POLITECNICO DI BARI

DEI

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Mobile Robotics

Prof. Ing. Luca De Cicco

Progetto:

Pose Control of a Drone in a Known and Unknown Virtual Environment

Studenti:
Marcello Bari
Alessandro Mitola

ANNO ACCADEMICO 2023-2024

Contents

1 Introduction

The goal of the assigned project is to develop multiple control loops that enable the movement of a drone in a virtual environment under different assumptions.
The fundamental tools for this work are, of course, ROS and CARLA, an open-source simulator with a client-server architecture designed for road traffic simulations. CARLA was chosen for its immediate availability of virtual environments in which the drone can operate. A "plug-in," the "carla-ros-bridge," was employed to allow both software frameworks to work in parallel. The remaining software tools will be introduced during the presentation when their respective purposes are explained.
The analyzed scenarios, for which the control systems have been designed, are as follows:

- **Pre-explored map**: The drone's position is known through a GPS system, considered "safe" apart from negligible noise, and a predefined trajectory is provided for it to follow.
- **Unexplored map with altitude constraints**: The drone's altitude cannot be varied, and its pose must be estimated using a SLAM algorithm based on LiDAR and IMU sensor data to reach a target pose.
- **Unexplored map with full navigability**: The drone is free to move in the virtual environment, and its pose must be estimated using a SLAM algorithm with the same sensor data as in the previous case to reach a target pose.

1.1 Hardware & Software Used

- GPU: AMD Radeon RX 6600
- CPU: AMD Ryzen 5 7600X (12) @ 4.700G
- OS: Ubuntu 20.04.6 LTS x86_64
- ROS: ROS Noetic
- CARLA 0.9.13
- Carla-ros-bridge: version compatible with CARLA 0.9.13
- Cartographer
- Move_base
- Explore_lite
- Aeplanner
- Octomap

To properly install CARLA, it is highly recommended to follow the steps outlined in the CARLA documentation, including updating drivers for non-Nvidia graphics cards.

2 CARLA-ROS-BRIDGE Explanation

The bridge enables bidirectional communication between CARLA and ROS, allowing vehicles in the simulation to be controlled through a structure of nodes and topics.
Once the CARLA server is started, the bridge can be launched using the dedicated launch file, which runs the bridge.py script while loading specific values defined in the launch file. The core component is the CarlaRosBridge class, which manages the connection with CARLA and the

exchange of data with ROS.
This class is responsible for initializing the bridge, publishing data to ROS, and receiving control commands.
When using synchronous mode, the bridge advances the simulation in a controlled manner through _synchronous_mode_update(), which updates the actors, executes the simulation tick(), and publishes the simulation time on /clock.
Additionally, objects are created to support the bridge's functionalities:

- ActorFactory: Manages the creation and updating of actors (vehicles, pedestrians, sensors).
- WorldInfo: Publishes information about the map and the simulated world.
- DebugHelper: Allows visualization of debugging data in CARLA.

Once the bridge is initialized, sensors can be created using CARLA's library, and the bridge ensures that they are correctly replicated within the ROS system.

```
...
    def main():

        ...

        drone = Drone(client, world)

        # Adding sensors to the drone
        drone.add_lidar_sensor()
        drone.add_imu_sensor()
        drone.add_rgb_camera()

        ...
```

The functions of the drone class for adding sensors follow this structure:

```
def add_lidar_sensor(self):
        """Adding a LiDAR sensor to the drone"""
        lidar_bp = self.world.get_blueprint_library()
                            .find('sensor.lidar.ray_cast')
        # Max range
        lidar_bp.set_attribute('range', '200.0')
        # Number of channel
        lidar_bp.set_attribute('channels', '64')
        # Points' density
        lidar_bp.set_attribute('points_per_second', '1000000')
        # Rotation frequency
        lidar_bp.set_attribute('rotation_frequency', '20')
        lidar_bp.set_attribute('upper_fov', '0')
        lidar_bp.set_attribute('lower_fov', '-90')

        # LiDAR's relative position w.r.t drone
        spawn_point = carla.Transform(carla.Location(x=0.0, y=0.0, ...
            z=0.0))

        self.lidar_sensor = self.world.spawn_actor(lidar_bp, ...
            spawn_point, attach_to=self.spectator)
        rospy.loginfo("LiDAR aggiunto con successo.")
```

Very simply, the CARLA blueprint library is used to define the sensor, its parameters are set accordingly, and it is then spawned in the CARLA world with the property attach_to=self.spectator .
The Drone class will be analyzed later in the discussion, but it is important to highlight that when sensors are created, the bridge automatically generates topics to publish the data collected from them and publishes the transformations between the drone and the sensors on /tf.

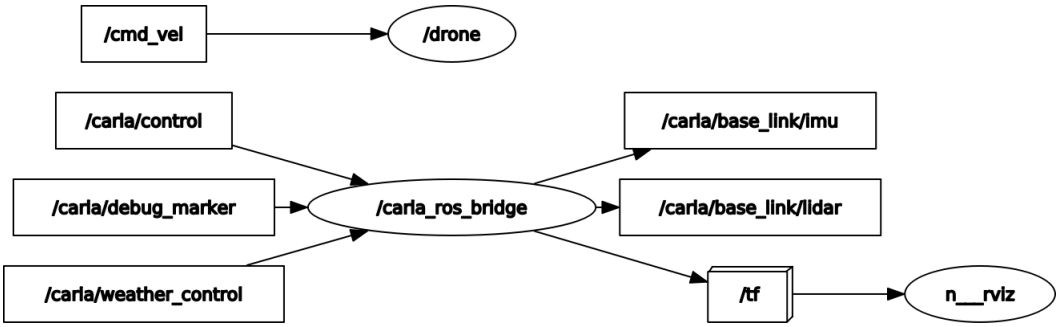

Figure 1: carla-ros-bridge's rqt_graph

Messages of type sensor_msgs/PointCloud2 will be published on the /carla/base_link/lidar topic, while messages of type sensor_msgs/Imu will be published on the /carla/base_link/imu topic.

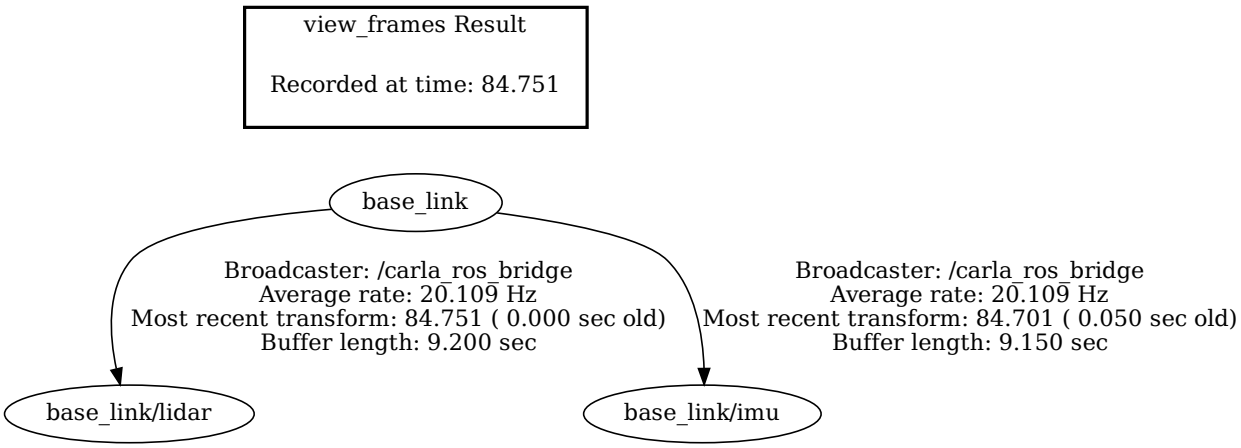

Figure 2: view_frames output

The topic name assigned automatically upon sensor creation is generated using the following formula:

$$/carla/[< PARENT\_ROLE\_NAME >]/ < SENSOR\_ROLE\_NAME >$$

The spectator, which in this case is the drone, does not have an assigned role name. Therefore, without modifying the corresponding file, the spectator's ID, "24," would have been used instead. A similar issue applies to the sensors: if their respective files were not modified, they would appear in the ROS system with the default names "front" for the LiDAR and "default" for the IMU.

```
class Lidar(Sensor):

    ...

    def __init__(self, uid, name, parent, relative_spawn_pose, node, ...
        carla_actor, synchronous_mode):

        super(Lidar, self).__init__(uid,
                                    name=name,    ----> name="lidar"
                                    parent=parent,
                                    node=node,
                                    carla_actor=carla_actor,
                                    synchronous_mode=synchronous_mode)
        ...
```

3 Pre-explored map for trajectory control

The first scenario considered assumes that the map has already been explored, meaning a predefined trajectory is available for the drone to follow. Additionally, the drone is equipped with a GPS, allowing its position to be known at all times.
The problem to be addressed is therefore a **pose control** task, where the drone's current position, obtained from the GPS, is compared to the goal position on the trajectory. From this, a **position error** and an **orientation error** are derived, which will be minimized using a controller—in this case, a PID controller.

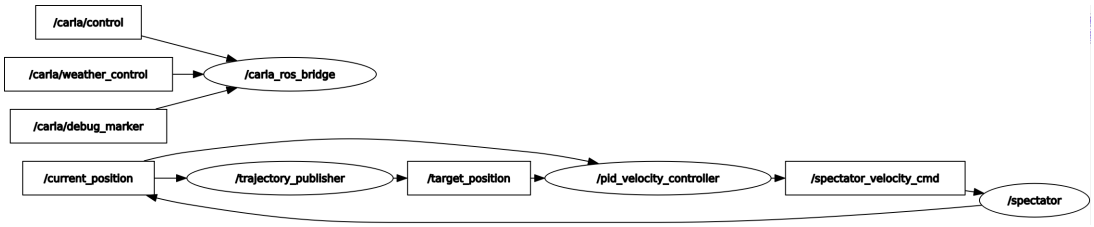

Figure 3: System's architecture for trajectory control

As shown in the diagram, for this first scenario, there is no need for LiDAR or IMU sensors mounted on the drone—only GPS is required. For simplicity, instead of replicating the functionality of a GPS as a physical sensor, the drone's position is obtained directly using CARLA's APIs.
The following section provides a detailed explanation of how the control system works.

3.1 Node /trajectory_publisher

As shown in **Figure 3**, the node that publishes the trajectory point is subscribed to the /current_position topic. This is because the node must publish the correct target point on /target_position only when the drone has reached the previous target point.
To demonstrate the system's functionality, a simple square trajectory is used. However, any sequence of points can be defined either within the script or in a JSON file to be loaded.
This requires appropriately adjusting the threshold, which defines the vicinity around the target point within which the objective is considered reached.

```
class TrajectoryPublisher:
    def __init__(self):
        rospy.init_node('trajectory_publisher')
        rospy.loginfo("Nodo 'trajectory_publisher' avviato.")
        self.pub = rospy.Publisher('/target_position', Point, ...
            queue_size=10)
        rospy.Subscriber('/current_position', Twist, ...
            self.position_callback)

        self.current_position = None
        self.threshold = 5.0
        self.rate = rospy.Rate(5)

        # Square trajectory
        self.trajectory = [
            Point(0, 200, 50),
            Point(200, 200, 5),
            Point(200, 0, 50),
            Point(0, 0, 5)
        ]
        self.index = 0


    def position_callback(self, msg: Twist):
        self.current_position = Point(msg.linear.x, msg.linear.y, ...
            msg.linear.z)
```

In the init() function, the node subscribes to the /current_position topic and defines the topic /target_position for publishing the current destination point. Additionally, the trajectory is defined (in this case, a square with vertices at different altitudes), along with a threshold, which is set to 5 meters.

```
        def run(self):
        while not rospy.is_shutdown():
            if self.current_position:
                target = self.trajectory[self.index]
                distance = ...
                    self.calculate_distance(self.current_position, ...
                    target)


                if distance ≤ self.threshold:
                    self.index = (self.index + 1) % len(self.trajectory)

            else:
                rospy.loginfo("Attendere l'aggiornamento della ...
                    posizione...")

            self.pub.publish(self.trajectory[self.index])
            self.rate.sleep()
```

The run() function, on the other hand, is responsible for publishing the next target point only when the distance between the current position and the current target point is smaller than the previously defined threshold.

3.2 Node /pid_velocity_controller

The main objective of the script is to control the drone's position by adjusting its linear and angular velocity to minimize the error between the current and desired positions. This is achieved using **six PID controllers**: three for the linear velocity components **(x, y, z)** and three for the orientation components **(roll, pitch, yaw)**.
The ROS node, named pid_velocity_controller, receives:

- The **target position** (/target_position) as a message of type Point.
- The **current position** (/current_position) as a message of type Twist.

After processing the position and orientation errors, it publishes the **velocity command** to /spectator_velocity_cmd.
The position error is simply calculated as the difference between the current position and the target position, using information directly obtained from the subscribed messages.
Regarding orientation errors, before defining the error itself, the desired orientation is determined based on the position errors. Essentially, the drone's orientation reference consists of the angles at which the drone, from its current position, would directly face the target point it is moving toward.

```
        # Orientation ref
            desired_yaw = math.atan2(error.y, error.x)
            desired_pitch = -math.atan2(error.z, math.sqrt(error.x**2 + ...
                error.y**2))
            desired_roll = 0.0  # Roll target specifico

        # Current orientation
            current_yaw = self.current_position.angular.z * math.pi / 180
            current_pitch = self.current_position.angular.y * math.pi / 180
            current_roll = self.current_position.angular.x * math.pi / 180

        # Error calculation
            yaw_error = desired_yaw - current_yaw
            pitch_error = desired_pitch - current_pitch
            roll_error = desired_roll - current_roll

        # Normalization to avoid discontinuity
            yaw_error = math.atan2(math.sin(yaw_error), math.cos(yaw_error))
            pitch_error = math.atan2(math.sin(pitch_error), ...
                math.cos(pitch_error))
            roll_error = math.atan2(math.sin(roll_error), ...
                math.cos(roll_error))

        # Angular velocity
            angular_yaw = self.pid_yaw.compute(yaw_error)
            angular_pitch = self.pid_pitch.compute(pitch_error)
            angular_roll = self.pid_roll.compute(roll_error)
```

The compute() function of the PIDController class calculates the control action, i.e., the velocity, based on the input error and applies a saturation limit to this signal, stated during the declaration of the PID object.

```
        def compute(self, error):
            current_time = time.time()
            Δ_time = current_time - self.last_time
```

```
            if Δ_time == 0:  # Evita divisioni per zero
                Δ_time = 1e-6

            # Componente proporzionale
            p = error * self.kp

            # Componente integrale
            self.integral += error * Δ_time
            i = self.integral * self.ki

            # Componente derivativa
            derivative = (error - self.prev_error) / Δ_time
            d = derivative * self.kd

            # Aggiorna gli stati precedenti
            self.prev_error = error
            self.last_time = current_time

            # Somma delle componenti
            output = p + i + d

            # Applica la saturazione
            output = max(-self.max_output, min(self.max_output, output))

            return output
```

The PID controller was tuned using an empirical approach based on the trial-and-error method. Initially, the integral and derivative gains (Ki = 0, Kd = 0) were set to zero in order to evaluate the effect of the proportional term (Kp) alone, which was gradually increased until a sufficiently responsive yet stable system behavior was achieved. The derivative gain (Kd) was then introduced to dampen any oscillations, followed by the integral gain (Ki) to eliminate steady-state error. The parameters were iteratively adjusted to achieve a suitable compromise between response time, stability, and accuracy.

3.3 Node /spectator

This node communicates with CARLA for the **drone's movement**. Since CARLA does not natively implement a "drone" vehicle, the spectator (i.e., the camera that the user can use to move freely in the CARLA world during simulations) has been used instead.
CARLA's APIs do not allow directly passing velocity commands to the spectator object. For this reason, in this node, a **delta_time** is defined, and the position changes are calculated. These variations are then added to the current positions at each delta_time, allowing the spectator to effectively move at the correct velocity.

```
class Drone:
    def __init__(self, client, world):
        self.world = world
        self.spectator = world.get_spectator()
        self.velocity = carla.Vector3D(0, 0, 0)
        self.angular_velocity = carla.Vector3D(0, 0, 0)
        rospy.Subscriber('/spectator_velocity_cmd', Twist, ...
            self.velocity_callback)
        self.position_pub = rospy.Publisher('/current_position', ...
            Twist, queue_size=10)
```

```
    def velocity_callback(self, msg: Twist):
        self.velocity.x = msg.linear.x
        self.velocity.y = msg.linear.y
        self.velocity.z = msg.linear.z
        self.angular_velocity.x = msg.angular.x
        self.angular_velocity.y = msg.angular.y
        self.angular_velocity.z = msg.angular.z

    def update_position(self, Δ_time):
        current_transform = self.spectator.get_transform()
        location = current_transform.location
        rotation = current_transform.rotation

        # Position updating
        location.x += self.velocity.x * Δ_time
        location.y += self.velocity.y * Δ_time
        location.z += self.velocity.z * Δ_time

        # Orientation updating
        rotation.pitch += self.angular_velocity.x * Δ_time
        rotation.yaw += self.angular_velocity.z * Δ_time
        rotation.roll += self.angular_velocity.y * Δ_time

        self.spectator.set_transform(carla.Transform(location, ...
            rotation))
```

Finally, mimicking the functionality of a GPS, the drone's current pose is published on the /current_position topic. This allows the control loop to be closed and also provides the necessary information to the node responsible for the trajectory.

Figure 4: CARLA view

```
Errore di posizione: x=4.73, y=-2.56, z=-0.41
Errore di orientamento: yaw=-0.15, pitch=0.02, roll=0.00
Errore di posizione: x=3.73, y=-2.55, z=-0.41
Errore di orientamento: yaw=-0.21, pitch=0.03, roll=0.00
Errore di posizione: x=2.73, y=-202.55, z=44.59
Errore di orientamento: yaw=-1.13, pitch=-0.29, roll=0.00
Errore di posizione: x=1.73, y=-201.55, z=43.59
Errore di orientamento: yaw=-1.09, pitch=-0.24, roll=0.00
Errore di posizione: x=0.73, y=-200.55, z=42.59
Errore di orientamento: yaw=-1.05, pitch=-0.19, roll=0.00
Errore di posizione: x=-0.27, y=-199.55, z=41.59
Errore di orientamento: yaw=-1.01, pitch=-0.14, roll=0.00
Errore di posizione: x=-1.27, y=-198.55, z=40.59
Errore di orientamento: yaw=-0.97, pitch=-0.10, roll=0.00
```

Figure 5: LiDAR view

4 Unexplored map with altitude constraints

The challenge presented in this new scenario is the complete lack of information about the environment in which the drone operates and, consequently, the absence of a predefined trajectory to follow. Additionally, unlike the previous scenario, the drone is not equipped with a GPS system. Therefore, the problem must be broken down into smaller subproblems to be addressed individually.
The first issue is the lack of environmental data, which necessitates mapping operations using data from the sensors mounted on the drone—specifically, the LiDAR and IMU. Naturally, for these data to be processed correctly, the drone's position must be known. Since GPS is unavailable, a **SLAM** algorithm must be employed to estimate the drone's pose based on the available sensor data.
These mapping and localization operations have been carried out using **Cartographer**, a SLAM system that utilizes LiDAR and IMU data to estimate the drone's pose and progressively build a map of the environment.
Thanks to Cartographer, the drone can generate a real-time representation of the scene, allowing it to localize itself within the environment without the need for a GPS.
Once a reliable pose estimate and a continuously updated map were obtained, an autonomous exploration system for the drone was implemented.
In this scenario, exploration was managed using the **explore_lite** package, which enables the drone to autonomously navigate toward unexplored areas. For navigation, **move_base** was employed to plan and generate safe trajectories based on the map created by Cartographer.
This configuration allowed the drone to progressively explore the environment, updating the map in real time and adjusting its path based on obstacles detected by the LiDAR.
The difference between this scenario and the next lies in the drone's ability to move freely in space. In this scenario, constraints have been imposed, restricting the drone's movement to the **XY plane**.
As a result, the only velocity commands the drone will receive are **linear velocities along X and Y** and **angular velocity around the Z-axis (yaw)**.

4.1 Cartographer Node for Mapping and Localization

Rather than presenting the entire control loop at once, it is more effective to examine each component individually and explain how each part functions on its own before discussing their coordination.
We begin with the Cartographer node, which is responsible for **mapping** the environment and **localizing** the drone.

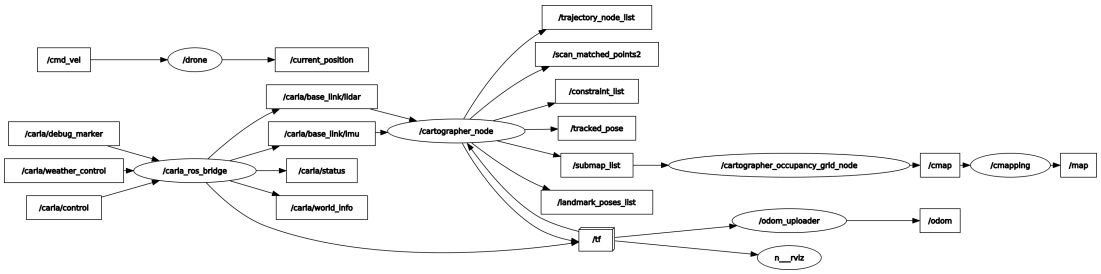

Figure 6: Cartographer's rqt_graph

The startup operations are the same as in the previous scenario: the CARLA server is launched, the CARLA-ROS bridge is initialized, and the drone is created along with its sensors.

The sensor data is published on the topics /carla/base_link/imu and /carla/base_link/lidar, which are then used as inputs for the new Cartographer node to construct a map of the environment and estimate the drone's pose. The configuration file of Cartographer specifies system parameters, including the sensor model, update frequency, and filters applied to raw data.

```
include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
  map_frame = "map",
  tracking_frame = "base_link",  -- Imu frame
  published_frame = "base_link",  -- Drone frame
  odom_frame = "odom",
  provide_odom_frame = true,
  publish_frame_projected_to_2d = false,
  publish_tracked_pose = true,
  use_odometry = false,
  use_nav_sat = false,
  use_landmarks = false,
  num_laser_scans = 0,
  num_multi_echo_laser_scans = 0,
  num_subdivisions_per_laser_scan = 4,
  num_point_clouds = 1,
  lookup_transform_timeout_sec = 0.2,
  submap_publish_period_sec = 0.3,
  pose_publish_period_sec = 5e-3,
  trajectory_publish_period_sec = 30e-3,
  rangefinder_sampling_ratio = 1.,
  odometry_sampling_ratio = 1.,
  fixed_frame_pose_sampling_ratio = 1.,
  imu_sampling_ratio = 1.,
  landmarks_sampling_ratio = 1.,
}

MAP_BUILDER.use_trajectory_builder_2d = true

TRAJECTORY_BUILDER_2D.min_range = 4.0  -- Distanza minima del LiDAR
TRAJECTORY_BUILDER_2D.max_range = 50.0  -- Distanza massima del LiDAR
TRAJECTORY_BUILDER_2D.missing_data_ray_length = 2.
TRAJECTORY_BUILDER_2D.use_imu_data = true
TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true
TRAJECTORY_BUILDER_2D.motion_filter.max_angle_radians = math.rad(0.1)
TRAJECTORY_BUILDER_2D.motion_filter.max_distance_meters = 0.1
TRAJECTORY_BUILDER_2D.motion_filter.max_time_seconds = 0.5

-- Configurazione del LiDAR
TRAJECTORY_BUILDER_2D.num_accumulated_range_data = 1
TRAJECTORY_BUILDER_2D.voxel_filter_size = 0.025

-- Configurazione del IMU
TRAJECTORY_BUILDER_2D.imu_gravity_time_constant = 10.
```

```
-- Configurazione del loop closure
POSE_GRAPH.constraint_builder.min_score = 0.65
POSE_GRAPH.constraint_builder.global_localization_min_score = 0.7
POSE_GRAPH.constraint_builder.log_matches = true
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher_3d
                                    .min_rotational_score = 0.9
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher_3d
                                    .min_low_resolution_score = 0.5

-- Ottimizzazione della mappa
POSE_GRAPH.optimize_every_n_nodes = 90
POSE_GRAPH.max_num_final_iterations = 200
POSE_GRAPH.global_sampling_ratio = 0.003
POSE_GRAPH.log_residual_histograms = true
POSE_GRAPH.global_constraint_search_after_n_seconds = 10.

return options
```

The Cartographer node processes the IMU measurements to estimate the drone's motion between consecutive LiDAR scans, while the LiDAR data is used to update the map and correct any errors in the pose estimation through a SLAM-based optimization.
The drone's pose estimate is published in the ROS transformation system (/tf), allowing other modules to know the drone's current position relative to the map, that is by default the fixed frame located in (0,0,0).

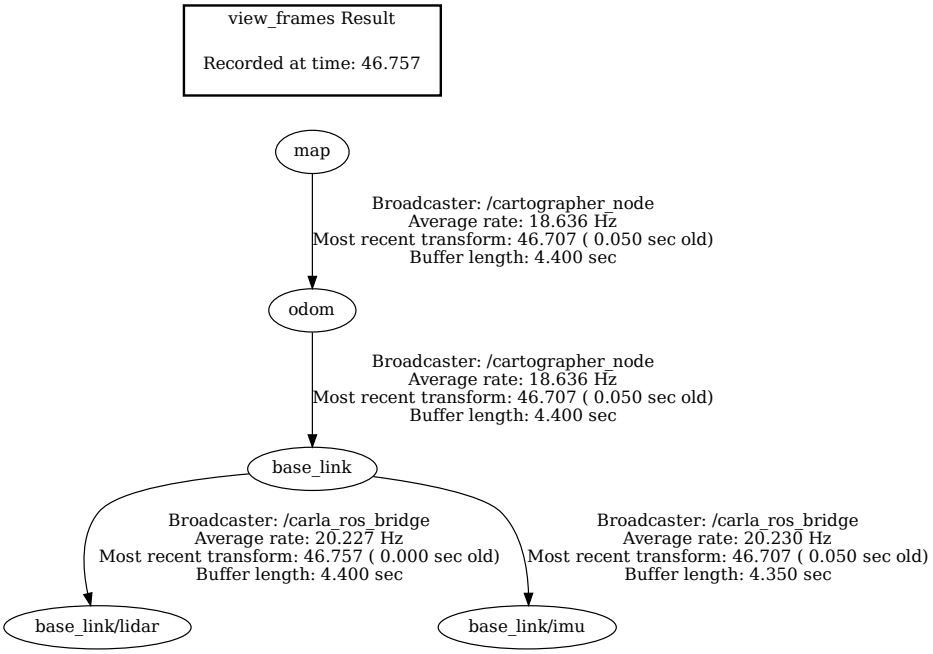

Figure 7: Cartographer's view_frames output

The map is built using a 2D grid representation, where each cell contains a probability value indicating whether the space is occupied, free, or unknown, and then trasmitted on /map as a

nav_msgs/OccupancyGrid message, which can be used by other navigation and path-planning algorithms.

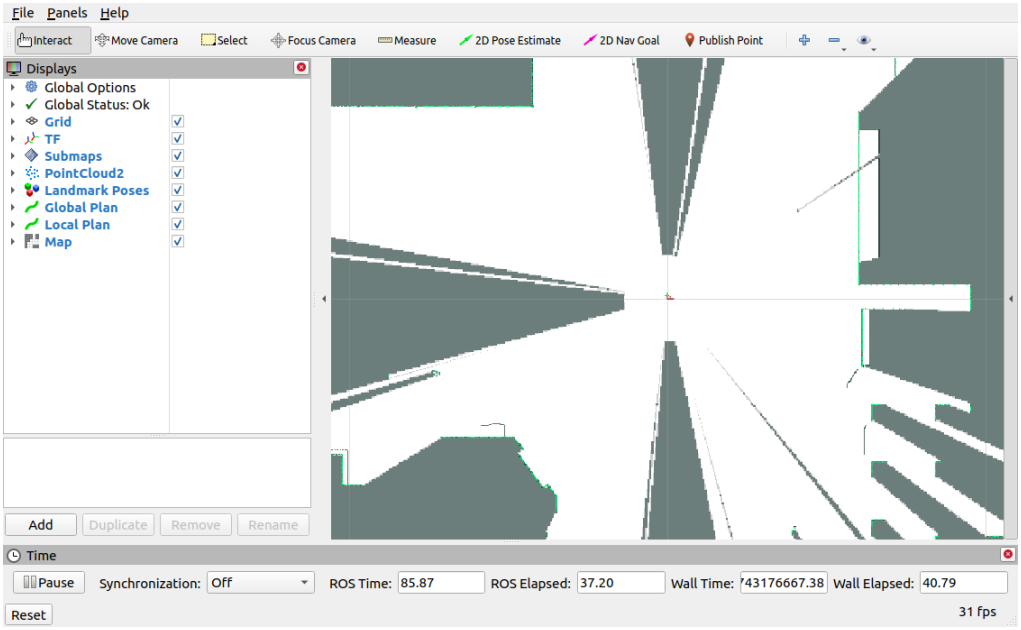

Figure 8: Cartographer's rviz screenshot

As seen in **Figure 6**, there are some nodes, not belonging to Cartographer, that allow evaluating the pose estimate produced by Cartographer and transforming it into input and output in a way that ensures compatibility with the rest of the control loop.

4.1.1 Node position_error_node

This node is not currently visible in the rqt graph, but it was used during the evaluation phase of the pose estimation error from Cartographer. The error is calculated by comparing the estimate produced by Cartographer on /tf with the exact position value obtained through the CARLA APIs, which is published on the /current_position topic.

```
class PositionErrorNode:
    def __init__(self):

        self.init_pos = carla.Vector3D(190,240,1.5)

        rospy.init_node('position_error_node', anonymous=True)

        # Sottoscrizione ai topic
        rospy.Subscriber('/current_position', Twist, ...
            self.real_position_callback)
        rospy.Subscriber('/tf', TFMessage, ...
            self.estimated_position_callback)

        self.real_position = None
        self.estimated_position = None

    def real_position_callback(self, msg):
```

```
            self.real_position = (msg.linear.x - self.init_pos.x, ...
                msg.linear.y - self.init_pos.y, msg.linear.z - ...
                self.init_pos.z)
            self.compute_error()

        def estimated_position_callback(self, msg):
            for transform in msg.transforms:
                if transform.child_frame_id == "base_link":
                    self.estimated_position = (
                        transform.transform.translation.x,
                        -transform.transform.translation.y,
                        transform.transform.translation.z
                    )
                    self.compute_error()

        . . .
```

The node finally publishes the difference between the positions obtained from the two callbacks on the terminal. An interesting detail that can be highlighted concerns the different reference frames used by **CARLA** and **ROS**. While ROS employs a right-handed coordinate system, CARLA uses a left-handed coordinate system. This difference can already be observed in the second callback, where we assign the value -transform.transform.translation.y to self.estimated_position.y.

4.1.2 Node cmapping

As previously mentioned, Cartographer publishes a map using a 2D grid format, assigning a value to each cell that represents the probability of the cell being free, occupied, or unexplored. However, this map format is not ideal for **move_base**, which we will use for navigation. move_base requires a more "deterministic" map format—one where the cells only take on three values to represent the states "free," "occupied," and "unexplored".

```
    # ***************
    #   OBSTACLE
    M = 75
    #   unknown
    N = 50
    #   free
    # ----0-----
    #   unknown
    # ***************
    def callback(cmap: OccupancyGrid):
        data = list(cmap.data)
        for y in range(cmap.info.height):
            for x in range(cmap.info.width):
                i = x + (cmap.info.height - 1 - y) * cmap.info.width
                if data[i] ≥ M:  # obstacle
                    data[i] = 100
                elif (data[i] ≥ 0) and (data[i] < N):  # free
                    data[i] = 0
                else:  # unknown
                    data[i] = -1
        cmap.data = tuple(data)
        pub.publish(cmap)
```

```
rospy.init_node('mapc_node', anonymous=True)
sub = rospy.Subscriber('/cmap', OccupancyGrid, callback)
pub = rospy.Publisher('/map', OccupancyGrid, queue_size=20)

rospy.spin()
```

When Cartographer publishes a map on the topic /cmap (originally /map, but remapped), the callback reads the values assigned to each individual cell and modifies them as follows:

- **Obstacles (M=75)**: Set to 100.
- **Free (0 <= value < N=50)**: Set to 0.
- **Unknown (value < 0 or >= N)**: Set to -1.

After these modifications, the new map, now in the GMapping format, is published on the /map topic.

4.1.3 Node odom_uploader

This node addresses another prerequisite for **move_base**, which we will use for navigation. move_base requires the drone's **odometry**, not its transformation in /tf. The issue is that it's not possible for Cartographer to publish **nav_msgs/Odometry** messages directly to provide to move_base. Therefore, as suggested online, this workaround was employed to derive nav_msgs/Odometry messages from the transformations published on /tf.

```
def tf_to_odometry():
    rospy.init_node('tf_to_odometry')

    # Inizializzazione del listener TF2
    tfBuffer = tf2_ros.Buffer()
    listener = tf2_ros.TransformListener(tfBuffer)

    odom_pub = rospy.Publisher('/odom', Odometry, queue_size=10)

    rate = rospy.Rate(10.0)
    while not rospy.is_shutdown():
        try:
            # Ottieni la trasformazione da 'odom' a 'base_link'
            transform = tfBuffer.lookup_transform('odom', ...
                'base_link', rospy.Time(0), rospy.Duration(1.0))

            # Crea un messaggio Odometry
            odom = Odometry()

            # Estrai la posizione e l'orientamento dalla trasformazione
            odom.header.stamp = rospy.Time.now()
            odom.header.frame_id = 'odom'
            odom.child_frame_id = 'base_link'

            # Posizione
            odom.pose.pose.position.x = ...
                transform.transform.translation.x
            odom.pose.pose.position.y = ...
                transform.transform.translation.y
```

```
                odom.pose.pose.position.z = ...
                    transform.transform.translation.z

                # Orientamento
                odom.pose.pose.orientation = transform.transform.rotation

                odom.twist.twist.linear.x = 0.0
                odom.twist.twist.linear.y = 0.0
                odom.twist.twist.linear.z = 0.0

                # Pubblica il messaggio di odometria
                odom_pub.publish(odom)

            except (tf2_ros.LookupException, ...
                tf2_ros.ConnectivityException, ...
                tf2_ros.ExtrapolationException):
                rospy.logwarn("Trasformazione non disponibile!")

            rate.sleep()

    if __name__ == '__main__':
        try:
            tf_to_odometry()
        except rospy.ROSInterruptException:
            pass
```

Using the **TF2 listener**, the node obtains the transformation between the **odom** and **base_link** frames, which represent the coordinate systems of the map and the robot, respectively. The node extracts the robot's position and orientation from this transformation and uses them to construct an **Odometry message**, which is then published on the /odom topic. The odometry message includes the robot's position in 3D space and its orientation, while the velocity is set to zero since it's not important for move_base.

4.2 move_base for navigation

move_base is one of the main components of the **ROS framework**, used for the **autonomous navigation** of robots and drones. It provides a complete solution for **planning** and **executing** movements, combining a series of algorithms that allow the robot to move safely and efficiently in complex and unknown environments.
The move_base node is responsible for generating safe trajectories for the robot and executing them in real-time, considering both the environment map and the robot's current position. Its operation mainly occurs in two phases:

1. **Global Planning**: Global planning involves determining a path to follow, starting from a starting position to a destination, using a pre-existing map. This path is calculated using planning algorithms like A* or Dijkstra, which find the best route considering known obstacles on the map.

2. **Local Planning**: Once the global path is calculated, local planning comes into play to allow the robot to move reactively, taking into account unexpected obstacles and dynamic changes in the environment. This happens through algorithms like the Dynamic Window Approach (DWA), which calculate safe trajectories in real-time and allow rapid course corrections.

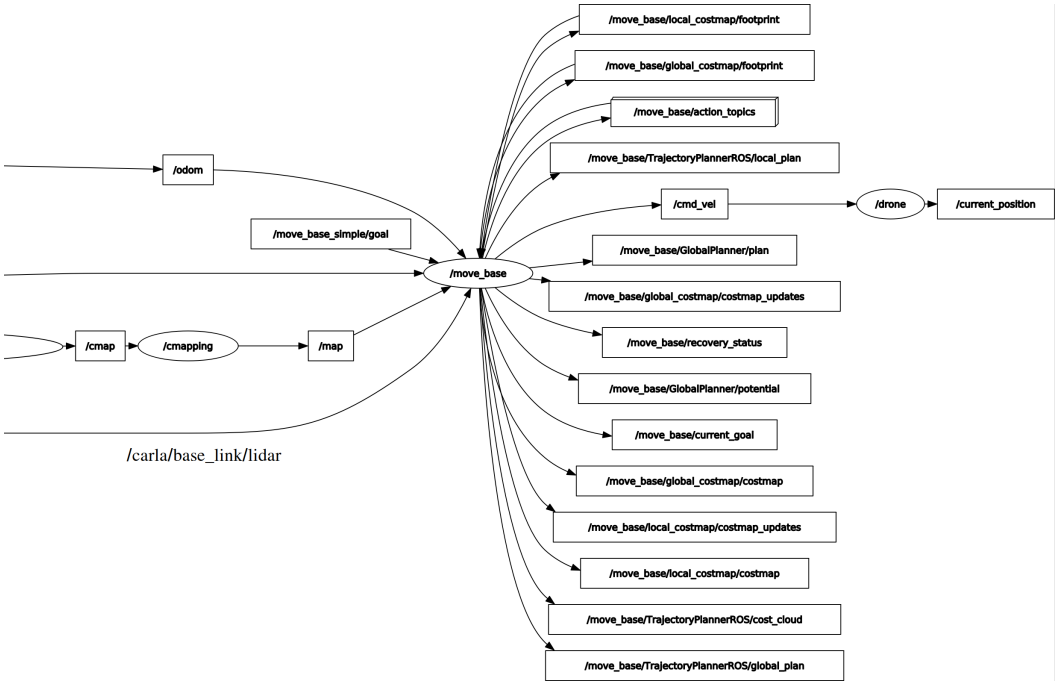

Figure 9: move_base's rqt_graph

The only input we haven't discussed yet is the topic /**move_base_simple/goal**. This topic allows you to pass the destination for the drone to the **move_base** node through the global and local planners, whose parameters have been defined in the configuration files.
Once the move_base node is started, it is sufficient to simply publish a **geometry_msgs/PoseStamped** message on the /move_base_simple/goal topic, so that the node can start processing velocity commands to send to the drone on the **/cmd_vel topic**.

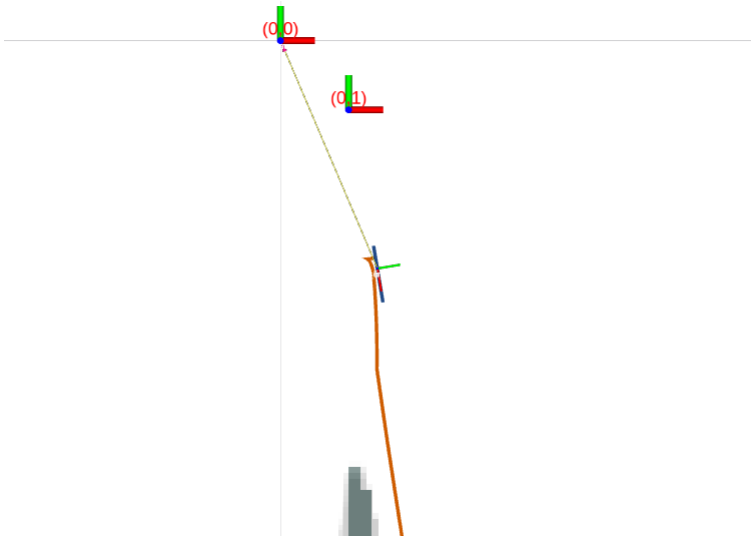

Figure 10: move_base's rviz visual

4.2.1 costmap_common_params.yaml

This file defines the common parameters used by both the global and local costmaps. Costmaps are 2D representations of the surrounding environment in which obstacles, free space, and unknown areas are marked.

Key parameters:

- **obstacle_range**: The maximum distance (in meters) at which obstacles detected by the sensor are considered.
- **raytrace_range**: The maximum distance for ray tracing, i.e., the range within which obstacles are cleared along the visible ray from the sensor.
- **footprint**: The projection of the robot's shape on the XY plane; used to avoid collisions during path planning.
- **inflation_radius**: The distance around each obstacle that is "inflated" in the costmap to maintain a safety buffer.
- **observation_sources**: Logical names of the sensors being used.
- **point_cloud_sensor**: Configuration of the 3D sensor (LiDAR) that provides data on obstacles and free space.

```
obstacle_range: 10.0
raytrace_range: 6.0
footprint: [[-0.3, -0.3], [-0.3, 0.3], [0.3, 0.3], [0.3, -0.3]]
inflation_radius: 0.55
observation_sources: point_cloud_sensor

point_cloud_sensor:
  sensor_frame: base_link
  data_type: PointCloud2
  topic: carla/base_link/lidar
  marking: true
  clearing: true
```

4.2.2 local_costmap_params.yaml

This file configures the local costmap, which is used by the local planner to make real-time motion decisions while accounting for dynamic obstacles.

Key Parameters

- **update_frequency**: The frequency (in Hz) at which the costmap is updated with new sensor data.
- **publish_frequency**: The frequency (in Hz) at which the costmap is published over ROS topics.
- **static_map**: If set to `true`, a static map is used (typically provided by a SLAM or mapping algorithm).
- **track_unknown_space**: If `true`, unknown areas (i.e., cells with a value of `-1`) are included in the costmap. This is useful for planning in partially explored environments.

- **plugins**: A list of plugins (layers) used to build the costmap. Each layer contributes specific information (e.g., obstacles, inflation).

Costmap Layers
voxel_layer
Handles three-dimensional obstacle data using a voxel grid. This layer is suitable for processing data from 3D sensors such as LiDAR.

- **observation_sources**: A list of sensor sources used for obstacle detection.
- **data_type**: Specifies the type of data received from the sensor (e.g., `PointCloud2`).
- **marking / clearing**: Determines whether incoming sensor data is used to mark obstacles (`marking`) or clear free space (`clearing`).
- **obstacle_range / raytrace_range**: The maximum range within which obstacles are detected / space is cleared.
- **min_obstacle_height / max_obstacle_height**: Minimum and maximum height (in meters) of obstacles to be considered valid.
- **origin_z, z_resolution, z_voxels**: Parameters defining the voxel grid in the vertical (z) direction.
- **publish_voxel_map**: If `true`, the voxel map is published to a ROS topic.
- **footprint_clearing_enabled**: If `true`, allows clearing the robot's footprint area to prevent the robot from being considered an obstacle to itself.

inflation_layer
Expands the area around detected obstacles to account for the robot's physical dimensions and to maintain a safety buffer.

- **inflation_radius**: The radius (in meters) around obstacles in which the cost values are increased.
- **cost_scaling_factor**: Determines how quickly the cost decreases with distance from the obstacle. A higher value results in a steeper cost gradient.

obstacle_layer
Handles obstacle data detected by sensors and projects this information onto the 2D costmap. It is particularly useful when working with 2D sensors or when focusing on a specific horizontal section of data from 3D sensors.

- **observation_sources**: A list of sensors used for obstacle detection. Each sensor specified here must be defined separately (e.g., lidar_sensor).
- **marking**: If set to true, incoming sensor data is used to mark cells as occupied (obstacles).
- **clearing**: If set to true, incoming sensor data is also used to clear cells, removing obstacles where no longer detected.
- **obstacle_range**: The maximum range (in meters) within which obstacles are considered valid. Points beyond this range are ignored.
- **raytrace_range**: The maximum range (in meters) used for raytracing clearing, which removes obstacles along the visible path of the sensor.

- **expected_update_rate**: The expected rate (in seconds) at which the sensor updates its data. This can be used to detect inactive or malfunctioning sensors.

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 10.0
  publish_frequency: 5.0
  static_map: false
  rolling_window: true
  width: 8.0
  height: 8.0
  resolution: 0.05

  plugins:
    - {name: voxel_layer, type: "costmap_2d::VoxelLayer"}
    - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

voxel_layer:
  observation_sources: lidar_sensor
  lidar_sensor: {
    data_type: PointCloud2,
    topic: carla/base_link/lidar,
    marking: true,
    clearing: true,
    obstacle_range: 5.0,
    raytrace_range: 6.0,
    expected_update_rate: 0.2,
    min_obstacle_height: 0.1,
    max_obstacle_height: 2.0
  }

  origin_z: 0.0
  z_resolution: 0.1
  z_voxels: 16
  publish_voxel_map: false
  footprint_clearing_enabled: true

inflation_layer:
  inflation_radius: 0.6
  cost_scaling_factor: 5.0

obstacle_layer:
  observation_sources: lidar_sensor
  lidar_sensor: {
    data_type: PointCloud2,
    topic: carla/base_link/lidar,
    marking: true,
    clearing: true,
    obstacle_range: 10.0,
    raytrace_range: 6.0,
    expected_update_rate: 0.2,
    min_obstacle_height: 0.1,
    max_obstacle_height: 2.0
  }
```

4.2.3 global_costmap_params.yaml

This file configures the global costmap, which is used by the global planner to compute the optimal path from the robot's current position to the target destination.
The parameters for this file are the same of the previous one, so they won't be described again here.

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  publish_frequency: 1.0
  static_map: true
  track_unknown_space: true

  plugins:
    - {name: voxel_layer, type: "costmap_2d::VoxelLayer"}
    - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

voxel_layer:
  observation_sources: lidar_sensor
  lidar_sensor: {
    data_type: PointCloud2,
    topic: /lidar_points,
    marking: true,
    clearing: true,
    obstacle_range: 5.0,
    raytrace_range: 6.0,
    expected_update_rate: 0.2,
    min_obstacle_height: 0.1,
    max_obstacle_height: 2.0
  }

  origin_z: 0.0
  z_resolution: 0.1
  z_voxels: 16
  publish_voxel_map: false
  footprint_clearing_enabled: true

inflation_layer:
  inflation_radius: 0.6
  cost_scaling_factor: 5.0
```

4.2.4 local_planner_params.yaml

This file configures the behavior of the local planner, in this case TrajectoryPlannerROS, which is responsible for generating velocity commands (cmd_vel) to follow the global path while avoiding local obstacles.

Key parameters:

- **max_vel_x, min_vel_x**: Maximum and minimum translational velocities along the X axis.
- **max_vel_theta, min_vel_theta**: Maximum and minimum angular velocities.

- **acc_lim_x, acc_lim_y, acc_lim_theta**: Acceleration limits along the X, Y axes and in rotation.
- **xy_goal_tolerance, yaw_goal_tolerance**: Tolerances for considering the goal position and orientation as reached.
- **sim_time**: The time horizon over which trajectories are simulated.
- **vx_samples, vtheta_samples**: Number of velocity samples used to generate and evaluate possible trajectories.
- **path_distance_bias, goal_distance_bias**: Weights used to balance the importance of staying close to the global path versus heading directly toward the goal.
- **occdist_scale**: Weight applied to obstacle proximity in trajectory scoring.

```
TrajectoryPlannerROS:
  max_vel_x: 0.3
  min_vel_x: -0.3

  max_vel_theta: 0.4
  min_vel_theta: -0.4

  acc_lim_x: 0.5
  acc_lim_y: 0.5
  acc_lim_theta: 0.5

  xy_goal_tolerance: 0.2
  yaw_goal_tolerance: 0.2

  sim_time: 2.0 #0.8
  sim_granularity: 0.025
  angular_sim_granularity: 0.025

  vx_samples: 8
  vy_samples: 0
  vtheta_samples: 20

  path_distance_bias: 5.0
  goal_distance_bias: 15.0
  occdist_scale: 0.01

  heading_lookahead: 0.5
  oscillation_reset_dist: 0.1
  escape_reset_dist: 0.2
  escape_reset_theta: 0.2
```

4.2.5 move_base_params.yaml

This file contains the general configuration for the move_base node, specifying which planners to use and how frequently they should update their plans.

Key parameters:

- **base_global_planner**: Plugin used for global planning (e.g., global_planner/GlobalPlanner).

- **base_local_planner**: Plugin used for local planning (e.g., base_local_planner/TrajectoryPlannerROS).
- **planner_frequency**: Frequency at which the global planner updates its plan.
- **controller_frequency**: Frequency at which the local planner generates control commands.
- **recovery_behavior_enabled**: Enables automatic recovery behaviors if the robot becomes stuck.
- **clearing_rotation_allowed**: Allows the robot to rotate in place to search for escape paths when blocked.

```
base_global_planner: global_planner/GlobalPlanner
base_local_planner: base_local_planner/TrajectoryPlannerROS

controller_frequency: 5.0
planner_frequency: 1.0
recovery_behavior_enabled: true
clearing_rotation_allowed: true
```

4.3 Autonomous Exploration with explore_lite

The next step is the automatic generation of target points that the drone must reach in order to explore the area. For this purpose, **explore_lite** is used.
explore_lite is a ROS package designed for autonomous exploration of environments. It is a simplified version of the explore package, allowing a robot to autonomously navigate an unknown area, exploring free spaces and avoiding obstacles. It uses an exploration algorithm based on a real-time map, aiming to efficiently cover the entire environment.
To achieve this, explore_lite continuously analyzes the occupancy grid provided by the mapping system, identifying frontier points—boundaries between known and unknown areas. The package selects the most suitable frontier as the next exploration goal, sending navigation commands to the drone accordingly. By iteratively moving towards these frontiers, the drone incrementally discovers and maps the environment.

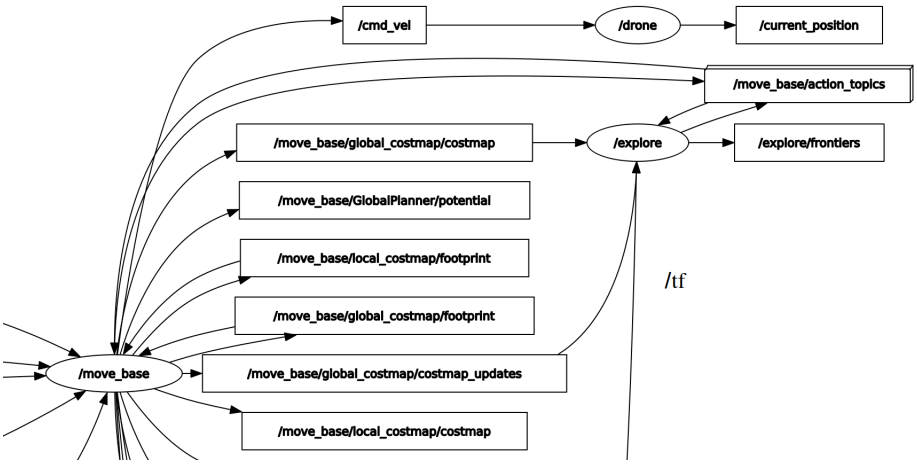

Figure 11: explore_lite's rqt_graph

Letting the exploration algorithm run, the drone will begin to explore the map, and the results will be visible in **RViz**. From these results, a problem related to CARLA becomes apparent. Since CARLA is primarily designed as a **traffic simulation** environment rather than for drone flight, many buildings appear **"empty"** when analyzed by the LiDAR. In other words, even though walls are visible in the camera feed, they often do not exist as **detectable points** in the LiDAR data, meaning they are not registered as obstacles. As a consequence, the exploration algorithm may attempt to guide the drone **through these undetected structures**.
To mitigate the impact of this issue, the drone was flown very close to street level, allowing all elements that would typically be considered obstacles for an autonomous vehicle to be treated as obstacles.

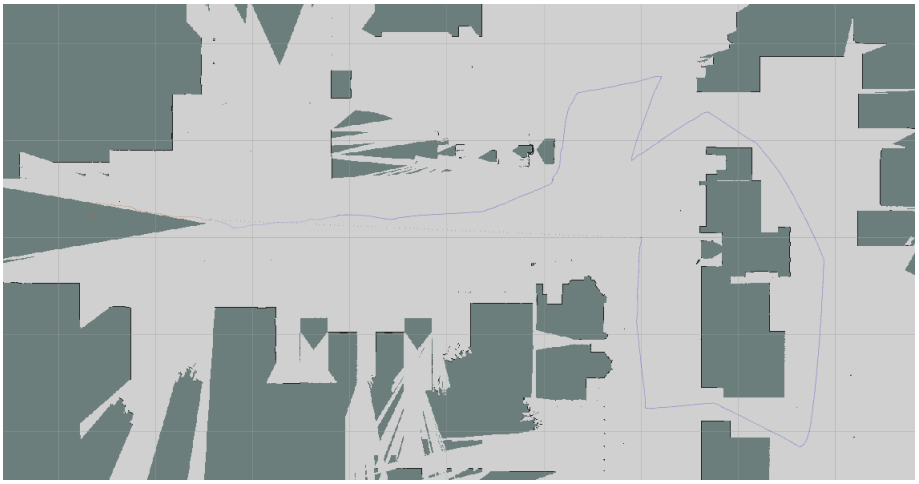

Figure 12: explore_lite's generated map and trajectory, approx. 13 minutes

To modify the behavior of the algorithm, it is necessary to edit the launch file located in the directory opt/ros/noetic/share/explore_lite. In this application, the following file was used:

```
<launch>
<node pkg="explore_lite" type="explore" respawn="false" ...
    name="explore" output="screen">
  <param name="robot_base_frame" value="base_link"/>
  <param name="costmap_topic" value="move_base/global_costmap/costmap"/>
  <param name="costmap_updates_topic" ...
      value="move_base/global_costmap/costmap_updates"/>

  <param name="visualize" value="true"/>
  <param name="planner_frequency" value="0.2"/>
  <param name="progress_timeout" value="120.0"/>
  <param name="min_frontier_size" value="2.0"/>
  <param name="gain_scale" value="2.5"/>
  <param name="potential_scale" value="1.0"/>
  <param name="orientation_scale" value="0.0"/>

  <param name="transform_tolerance" value="0.3"/>
  <param name="track_unknown_space" value="true" />
</node>
</launch>
```

Key Params:

- **planner_frequency**: Defines how frequently explore_lite computes new exploration goals (in Hz). By reducing this value, the system plans less frequently, which helps avoid frequent re-planning and directional changes during exploration.
- **progress_timeout**: Specifies the maximum amount of time (in seconds) the robot is allowed to make no observable progress toward its current goal before a new goal is selected. Increasing this value ensures that the drone has sufficient time to reach distant or complex frontiers without being prematurely redirected.
- **min_frontier_size**: Sets the minimum size (in meters) for a frontier to be considered valid. Larger values filter out small or insignificant frontiers that could distract the exploration planner, resulting in more stable and meaningful goal selection.
- **gain_scale**: Determines the influence of a frontier's information gain (i.e., the amount of unknown space it reveals) on the goal selection process. A higher value increases the preference for larger unexplored areas, guiding the drone toward richer regions.
- **potential_scale**: Adjusts the weight given to the path cost (distance and accessibility) when evaluating frontiers. Lowering this parameter reduces the bias toward closer frontiers, allowing the drone to pursue potentially more valuable frontiers even if they are farther away.
- **orientation_scale**: Controls the influence of the robot's orientation in the goal evaluation process. It is intentionally left at zero to allow free orientation selection without penalizing direction changes.

The configured parameters influence the drone's behavior during exploration. The system is set to explore unknown space (track_unknown_space = true), and frontiers smaller than 2 m^2 are ignored (min_frontier_size = 2.0) to avoid noise or insignificant openings.**Goal selection** is primarily driven by **frontier size** (gain_scale = 2.5) and, to a lesser extent, by **proximity** (potential_scale = 1.0), while orientation is disregarded (orientation_scale = 0.0). The planner_frequency is set to 5 Hz, allowing the node to **replan frequently and respond promptly** to changes in the environment. Additionally, if no progress is made toward the selected goal within 120 seconds (progress_timeout = 120.0), the current goal is discarded and a new frontier is chosen. This configuration supports continuous and efficient exploration, ensuring that the drone actively seeks out unexplored areas while avoiding stagnation.
In this case, no official tuning guide is available, and the configuration was defined through trial and error by evaluating the algorithm's performance and the results obtained from the mapping process.

4.4 Octomap for 3D Map Visualization

An interesting addition, which will be essential in the third scenario, is **OctoMap**. OctoMap is an open-source library for **3D mapping and spatial representation**, based on a data structure called Octree, which enables efficient modeling of three-dimensional space.
OctoMap is particularly useful in **robotics and autonomous navigation**, as it allows the construction of 3D maps from sensor data such as **LiDAR** or depth cameras.

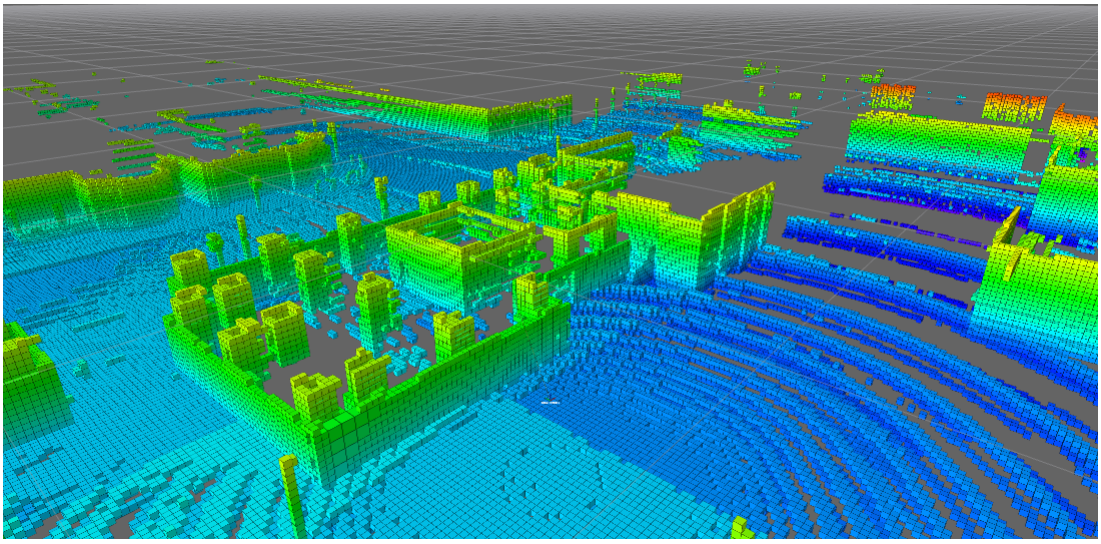

Figure 13: octomap's generated map

The map obtained is the result of data coming from the LiDAR and the resolution chosen for OctoMap, which in this case is 0.2, meaning each voxel is $0.2^3 m^3$.

```
    def add_lidar_sensor(self):
        lidar_bp = self.world.get_blueprint_library().
                            find('sensor.lidar.ray_cast')
        lidar_bp.set_attribute('range', '125.0')
        lidar_bp.set_attribute('channels', '32')
        lidar_bp.set_attribute('points_per_second', '2000000')
        lidar_bp.set_attribute('rotation_frequency', '20')
        lidar_bp.set_attribute('upper_fov', '0')
        lidar_bp.set_attribute('lower_fov', '-60')
```

5 Unexplored map with full navigability

In the last scenario considered, the constraint that restricted the drone to the XY plane is removed. Therefore, all the assumptions from the previous scenario are maintained, but this time the drone can move along the Z-axis and potentially perform rotations around the X and Y axes (pitch and roll).
The software used includes **Cartographer** for the SLAM algorithm, configured for Localization-Only, and **Aeplanner**, an exploration planning package for 3D environments.[1]
The strategy relies on representing the environment through an **information graph**, where nodes correspond to reachable positions, and edges define valid connections between them. The method adopts a **Next-Best-View (NBV)** approach to identify the most informative points to explore, maximizing map coverage while minimizing resource consumption.
The algorithm incorporates **Rapidly-exploring Random Trees (RRT)** for efficient path generation and an **optimization framework** to minimize exploration time. The robot continuously updates its map using sensor data (LiDAR, IMU, etc.), enhancing navigation and reducing redundant exploration. This results in a faster and more efficient exploration process, especially suitable for complex environments such as urban or industrial areas.

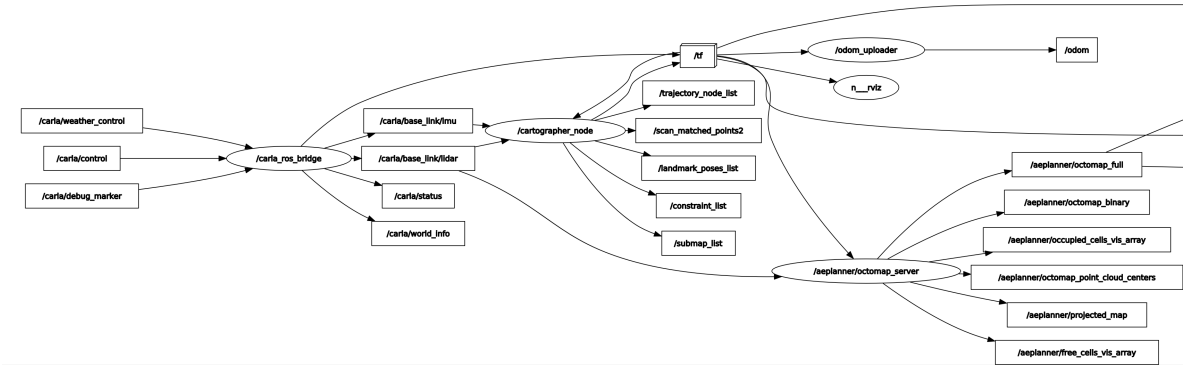

Figure 14: rqt_graph 1

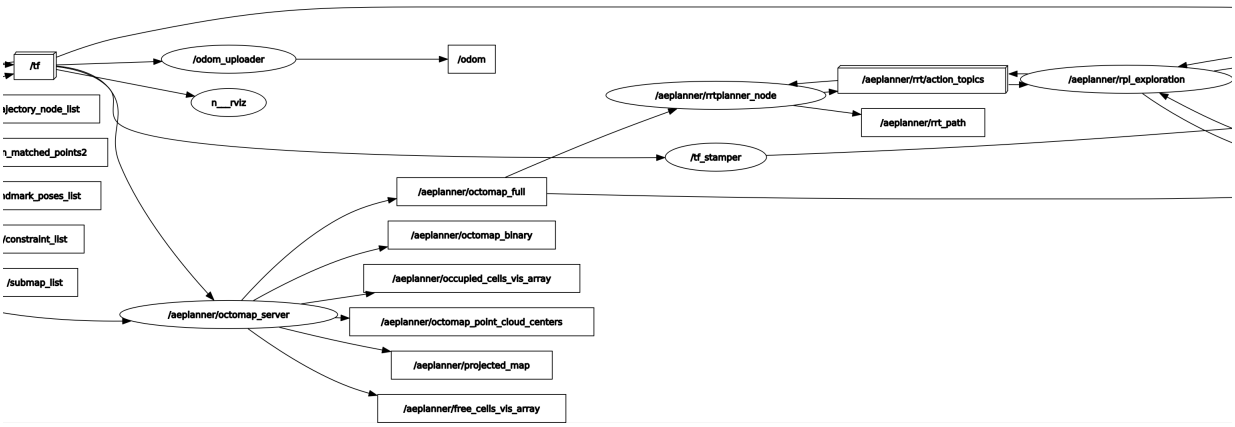

Figure 15: rqt_graph 2

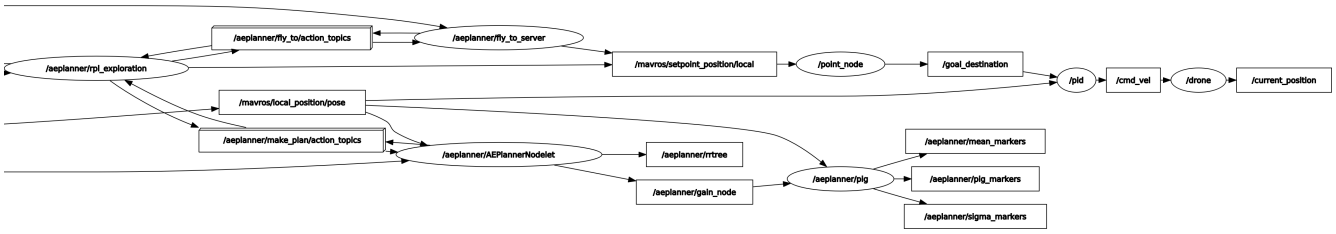

Figure 16: rqt_graph 3

Aeplanner is a fully developed and ready-to-use package; however, it requires the drone's transformation with respect to the fixed reference frame to be available on /tf, accompanied by a timestamp. To meet this requirement, an additional node, tf_stamper, was created to publish the transformation with an updated timestamp, ensuring compatibility with Aeplanner.
As in the second scenario, Cartographer was employed to localize the drone within the environment using data from the LiDAR and IMU. This enables the drone to estimate its position in real time and update the map dynamically as it explores the three-dimensional space.
And it is precisely with **Cartographer** that issues arise in this scenario. As clearly demonstrated in this [GitHub issue page], Cartographer fails to accurately map or localize the drone when it moves vertically. This results in errors in the **pose estimation**, which in turn disrupt the proper functioning of the control loop.
To illustrate this problem, we **disabled the drone's ability to rotate** in the class responsible for its movement. This class, aside from a few modifications, remains identical to the one used in previous scenarios. The drone was then commanded to **ascend vertically at an extremely low speed of 20 cm/s**.
As shown in **Figures 16** and **17**, the visualization of LiDAR data in **RViz** appears as if the drone's reference frame (base_link) were tilted, despite it actually moving strictly along the vertical axis. **Figures 18** and **19** show screenshots **comparing** the drone's **pose data** at two different time instants. On the **left**, the pose **estimate from Cartographer** is displayed, clearly showing that, according to Cartographer, the base_link frame appears tilted. On the **right**, the **actual drone position** is shown, retrieved through the **CARLA API**.
The **position offset** is due to the fact that the drone is initially spawned at coordinates (190, 240, 0.5), whereas, as expected, the SLAM algorithm assumes the starting point as the origin.

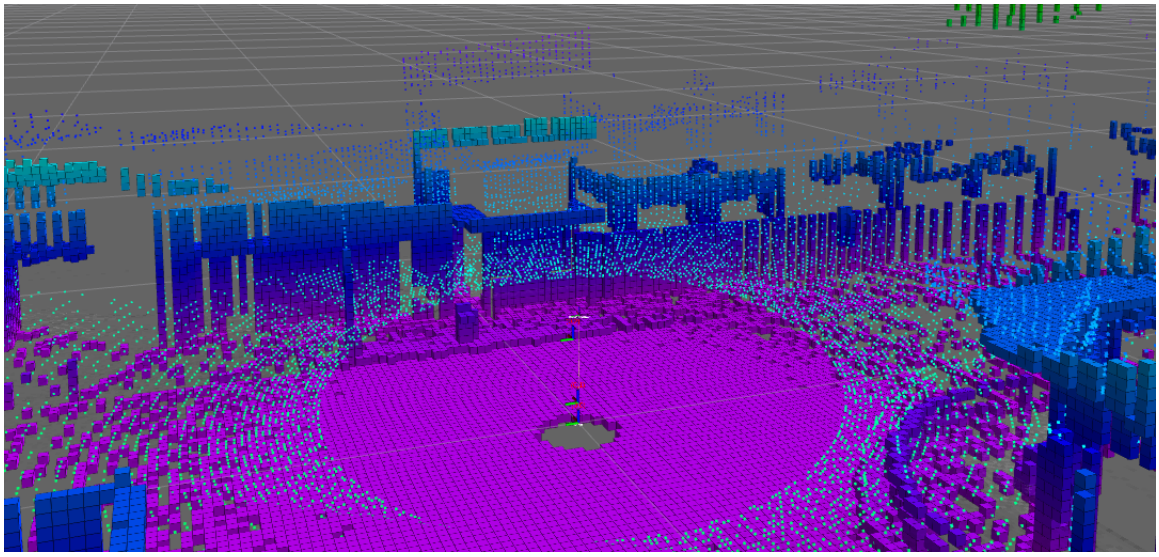

Figure 17: RViz screenshot showing Cartographer problem

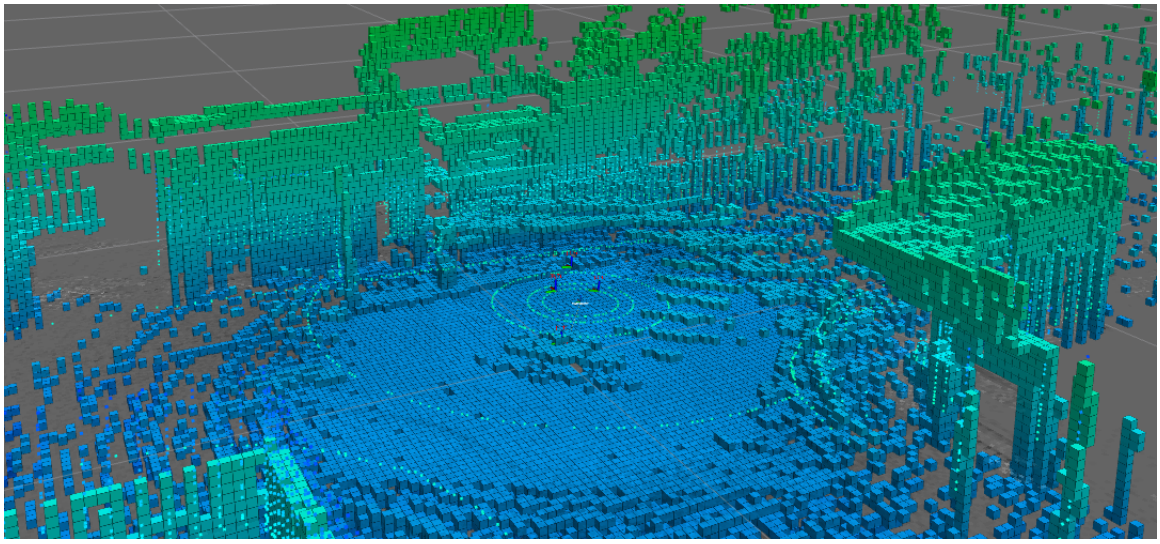

Figure 18: RViz screenshot showing some time instants later

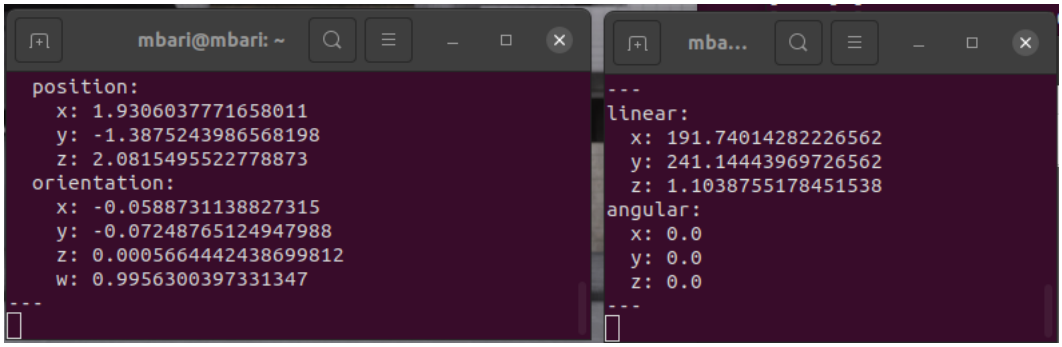

Figure 19: Left: Cartographer's /tracked_pose Right: /current_pose

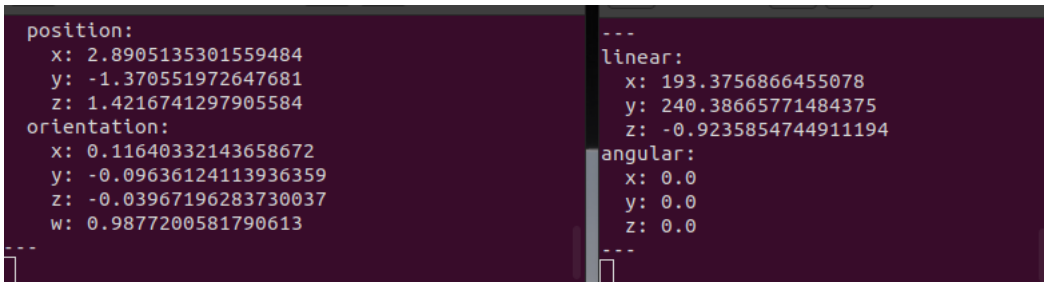

Figure 20: Left: Cartographer's /tracked_pose Right: /current_pose

After exploring alternative solutions and testing different approaches, such as **LIO-SAM**, without obtaining satisfactory results, an attempt was made to modify the initial problem assumptions while continuing to use **Cartographer**.
The following adjustments were implemented:

- The **maximum velocity** of the drone along all axes was further reduced to **5 cm/s**, in order to achieve slower and more stable movements. This significantly improved the quality of pose estimation by minimizing errors caused by rapid motions, which can degrade scan quality and data consistency.
- The **number of iterations** required to generate a **submap** was increased **from 30 to 1000**. This adjustment allows Cartographer to accumulate more data before consolidating a submap, resulting in higher-quality local maps and reduced alignment errors.
- The **iterations for pose estimation** were raised **from 90 to 10,000**, enabling more thorough optimization during the scan matching process. This contributed to improved localization accuracy, especially in complex or sparsely structured environments.
- The **rotation constraint** was maintained, effectively transforming the **pose control** (with the reference pose from **Aeplanner**) **into position control**.

Since the **LiDAR** mounted on the drone scans the environment in a ring around the vehicle, this constraint did not introduce issues with input data.
To obtain these new values, in addition to following the **Cartographer tuning guide** and reviewing information on **GitHub issue pages**, we conducted a series of trial-and-error tests until achieving positive results in simulation.
After **tuning the parameters** of both Cartographer and Aeplanner, significantly **better results** were observed compared to previous attempts.

From **Figure 21**, it can be observed that the **orientation suggested by Aeplanner** is completely **ignored** in the generation of velocity commands for the drone.
Additionally, while the **tilted pose estimation issue** still persists, its impact is **significantly reduced** due to the recent modifications. Unlike previous cases, this **no longer affects the generation of Cartographer's submaps**. Instead, the submaps are now correctly stacked one on top of the other, depending on the drone's altitude at the time of generation.

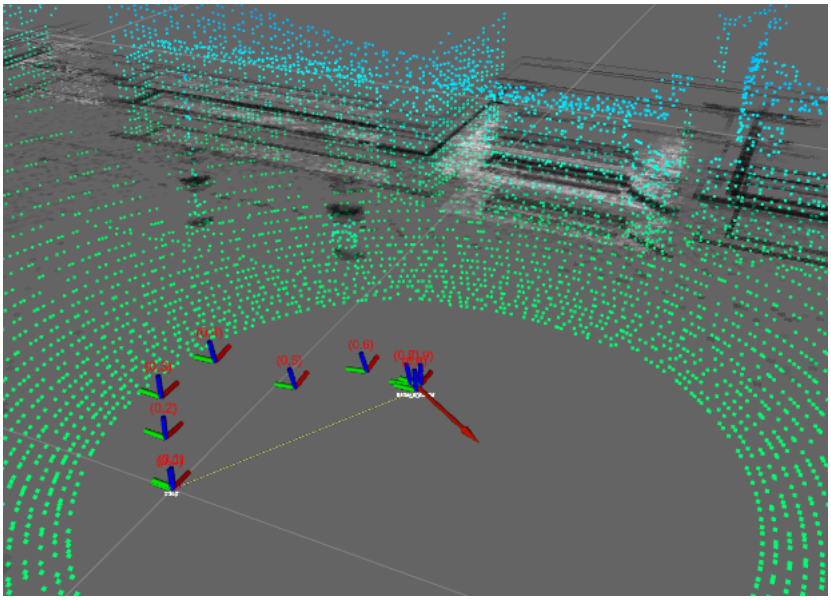

Figure 21: Position control instead of pose control

In this scenario, to visualize the trajectory in RViz, it was necessary to create a node that subscribed to /tracked_pose, a topic generated by Cartographer of type PoseStamped, where the pose estimate is published in addition to /tf. Based on the information contained in /tracked_pose, the node generated a new topic /drone_path of type Path, allowing RViz to subscribe and display the trajectory.

```
class PathPublisher:
    def __init__(self):
        # Pubblicazione del path
        self.path_pub = rospy.Publisher('/drone_path', Path, ...
            queue_size=10)
        self.path = Path()
        self.path.header.frame_id = "map"  # Imposta il frame di ...
            riferimento a "map"

        # Inizializza il subscriber per il topic /tracked_pose
        rospy.Subscriber("/tracked_pose", PoseStamped, ...
            self.pose_callback)

    def pose_callback(self, msg):
        # Aggiungi la nuova posa al path
        self.path.poses.append(msg)
        self.path.header.stamp = rospy.Time.now()
        # Pubblica il path aggiornato
        self.path_pub.publish(self.path)
```

```
if __name__ == '__main__':
    rospy.init_node('path_publisher')
    PathPublisher()
    rospy.spin()
```

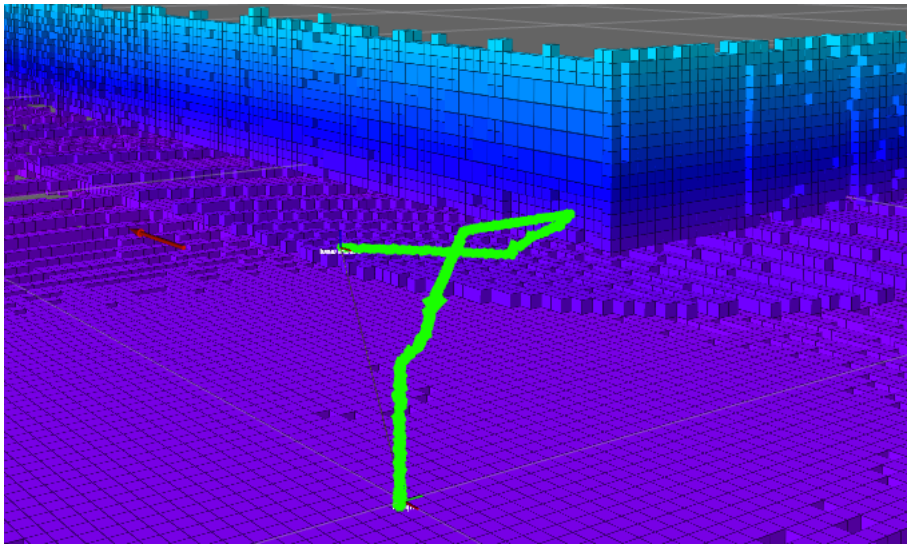

Figure 22: Screenshots during aeplanner runtime, approx. 9 minutes

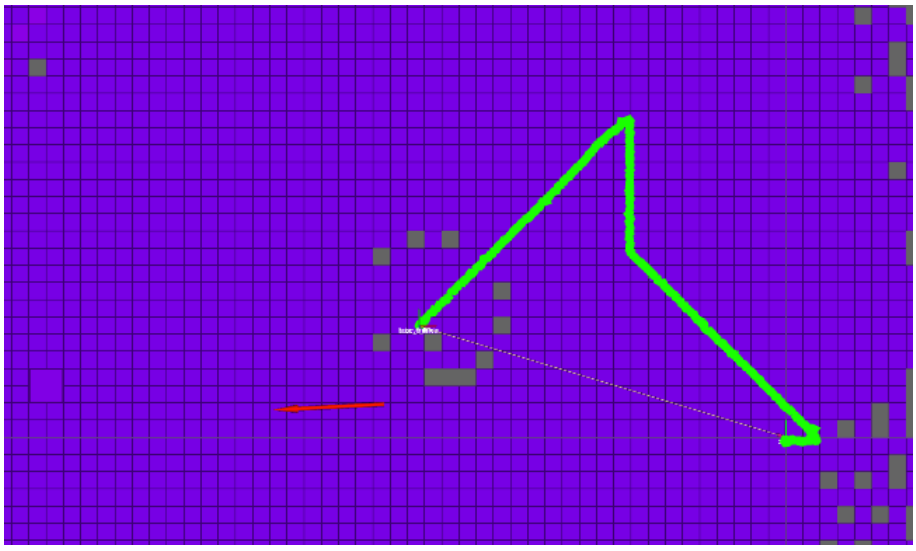

Figure 23: Same trajectory seen from above

5.1 Aeplanner's Config File

The configuration file used for Aeplanner was modified to suit the specific requirements of our case study.

```
raycast/dr:      1
raycast/dphi:    5
raycast/dtheta: 5

system/bbx/r: 1.5
system/bbx/overshoot: 0.3

aep/gain/r_min: 0
aep/gain/r_max: 75
aep/gain/zero: 25
aep/gain/lambda: 0.3
aep/gain/sigma_thresh: 0.2

aep/tree/extension_range: 4
aep/tree/max_sampling_radius: 50
aep/tree/initial_iterations: 10
aep/tree/cutoff_iterations: 150

rrt/min_nodes: 100

boundary/min: [-500.0, -500.0, -2.0]
boundary/max: [ 500.0,  500.0, 5.0]


# Frame di riferimento
robot_frame: "base_link"
world_frame: "map"
```

The parameters used are thoroughly explained in the paper *Efficient Autonomous Exploration Planning of Large-Scale 3-D Environments*[1], authored by the developers of Aeplanner. Below is a brief summary of the purpose of each parameter:

- **raycast/dr**: The radial increment (in meters) used for stepping along each ray during raycasting. Smaller values increase the resolution of the ray traversal but also increase computational cost.
- **raycast/dphi**, **raycast/dtheta**: Angular resolutions (in degrees) for horizontal (phi) and vertical (theta) directions. These control the density of the rays cast from the sensor. Lower values result in denser raycasting, capturing finer details, but with higher computational overhead.
- **system/bbx/r**: The radius (in meters) of the local bounding box used for local planning and evaluation of candidate viewpoints. A larger value allows the planner to consider a broader area for exploration.
- **system/bbx/overshoot**: A buffer added to the bounding box to allow some tolerance when planning paths, helping the system avoid getting stuck at the edge of the planning volume.
- **aep/gain/zero**: Distance at which the gain value is zero; typically, this acts as a midpoint in a decay function.
- **aep/gain/lambda**: Weighting factor that penalizes gain based on the required travel distance; higher values prioritize faster or closer movement over more informative but distant points.

- **aep/gain/sigma_thresh**: Threshold for including voxels in gain computation based on their uncertainty (sigma). Lower values include voxels with less uncertainty, allowing the system to focus on covering known areas more thoroughly.
- **aep/tree/extension_range**: The maximum step size (in meters) that a tree branch can extend towards a sampled point. Controls the growth granularity of the tree.
- **aep/tree/max_sampling_radius**: The maximum radius within which new points are randomly sampled for tree expansion.
- **aep/tree/initial_iterations**: The number of initial iterations the RRT runs before evaluating the tree for a valid exploration candidate.
- **aep/tree/cutoff_iterations**: The maximum number of iterations allowed before the tree search is terminated. Higher values allow for a more exhaustive search but at a computational cost.
- **rrt/min_nodes**: The minimum number of nodes the RRT must contain before the planner considers terminating or declaring a failure. Ensures the planner has explored sufficiently before making a decision.
- **boundary/min**, **boundary/max**: Define the 3D bounds of the exploration environment. These prevent the planner from attempting to explore outside the valid or simulated area.

As the performance is highly dependent on the environment in which the drone operates, there is **no definitive tuning guide**. Therefore, to arrive at the values presented, we started from the default settings provided by the developers and **adjusted the parameters** in a way that appeared appropriate, increasing or decreasing each value based on the results observed in simulation.

5.2 Solving Aeplanner's small problems

The application for autonomous navigation in 3D space using **Aeplanner** was originally developed to work with drones integrated through the **MAVROS** library. Therefore, it is expected that minor adjustments are required to ensure full compatibility with the current setup.
The first issue encountered concerns the topic that contains the drone's target pose: /mavros/setpoint_position/local. The messages published on this topic, which are of type geometry_msgs/PoseStamped, were missing the "frame_id" attribute in their header. As a result, a custom node was implemented to subscribe to this topic and republish the same messages, adding "$frame\_id = map$" to the header.
This ensures that ROS correctly interprets the coordinate frame in which the target pose is defined.

```
def callback(msg):
    corrected_msg = PoseStamped()
    corrected_msg.header.stamp = rospy.Time.now()
    corrected_msg.header.frame_id = "map"  # Imposta il frame_id ...
        corretto
    corrected_msg.pose = msg.pose  # Copia la posa originale

    pub.publish(corrected_msg)


if __name__ == '__main__':
    rospy.init_node('fix_setpoint_frame_node')
```

```
    pub = rospy.Publisher('/goal_destination', PoseStamped, ...
        queue_size=10)
    sub = rospy.Subscriber('/mavros/setpoint_position/local', ...
        PoseStamped, callback)

    rospy.loginfo("Nodo fix_setpoint_frame_node avviato...")
    rospy.spin()
```

Another issue involves the input topic /mavros/local_position/pose. Aeplanner requires the drone's pose—estimated by the SLAM algorithm, in this case Cartographer—to be provided as a PoseStamped message. However, Cartographer only publishes the pose estimate through the /tf transform tree.
To address this, a custom node was implemented that subscribes to /tf and, at a fixed interval of 0.1 seconds (this frequency is configurable, but no specific value is suggested in the Aeplanner documentation, so this value was chosen arbitrarily), constructs a PoseStamped message using the data extracted from the transform, and republishes it on the topic /mavros/local_position/pose. To retrieve the transformation from the TF tree and convert it into a PoseStamped message, it is therefore appropriate to use a tf2_ros.Buffer along with a tf2_ros.TransformListener. This ensures that Aeplanner receives the drone's pose in the expected format and coordinate frame.

```
def tf_callback():
    try:
        trans = tf_buffer.lookup_transform("map", "base_link", ...
            rospy.Time(0))
        pose_stamped = PoseStamped()
        pose_stamped.header = trans.header
        pose_stamped.pose.position = trans.transform.translation
        pose_stamped.pose.orientation = trans.transform.rotation
        pub.publish(pose_stamped)
    except (tf2_ros.LookupException, tf2_ros.ConnectivityException, ...
        tf2_ros.ExtrapolationException):
        rospy.logwarn("Nessuna trasformata /tf ricevuta")

if __name__ == '__main__':
    rospy.init_node('tf_to_posestamped')
    pub = rospy.Publisher('/mavros/local_position/pose', ...
        PoseStamped, queue_size=10)
    tf_buffer = tf2_ros.Buffer()
    tf_listener = tf2_ros.TransformListener(tf_buffer)

    rate = rospy.Rate(10)  # 10 Hz
    while not rospy.is_shutdown():
        tf_callback()
        rate.sleep()
```

6 Using CARLA to Generate New And Moving Obstacles

Until now, in both the second and third scenarios, the drone has been able to navigate through space, avoiding obstacles and pre-existing traffic signage within the CARLA world. The initial reason for choosing CARLA to simulate the drone's flight, rather than using something like Gazebo, was the **ability to generate vehicles** and make them **move** independently on the map. This allows these vehicles to be considered **dynamic obstacles** that the drone must detect, avoid, and map.

```
# Connessione al server CARLA
client = carla.Client('localhost', 2000)
client.set_timeout(10.0)
world = client.get_world()
map = world.get_map()

# Ottieni tutti i punti di spawn validi: su corsia e con collegamenti
all_spawn_points = map.get_spawn_points()
valid_spawn_points = []
for p in all_spawn_points:
    wp = map.get_waypoint(p.location, project_to_road=True, ...
        lane_type=carla.LaneType.Driving)
    if wp and wp.next(1.0):
        valid_spawn_points.append(p)

# Mischia e seleziona i primi N spawn point disponibili
num_vehicles = min(40, len(valid_spawn_points))
random.shuffle(valid_spawn_points)
selected_spawn_points = valid_spawn_points[:num_vehicles]

# Ottieni la libreria dei blueprint dei veicoli
blueprint_library = world.get_blueprint_library()
vehicle_blueprints = blueprint_library.filter('vehicle.*')

# Inizializza il Traffic Manager
traffic_manager = client.get_trafficmanager(8000)
traffic_manager.set_hybrid_physics_mode(True)
traffic_manager.set_global_distance_to_leading_vehicle(5.0)
# traffic_manager.global_percentage_speed_difference(20.0)  # ...
    (opzionale)

spawned_vehicles = []

# Spawna i veicoli
for i, spawn_point in enumerate(selected_spawn_points):
    blueprint = random.choice(vehicle_blueprints)

    # Colore casuale (se supportato)
    if blueprint.has_attribute('color'):
        color = random.choice(blueprint.get_attribute('color')
                                                .recommended_values)
        blueprint.set_attribute('color', color)

    try:
        vehicle = world.spawn_actor(blueprint, spawn_point)
        vehicle.set_autopilot(True, traffic_manager.get_port())
        spawned_vehicles.append(vehicle)
        print(f"Veicolo {i+1} spawnato a {spawn_point.location}")
    except Exception as e:
        print(f"Errore nello spawn del veicolo {i+1}: {e}")

print(f"\n{len(spawned_vehicles)} veicoli spawnati con successo.")
print("Simulazione in esecuzione. Premi CTRL+C per terminare.")

# Ciclo principale per mantenere la simulazione attiva
try:
```

```
        while True:
            time.sleep(1.0)
    except KeyboardInterrupt:
        print("\nInterruzione manuale ricevuta. Distruzione dei veicoli...")

    # Distruzione dei veicoli alla fine
    finally:
        for v in spawned_vehicles:
            v.destroy()
        print("Tutti i veicoli sono stati distrutti.")
```

The provided code aims to **generate** a number of **vehicles** within a simulated world in **CARLA**, ensuring that these vehicles autonomously follow traffic rules. After establishing a connection with the CARLA server via Python, the code retrieves the map and available spawn points within the simulated scene. Then, a list of vehicle blueprints is obtained from the CARLA blueprint library, and the **vehicles are spawned at random positions** on the map, **avoiding overlaps** with other actors in the simulated world. To ensure that vehicles do not overlap, the code uses a verification function based on checking the bounding boxes of the objects present.
Once the vehicles are spawned, the code activates the **autopilot** for each vehicle through CARLA's **Traffic Manager**. The Traffic Manager is a module that autonomously manages the behavior of vehicles, ensuring that they follow traffic rules, obey traffic lights, and stay in the correct lanes. The Traffic Manager is configured with additional parameters such as the safety distance between vehicles and a global speed variation, to simulate a realistic traffic situation. Each vehicle is connected to the Traffic Manager through the specified port (in this case, port 8000), allowing the management of their interaction and movement within the simulation. In summary, the code creates autonomous vehicles on the map and enables them to navigate independently within a complex simulated environment.
So, after starting the ROS-CARLA bridge and setting up the sensors on the drone, it is possible to use this script to generate traffic.

Figure 24: CARLA view

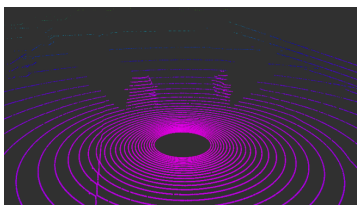

Figure 25: LiDAR view

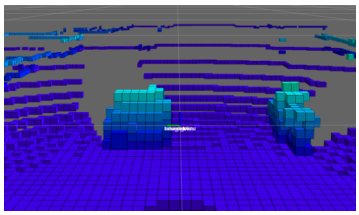

Figure 26: Vortex view

In the **second scenario**, where the drone is constrained to motion on a plane, the addition of this feature requires modifications to the move_base configuration files, specifically by adding the layers described in subsections **4.2.2** and **4.2.3**.
If the **speed** of the vehicles in traffic is set **low enough** to allow the drone **to detect obstacles** within the radius defined in the configuration files (10 meters) and **to compute a new trajectory**, then the drone will be able to successfully avoid the vehicle.
This solution, however, has a **very high computational cost**. In fact, trajectory modifications to avoid vehicles must be executed at **every simulation step**, as the vehicle is constantly moving. When also accounting for the computational load introduced by the explore_lite algorithm and Cartographer, the system slows down to critical levels and, in some cases, completely halts the application.
In the scenario where the drone has full mobility, this issue becomes even more evident, as the computational load of Aeplanner combined with Cartographer is significantly higher. Additionally,

the drone's maximum speed (5 cm/s in this case) poses a further challenge: a significantly larger obstacle detection area would be required to avoid vehicles effectively. However, as one can easily infer, this would exponentially increase the computational load.
Therefore, initially, we detached the spawned vehicles from the traffic manager, effectively making them stationary obstacles, and eventually **abandoned this idea**.

7 Conclusion

This project focused on the development and evaluation of various control systems enabling the autonomous navigation of a drone within a virtual environment, leveraging the CARLA simulator integrated with ROS. By addressing three progressively complex scenarios—pre-explored map, unexplored map with altitude constraints, and fully navigable unexplored map—the project explored different levels of autonomy and control strategies, adapting the system architecture to the specific assumptions of each case.
In the first scenario, the availability of GPS-based localization and a predefined trajectory allowed the implementation of precise trajectory tracking. The subsequent scenarios, which lacked external localization, required the integration of SLAM algorithms based on LiDAR and IMU data, thereby enabling the estimation of the drone's pose in unknown environments. The final scenario, involving full three-dimensional navigation, challenged the system to operate autonomously with minimal prior knowledge, validating the effectiveness of the implemented planning and control framework.
The results demonstrate the potential of combining realistic simulation, advanced SLAM techniques, and modular control strategies to achieve robust autonomous navigation. This work provides a solid foundation for future developments in both simulated and real-world autonomous drone applications.

References

[1] M. Selin et al. "Efficient Autonomous Exploration Planning of Large-Scale 3-D Environments". In: *IEEE Robotics and Automation Letters* 4.2 (Apr. 2019), pp. 1699–1706. ISSN: 2377-3766. DOI: 10.1109/LRA.2019.2897343.