, we need to use 7 terminals on the drone — 4 in the docker container for running `roscore`, `ros1_bridge`, Fast-Planner, and `remap_pos_cmd.py`, 3 in the drone host for running `voxl_mpa_to_ros2` , `remap_odometry.py`, and `px4_offboard_py`, as shown in Listing 3.5.1.

```
# t1_roscore.sh
#!/bin/bash
source /opt/ros/noetic/setup.bash
roscore

# t2_pub_pos_cmd.sh
#!/bin/bash
source /opt/ros/noetic/setup.bash
source ros1_px4_msgs_ws/devel/setup.bash
source ros1_fast_planner_ws/devel/setup.bash
cd ros1_fast_planner_ws
python3 remap_pos_cmd.py

# t3_fast_planner.sh
#!/bin/bash
source /opt/ros/noetic/setup.bash
source ros1_fast_planner_ws/devel/setup.bash
roslaunch plan_manage voxl_replan.launch
```

```
# t4_ros1_bridge.sh
#!/bin/bash
source /opt/ros/noetic/setup.bash
source ros1_bridge_ws/install/setup.bash
export ROS_MASTER_URI=http://localhost:11311
ros2 run ros1_bridge dynamic_bridge

# t5_voxl_mpa.sh
#!/bin/bash
ros2 run voxl_mpa_to_ros2 voxl_mpa_to_ros2_node

# t6_remap_qvio.sh
#!/bin/bash
python3 remap_qvio.py

# t7_offboard.sh
#!/bin/bash
source ros2_offboard_ws/install/setup.bash
ros2 run px4_offboard_py offboard
```

Listing 1: Shell scripts for drone setup

3.6 Discussion

In the real drone flight test, we first used a method similar to the simulation to publish a random waypoint every 30 seconds. It seemed like the drone was executing some trajectories, but we were not sure if it reached the published waypoint since we did not record the data. We also tried to use the Fast-Planner to track the waypoints associated with the AprilTags; however, it did not work as expected.

Given that the Fast-Planner-PX4-Offboard framework works well in Gazebo simulation, we think the issue may arise from the inaccuracy of either vehicle odometry or point cloud data. We found that the drone was shooting up once we switched to offboard mode, even though the height of the received position setpoint was set to 0.5. We also encountered the error of `Shot in first search loop!` and the warning of `[optimization]: nlopt exception` frequently when running on the real drone. More inspections of the Fast-Planner source code are needed to identify why this error occurred. Another issue that might have caused the drone to not act properly could be the data transmission delay due to the use of the ROS1 bridge. More flight tests are required to determine which component of the Fast-Planner implementation is not properly set.

Another concern is that the Fast-Planner is mainly designed for efficient and fast replanning, which includes kinodynamic path search and B-spline optimization. For our scenarios of having static AprilTags and visiting them one by one, using only B-spline optimization should be more than enough for this task.

References

[1] Boyu Zhou, Shaojie Shen, and Fei Gao. A robust and efficient trajectory planner for quadrotors, 2020. https://github.com/HKUST-Aerial-Robotics/Fast-Planner.

[2] Nilaos. Ros 2 package that provides bidirectional communication between ros 1 and ros 2, 2023. https://github.com/ros2/ros1_bridge.

[3] Benjamin Perseghetti. Integration between ros (1 and 2) and gazebo simulation, 2019. https://github.com/gazebosim/ros_gz.

[4] Isaac Saito. Ubuntu install of ros noetic, 2023. https://wiki.ros.org/noetic/Installation/Ubuntu.

[5] Nilaos. Documentation on how ros 1 and ros 2 interfaces are associated with each other., 2023. https://github.com/ros2/ros1_bridge/blob/master/doc/index.rst.

[6] Steven G. Johnson. Library for nonlinear optimization, wrapping many algorithms for global and local, constrained or unconstrained, optimization, 2021. https://github.com/stevengj/nlopt.

[7] Elia Bonetto. Migrating to ubuntu 20.04/noetic, 2021. https://github.com/HKUST-Aerial-Robotics/Fast-Planner/issues/85.

[8] EdLou. Ubuntu 20 segmentation fault, 2023. https://github.com/HKUST-Aerial-Robotics/Fast-Planner/issues/117.

Appendix

1 Building ROS 1 bridge

Pull an Ubuntu 20.04 docker image and install ROS1 from the packages [4] and ROS2 from source. Note that we do not need to use `sudo` for installations since we are running as the root user in a docker container.

The example workspace of the bridge for `px4_msgs` is shown in Fig. 6. Once the workspace and packages are set up correctly, follow the quick guide in Listing. 3.6 to build the bridge. See [5] for more details on how ROS 1 and ROS 2 interfaces are associated with each other.

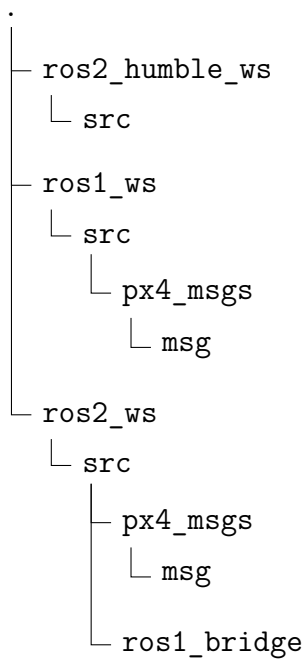

Figure 6: Directory Layout

```
# Shell 1 (ROS 1 only):
source /opt/ros/neotic/setup.bash
cd ros1_ws
catkin_make

# Shell 2 (ROS 2 only):
# source ROS 2 humble
source ros2_humble_ws/install/setup.bash
cd ros2_ws
colcon build --packages-select px4_msgs

# Shell 3 (ROS 1 + ROS 2):
# source ROS 1
source ros1_ws/devel/setup.bash
# source ROS 2
source ros2_ws/install/setup.bash
colcon build --packages-select ros1_bridge
```

Listing 2: Build ROS2 humble, PX4 in ROS1 and ROS2, and then ROS1 bridge

2 Migrate the Fast-Planner to Ubuntu 20.04

1. Install the dependency NLopt package [6], used to solve the nonlinear optimization problem, from source by the following commands (since using `apt install` on Ubuntu 20.04 may cause issues):

```
    sudo apt install libarmadillo-dev
    git clone https://github.com/stevengj/nlopt.git
    mkdir build && cd build
    cmake ..
    make
    sudo make install
```

2. Specify the C++ version standard as C++14. Add the following code after the project line in each CMakeLists.txt file.

```
    set(CMAKE_CXX_STANDARD 14)
```

3. Modify the `CMakeLists.txt` in `bspline_opt` to find the `nlopt` installed from source, since it is looking for `ros-noetic-nlopt` (issue #85) [7].

```
cmake_minimum_required(VERSION 2.8.3)
project(bspline_opt)

set(CMAKE_BUILD_TYPE "Release")
set(CMAKE_CXX_FLAGS "-std=c++11")
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -Wall")

find_package(Eigen3 REQUIRED)
find_package(PCL 1.7 REQUIRED)
find_package(nlopt)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  visualization_msgs
  plan_env
  active_perception
  cv_bridge
)

catkin_package(
 INCLUDE_DIRS include
 LIBRARIES bspline_opt
 CATKIN_DEPENDS plan_env active_perception
#  DEPENDS system_lib
)

include_directories(
    SYSTEM
    include
```

```
    ${catkin_INCLUDE_DIRS}
    ${Eigen3_INCLUDE_DIRS}
    ${PCL_INCLUDE_DIRS}
    ${nlopt_INCLUDE_DIRS}
)

add_library( bspline_opt
    src/bspline_optimizer.cpp
    )
target_link_libraries( bspline_opt
    ${catkin_LIBRARIES}
    nlopt
    )
```

4. Fix the return values of some functions (issue #117)[8]