Lab 2 Report

Zi Shengbo 23020036099 Lab group: 32.

Lab date: 14th November 2025. Report date: 15th November 2025.

Summary

The summary provides an overview of this Lab 1 report. It aims to concisely outline the experiment's objectives, key procedures, and main conclusions, serving as a stand-alone section for readers to grasp the report's core content without repeating the introduction. This Lab 1 focuses on Linux system operations, pro- gramming tools, and C++ basics, with the goal of establishing a solid foundation for future technical learning. The report details the experimental process, indi- vidual work, discussions, and conclusions, ensuring clarity and completeness in conveying the learning outcomes.

1. Lab Background

ROS (Robot Operating System) is not a traditional operating system; rather, it is the core middleware for robot development, capable of addressing communication, data management, and functional integration issues among multiple modules (such as sensors, controllers, and actuators) of robots. It serves as the fundamental framework for achieving robot navigation and autonomous control. In the field of vehicle navigation (VNav), robots need to process coordinate system data from various components such as lidar, cameras, and wheel odometers in real time. The tf tool and homogeneous transformation are key technologies for unifying coordinate systems and achieving data fusion. Among them, tf is responsible for tracking the dynamic relationships of different coordinate systems, while homogeneous transformation uses mathematical methods to uniformly represent "translation + rotation". The combination of the two ensures the compatibility of data from different sensors in the same coordinate system, providing accurate data support for subsequent navigation algorithms (such as path planning, positioning). Additionally, the ROS version needs to be compatible with the Ubuntu system (e.g., Ubuntu 20.04 corresponds to ROS Noetic; Ubuntu 18.04 corresponds to ROS Melodic). This experiment is conducted using "Ubuntu 20.04 + ROS Noetic" as an example.

2. Core Goal

1.Master the complete installation process of ROS, including software source configuration and environment variable setup, to ensure that the core service (roscore) of ROS can be started finally.

2.Understand the basic concepts of ROS: nodes, topics, and packages, and be able to verify node communication through the turtle example.

3.Master the core logic of the tf tool (coordinate system tree management, transformation publishing and listening), and understand the mathematical principle of homogeneous transformation (4×4 matrix structure).

4.Complete 5 practical exercises, specifically including installing ROS, using TF to control unmanned movement, controlling the unmanned aircraft to move according to the trajectory, conducting relevant mathematical derivations (such as proving that the AV2 trajectory is a parabolic arc, calculating relative positions, etc.), and exploring quaternion properties (such as proving that $\Omega_1(q)$ and $\Omega_2(q)$ are orthogonal matrices, etc.).

3. Lab Content

PART 1: Install ROS and Successfully Run the Turtlebot Simulation

The starting point of the entire LAB is the installation of ROS. First, open Ubuntu 20.04 in VMware Workstation Pro. Open the terminal, and enter the following commands one by one in the terminal: "sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list"', "curl -s https: //raw. Githubusercontent .com/ros/rosdistro/master/ros.asc | sudo apt-key add -".At this point, an error will occur indicating that curl has not been downloaded and installed. Therefore, the following commands should be used to download, install and update curl: "sudo apt update", "sudo apt install curl". After the download is complete, re-add the key and then continue with the installation of ROS: "sudo apt update", "sudo apt install ros-noetic-desktop-full".After a long wait, ROS has been successfully installed on Ubuntu. Next, set the ROS environment variables and install the additional dependencies. Use the following command: "echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc", "source ~/.bashrc", "sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-vcstool build-essential python3-catkin-tools python-is-python3". Once everything is ready, you can initialize rosdep: "sudo rosdep init", "rosdep update".Now I can fully use ROS on Ubuntu.

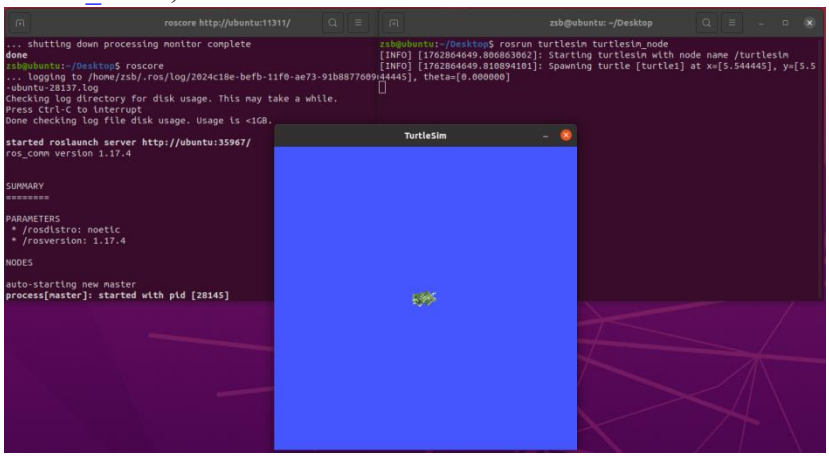
Next, we need to consult relevant materials to understand the composition, functions, and operations of the ROS file system. The ROS file system is organized in a hierarchical structure, with the basic unit being the ROS package (package), which serves as the atomic building and publishing unit of ROS software. Each package contains resources such as ROS runtime processes (nodes), libraries, configuration files, etc., and records the metadata information of the package through the package manifest file (package.xml), including the author, license, dependencies, etc. Messages (message) and services (service) are two important communication mechanism definitions in ROS: message files (.msg) define the data structure for data transmission between processes, while service files (.srv) define the data types for request/reply interactions. The typical organization structure of a ROS package includes directories such as the configuration file directory (config), the startup file directory (launch), the mesh resource directory (mesh), and the source code directory (src).

The workspace, as a centralized development environment, provides a unified directory structure for compiling multiple related software packages. This design enables developers to manage and compile multiple interdependent packages simultaneously, greatly facilitating the development and deployment of complex systems. In the workspace, the compilation system can convert source code into executable files and set up the corresponding runtime environment.

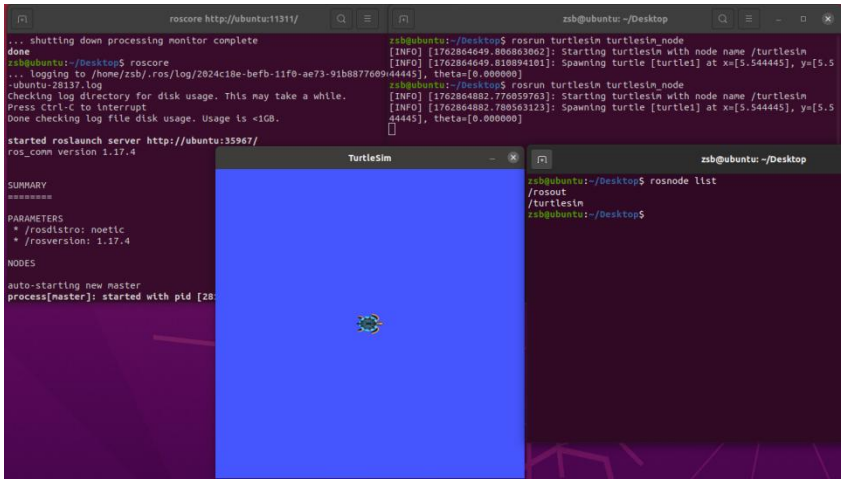
ROS adopts a distributed communication architecture, with its core being the ROS Master, which acts as a naming and registration service center for nodes, similar to the role of a DNS server. When a node starts, it registers its own information with the Master. The Master is responsible for tracking the status changes of all running nodes. In the topic communication mechanism, publishers and subscribers match through the Master: Publishers first register topic information with the Master, and the Master returns the matching subscriber information. Then, both parties establish direct communication based on the TCPROS protocol. This design achieves decoupling between the communicating parties, allowing nodes to dynamically join or exit the system while ensuring the efficiency of data transmission.

Nodes, as the basic execution units in the ROS system, each typically handles a specific functional module of the system, such as sensor data acquisition or data processing. This modular design enables the system to have excellent scalability and maintainability. Nodes can be distributed across different computing devices, and distributed computing across machines is achieved through coordination by the ROS Master. The entire architecture not only ensures the flexibility of the system but also ensures the collaborative work among various components through standardized communication mechanisms.

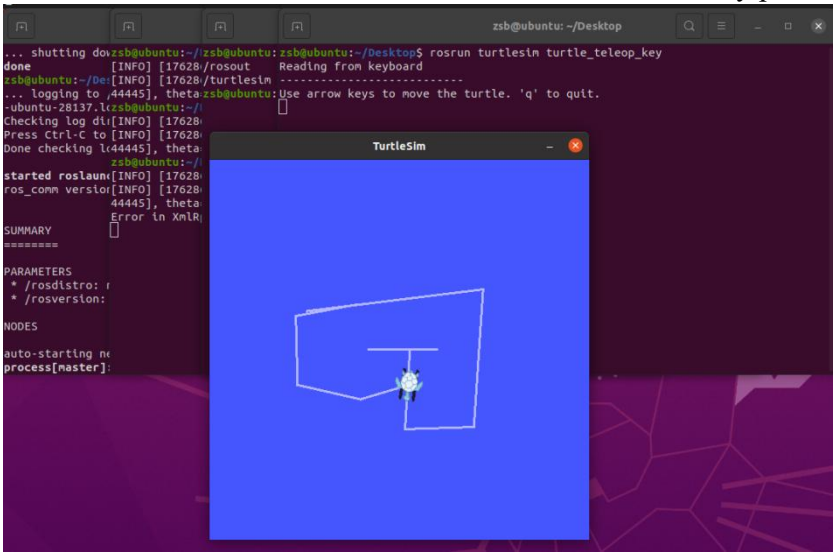
Taking the simulation of baby turtles as an example, I learned the usage methods and basic functions of ROS. To start ROS master, open a terminal and run: “roscore”,ROS has been started normally(Exiting by pressing “ctrl+c”). Open another terminal and enter the following command: “ rosrun turtlesim turtlesim_node”,Then I can see a vivid animation:



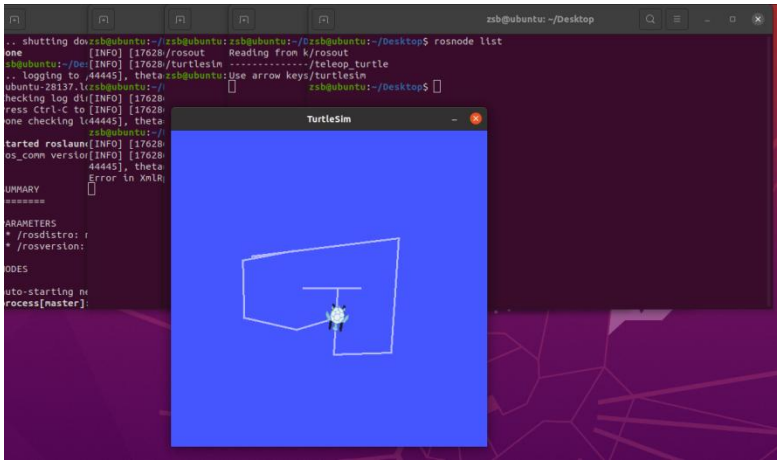
Then, I need to open a new terminal to ask the ROS master about the running nodes with: “rosnode list”, it returns me:



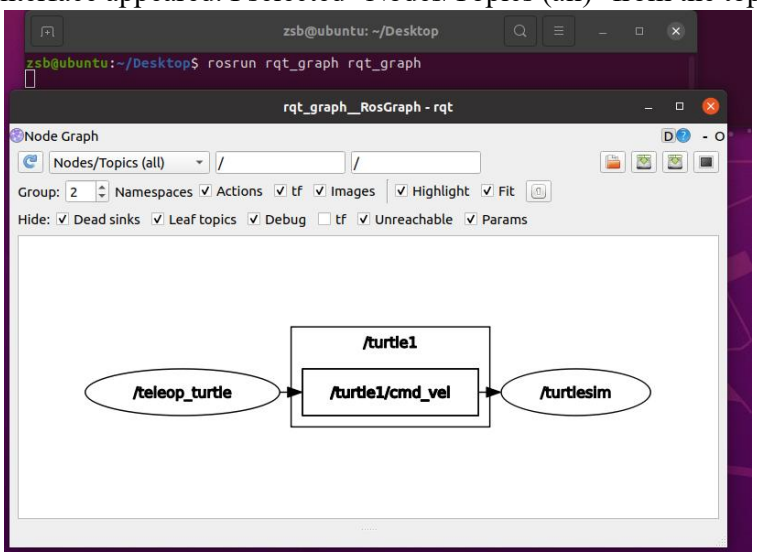
I saw the processes that are currently running in their current state. In order to make the turtle move, I opened a new terminal again and entered the command: “rosrun turtlesim turtle_teleop_key”. I found that in this terminal, I can use the arrow keys to control the movement of the turtle, the left and right keys to control its rotation, and the up and down keys to control its movement. The turtle's movement will leave a trail. And I can exit this mode by pressing "q".



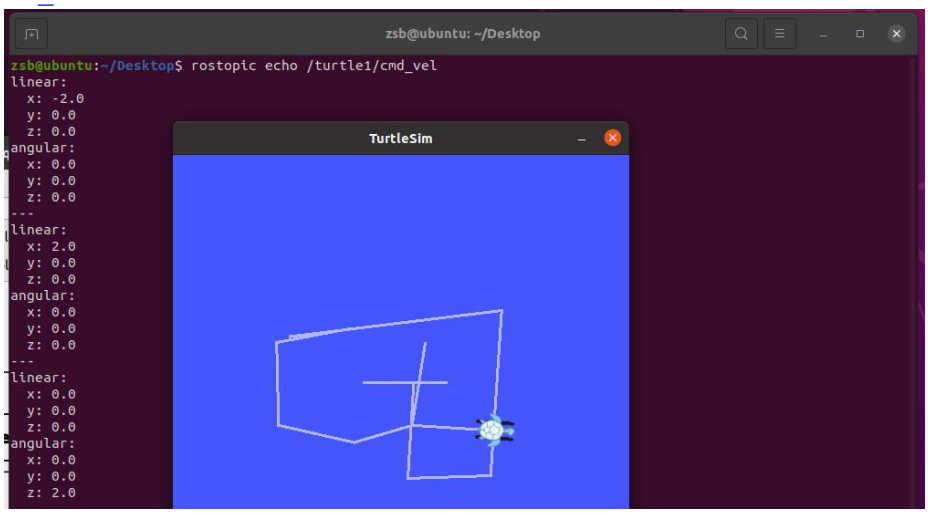
To confirm that this new process is indeed underway, I once again used“rosnode list”, and found that at this point there was an additional node running, which is the current node controlling the movement of the turtle.



If I want to know what the specific code for implementing this function looks like, I need to look it up in the "Topics" section. A topic is the way a node transmits data, representing the channel through which the message is sent, and each topic has an associated message type (different types of messages cannot be sent within the same topic). In ROS, the generation and consumption of data are independent of each other, which means a node can either publish messages (i.e., the producer) or subscribe to a certain topic (i.e., the consumer). So I open a new terminal and use "rosrun rqt_graph rqt_graph" which shows the nodes and topics currently running. A new and complex interface appeared. I selected "Nodes/Topics (all)" from the top left. I saw:



In this chart, the ellipse represents a node and the square represents a topic. It is quite obvious from the diagram that teleop_turtle is publishing data to the /turtle1/cmd_vel topic. The node /turtlesim will subscribe to this topic and use the received messages to move the turtle. I can print these contents to the terminal, and this requires the use of the command: "rostopic echo/turtle1/cmd_vel". I obtain such data:

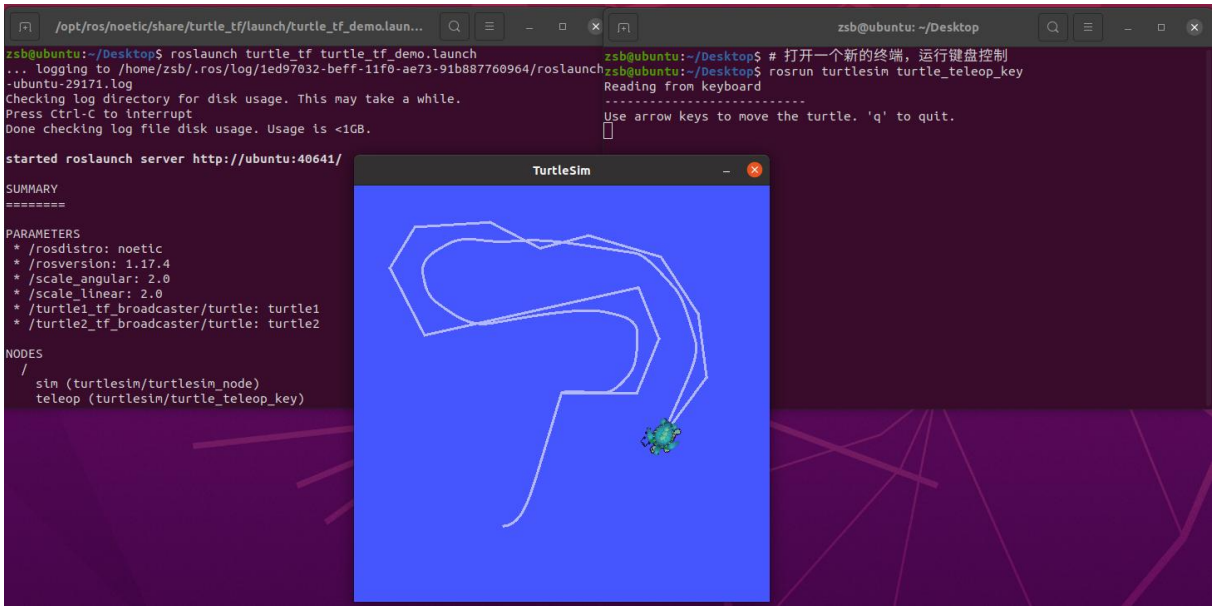


By printing the real-time data to the terminal, I can observe the running results of the code in real time. This is very beneficial for me to debug efficiently.

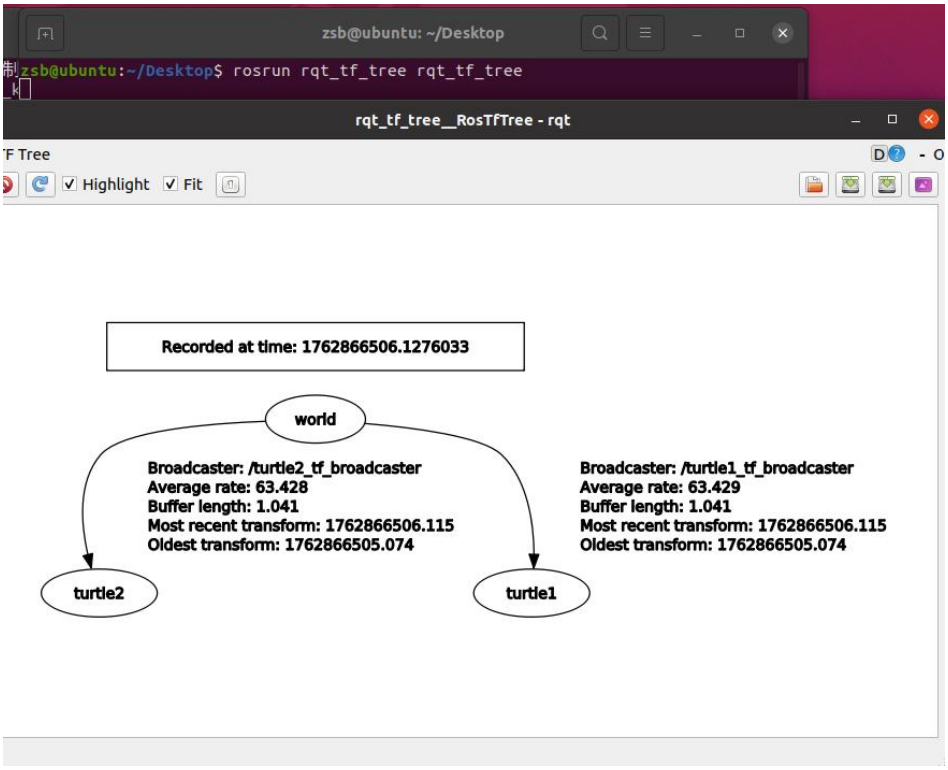
The tf2 library of ROS has been developed, aiming to provide a standard method for tracking coordinate systems and converting data throughout the system, enabling users to have confidence in the consistency of data in a specific coordinate system without needing to understand all other coordinate systems and their relationships within the system.

tf2 is distributed across various nodes (and eventually on different machines as well), and tf2 nodes come in two types.- Listeners: responsible for listening to the `/tf` topic and caching all the data collected (until the cache limit is reached) - Broadcasters: responsible for publishing transformation information between coordinate systems on the `/tf` topic

In tf2, transformations and coordinate systems are represented as a graph, where transformations are represented as edges and coordinate systems as nodes. The advantage of this representation is that the relative orientation between two nodes is simply the product of the edges connecting those two nodes. Another advantage of the tree structure is that it is convenient for dynamic changes. tf handles the ambiguity of transformations and does not allow the formation of cycles in the transformation graph. I ran " roslaunch turtle_tf turtle_tf_demo.launch". In this animation, apart from the turtle I controlled, there was also a new turtle constantly following the turtle I was controlling. I discovered that the new tracking and monitoring route for my turtle is different from what I was controlling. The turtle often chooses the route that is closest to me for its movements. This is in perfect agreement with the fact that the relative orientation between two nodes is simply the product of the edges connecting those two nodes.



In this demonstration application, the ROS TF library is used to create three coordinate systems: a world coordinate system, a coordinate system for Turtle 1, and a coordinate system for Turtle 2. Additionally, a TF publisher is created to publish the coordinate system information of the first Turtle, and a TF listener is set up to calculate the differences in the coordinate systems between the first Turtle and the following Turtle, and to drive the second Turtle to follow the first Turtle. The rqt_tf_tree tool can display the visualized situation of the frame tree published via ROS in real time. I can open a new terminal and enter the command: "rosrun rqt_tf_tree rqt_tf_tree". But at this point, I must ensure that the terminal used to control the movement of the turtle is always in an operational state.

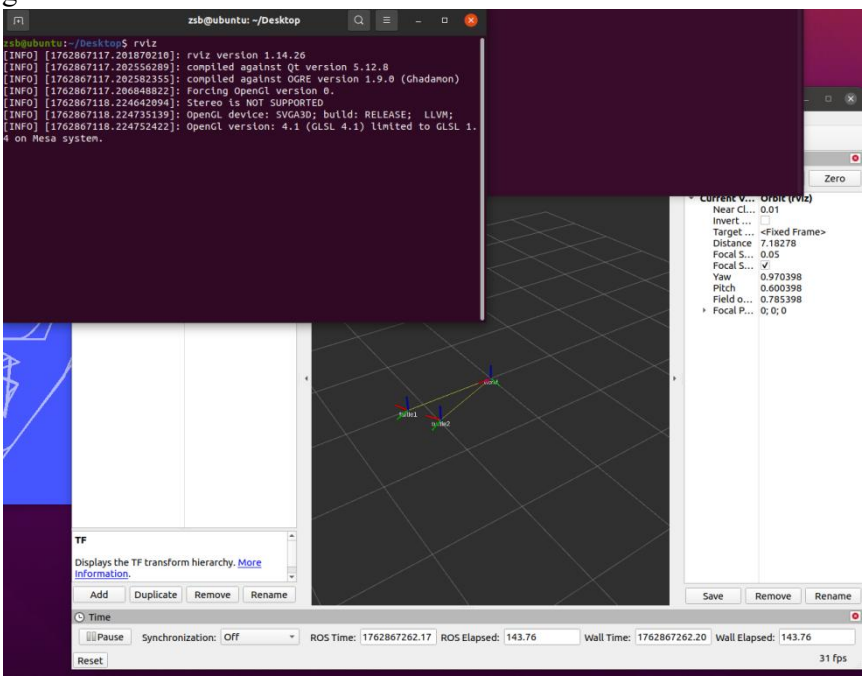


Here, I can see that TF has transmitted three frames - the "World" frame, the "Turtle 1" frame and the "Turtle 2" frame. Among them, the "World" frame is the parent frame of the "Turtle 1" frame and the "Turtle 2" frame. I want to observe the transformation between turtle1, turtle2 and the world frame. I can use "echo" to print the relative pose at each moment in the terminal. I use the command: "rosrun tf tf_echo turtle1 turtle2"

```
zsb@ubuntu:~/Desktop$ rosrun tf tf_echo turtle1 turtle2
At time 1762866640.438
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
At time 1762866641.158
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
At time 1762866642.166
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
At time 1762866643.158
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
At time 1762866644.166
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
At time 1762866645.157
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
At time 1762866646.167
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.940, 0.340]
            in RPY (radian) [0.000, -0.000, 2.447]
            in RPY (degree) [0.000, -0.000, 140.189]
```

The position and orientation at each moment are printed out on the terminal (since I did not move the turtle, there was no change in the position and orientation). The last part of PART1 introduces a powerful tool: RViz is a graphical 3D visualization tool that is useful for viewing the relationship between TF frames within the ROS system. While ensuring that the terminal used to

control the movement of the turtle is running, I opened a new terminal and entered the command: “rviz”. I then opened the RViz interface. After entering the interface, I first needed to change the Fixed Frame to the World Frame, and then click "Add" to incorporate the TF into this interface. From there, I could visually and intuitively see the coordinate axis transformations of the two turtles during the movement.



After completing all the above steps, I gained a general understanding of how to use ROS and its various powerful features. The most important thing is that I directly observed TF, intuitively felt the translation and rotation of the coordinate axes, and learned how to capture the current pose at a certain moment. This plays a crucial foundational role in laying the groundwork for the subsequent experimental content.

PART 2: Control the Movement of the Unmanned Aircraft

I. Set up workspace

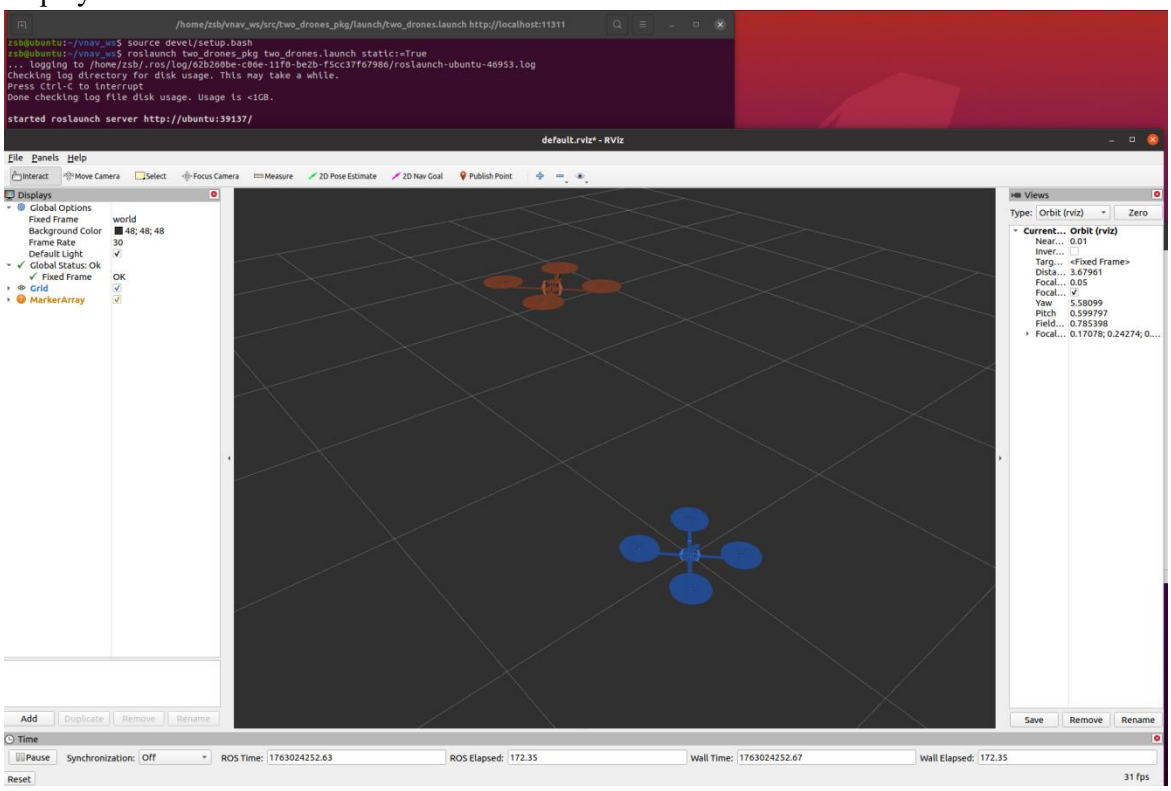
To control the movement of the drone, one must first be able to observe its static state.First, set up a workspace. Open the terminal and use the following commands sequentially: “mkdir -p ~/vnav_ws/src”, “cd ~/vnav_ws/”, “catkin init”.Create a catkin workspace. The sample code will create the workspace folder vnav_ws in the ~/ directory. The following output result is obtained:

```
Initializing catkin workspace in `/home/zsb/vnav_ws`.
----------------------------------------------------------
Profile:                     default
Extending:             [env] /opt/ros/noetic
Workspace:                   /home/zsb/vnav_ws
----------------------------------------------------------
Build Space:       [missing] /home/zsb/vnav_ws/build
Devel Space:       [missing] /home/zsb/vnav_ws/devel
Install Space:      [unused] /home/zsb/vnav_ws/install
Log Space:         [missing] /home/zsb/vnav_ws/logs
Source Space:       [exists] /home/zsb/vnav_ws/src
DESTDIR:            [unused] None
----------------------------------------------------------
Devel Space Layout:          linked
Install Space Layout:        None
----------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
Internal Make Job Server:    True
Cache Job Environments:      False
----------------------------------------------------------
Buildlisted Packages:        None
Skiplisted Packages:         None
----------------------------------------------------------
Workspace configuration appears valid.
----------------------------------------------------------
```

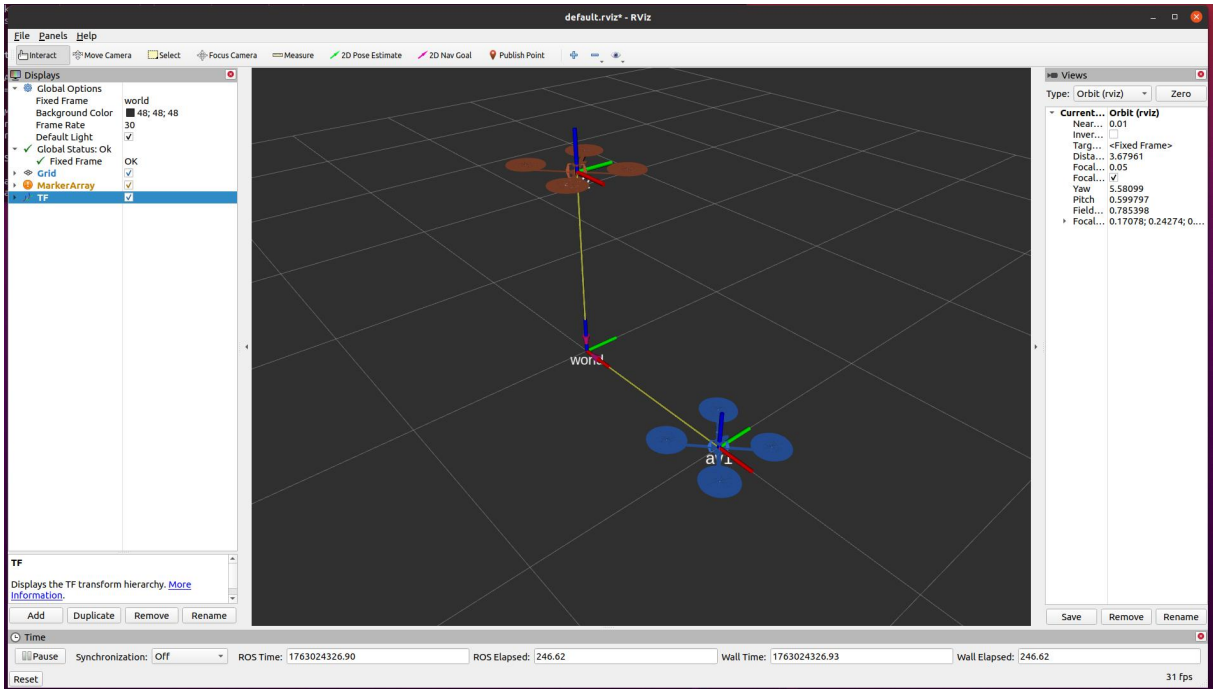
Then, I manually copied the "two_drones_pkg" folder from the downloaded "lab2" file to the "src" directory that I just created. And I used the command: “source devel/setup.bash”Index the compiled files into the terminal to ensure that the terminal can recognize the location of the compiled files.

```
zsb@ubuntu:~/vnav_ws$ catkin build
--------------------------------------------------------------
Profile:                     default
Extending:             [env] /opt/ros/noetic
Workspace:                   /home/zsb/vnav_ws
--------------------------------------------------------------
Build Space:        [exists] /home/zsb/vnav_ws/build
Devel Space:        [exists] /home/zsb/vnav_ws/devel
Install Space:      [unused] /home/zsb/vnav_ws/install
Log Space:         [missing] /home/zsb/vnav_ws/logs
Source Space:       [exists] /home/zsb/vnav_ws/src
DESTDIR:            [unused] None
--------------------------------------------------------------
Devel Space Layout:          linked
Install Space Layout:        None
--------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
Internal Make Job Server:    True
Cache Job Environments:      False
--------------------------------------------------------------
Buildlisted Packages:        None
Skiplisted Packages:         None
--------------------------------------------------------------
Workspace configuration appears valid.

NOTE: Forcing CMake to run for each package.
--------------------------------------------------------------
[build] Found 1 packages in 0.0 seconds.
[build] Updating package table.
Starting  >>> catkin_tools_prebuild
Finished  <<< catkin_tools_prebuild                [ 1.1 seconds ]
Starting  >>> two_drones_pkg
Finished  <<< two_drones_pkg                       [ 4.6 seconds ]
[build] Summary: All 2 packages succeeded!
[build]   Ignored:   None.
[build]   Warnings:  None.
[build]   Abandoned: None.
[build]   Failed:    None.
[build] Runtime: 5.7 seconds total.
[build] Note: Workspace packages have changed, please re-source setup files to use them.
```

The code in the folder builds two scenarios of drones. By using the command: “roslaunch two_drones_pkg two_drones.launch static:=True”. Running ROS, the static drone scene can be displayed in RVIZ.

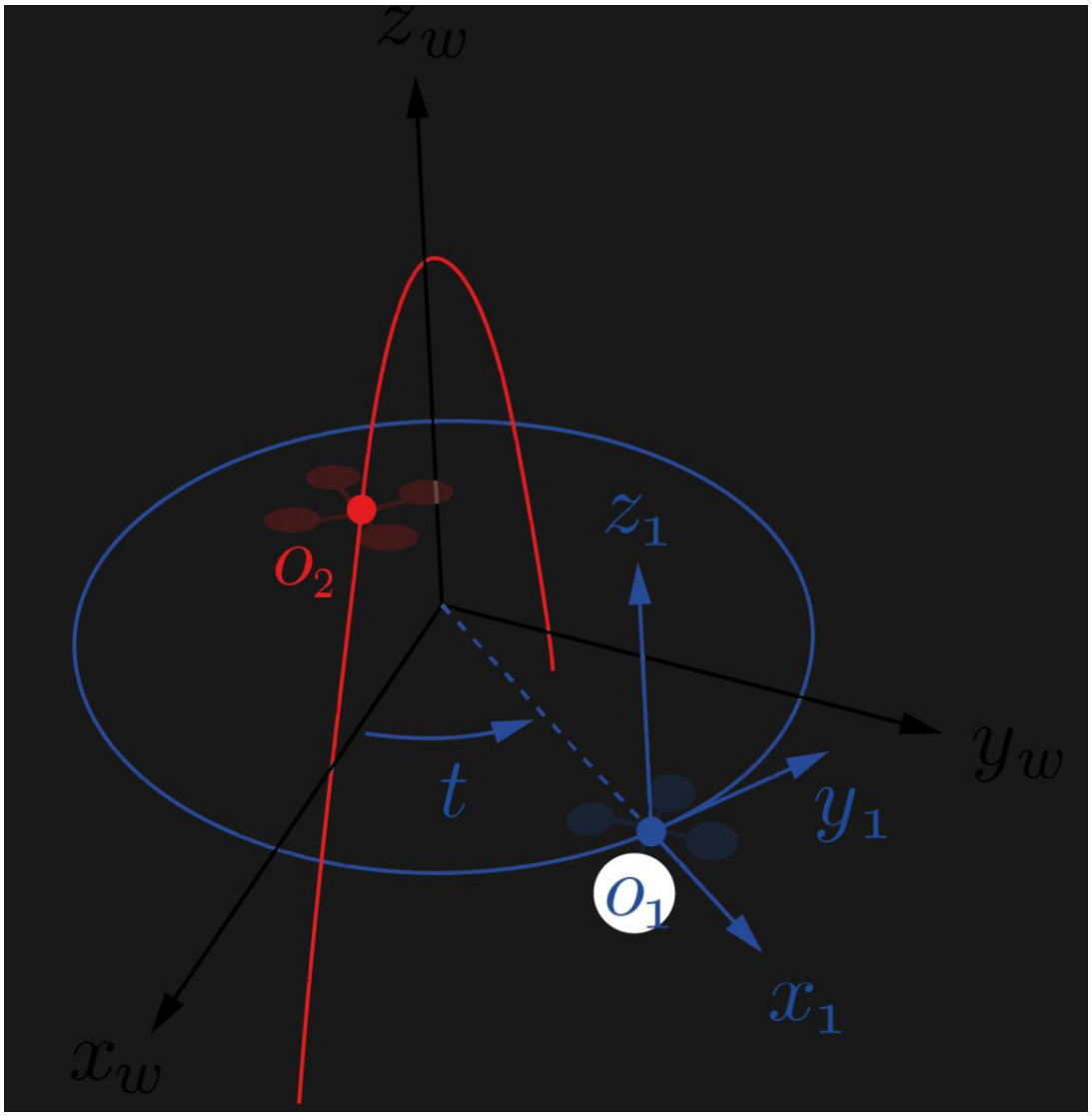


I can see that two drones are hovering in the space. To facilitate the display of the subsequent movements and data of the drones, I click on the "Add" button at the lower left corner and select TF. This will enable the display of the world frame, as well as the frames of the two drones respectively.



The next objective is to develop control nodes to enable the unmanned aircraft to move, allowing it to perform stationary hovering and simple trajectory movements.

II. Problem Description



This experiment established a three-layer coordinate system to describe the movement of the unmanned aerial vehicle. The world coordinate system (x_w, y_w, z_w) is a fixed global reference frame. The AV1 coordinate system(x_1, y_1, z_1) changes along with the movement of the unmanned aerial vehicle, and its y_1 axis always points in the direction of motion. The AV2 coordinate system (x_2, y_2, z_2) is relatively simple and only undergoes translational motion, and its direction is always consistent with the world coordinate system.

Two unmanned aircraft exhibit different movement patterns. AV1 moves along a circular trajectory, with the position function being $\mathbf{o}_1^w = [\cos(t), \sin(t), 0]^T$, performing uniform circular motion in the $x_w - y_w$ plane. The distinctive feature is that the y_1 axis of the aircraft's coordinate system always points in the direction of the motion tangent. The motion trajectory of AV2 is more complex. Its position function is $\mathbf{o}_2^w = [\sin(t), 0, \cos(2t)]^T$, forming a periodic curve in the $x_w - z_w$ plane. The body coordinate system of AV2 maintains a simple relationship, with each axis parallel to the world coordinate system and only experiencing positional changes.

The direction of AV1 is strictly constrained: the roll and pitch angles are zero, and the yaw angle $\psi = t$ changes linearly with time. This ensures that the drone remains level and that the forward axis precisely follows the movement direction. The corresponding rotation matrix is:

$$R_1^w = \begin{bmatrix} -\sin(t) & \cos(t) & 0 \\ -\cos(t) & -\sin(t) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The direction constraints of AV2 are relatively simple. The rotation matrix is the identity matrix $R_2^w = I$, indicating that the direction of its body coordinate system remains unchanged. This difference reflects the motion characteristics of different types of unmanned aerial vehicles.

The transformation between coordinate systems is achieved through homogeneous transformation matrices. The transformation matrix from the world coordinate system to the AV1 coordinate system is:

$$T_{world \to av1} = \begin{bmatrix} R_1^w & \mathbf{o}_1^w \\ \mathbf{0} & 1 \end{bmatrix}$$

The transformation from the world to AV2 is:

$$T_{world \to av2} = \begin{bmatrix} I & \mathbf{o}_2^w \\ \mathbf{0} & 1 \end{bmatrix}$$

The relative transformation from AV1 to AV2 is obtained through matrix operations:

$$T_{av1 \to av2} = T_{world \to av1}^{-1} \times T_{world \to av2}$$

This relationship provides a mathematical foundation for analyzing the relative motion between drones.

III. Basic ROS Commands

1. List the nodes that are operating in the static scene of the two unmanned aircraft.

frames_publisher_node: Responsible for publishing the coordinate system transformation information of the unmanned aerial vehicle

plots_publisher_node: Responsible for the publication of trajectory visualization data

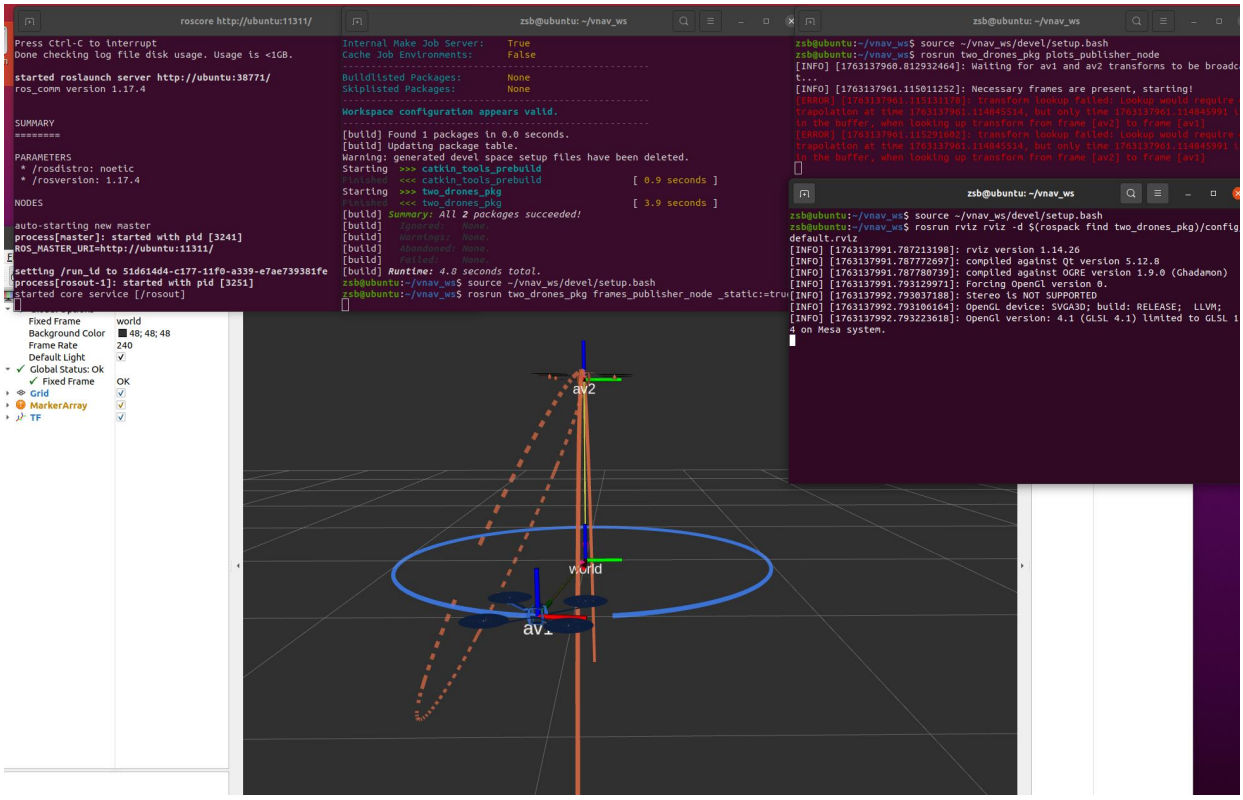
rviz: A ROS visualization tool used to display the model and trajectory of the unmanned aircraft.

rosout: ROS System Log Node

But before I had completed the code or set static to true：

static_transform_publisher、plots_publisher_node、rviz、rosout

2. How to run a static scene with two drones without using the roslaunch command?



Firstly, the ROS core service terminal is launched. This terminal can be opened in any directory location and can be executed by simply typing the "roscore" command. No additional environment configuration is required. This terminal serves as the management center of the ROS system and must always remain running to provide communication coordination services for other nodes.

The second terminal is used to start the coordinate system publishing node. This terminal also supports any directory location, but it is necessary to first execute the command "source ~/vnav_ws/devel/setup.bash" to set up the environment to ensure that the system can correctly identify the path of the function package. After completing the environment configuration, the static coordinate system publishing function can be started by the command "rosrun two_drones_pkg frames_publisher_node _static:=true", where the parameter "_static:=true" ensures that the drones remain at a fixed position.

The third terminal is responsible for the trajectory visualization function, and its directory selection is also unrestricted. After executing the necessary environment setting command "source ~/vnav_ws/devel/setup.bash", use the command "rosrun two_drones_pkg plots_publisher_node" to start the trajectory publishing node. This node will process and publish the motion trajectory data of the drones for use by the visualization tool.

The fourth terminal is used to start the RViz visualization interface, serving as the main window for users to observe the drone scene. In any directory, first configure the environment by executing the command "source ~/vnav_ws/devel/setup.bash", and then execute the command "rosrun rviz rviz -d $(rospack find two_drones_pkg)/config/default.rviz" to start RViz and automatically load the predefined configuration file. If there is a problem with the configuration file path, you can first run the command "rosrun rviz rviz" to start a blank RViz interface, and then manually load the corresponding configuration file through the graphical interface.

3. List the topics (subjects) for which each node publishes and subscribes. Which nodes are responsible for publishing av1, av2, and frames? Which topic is used to draw the drone grid in rViz?

In the ROS system of the static drone scenario, each node established a complete communication network through specific topics, achieving the effective transmission of coordinate system data and visualization information. The frames_publisher_node serves as the data source of

the entire system and assumes the core responsibility of coordinate system publishing. It simultaneously publishes coordinate transformation messages to the /tf and /tf_static topics. The /tf_static topic is specifically used to transmit static and unchanging coordinate system relationships, while the /tf topic handles dynamic transformation data. This node precisely maintains the relative transformation relationship between av1, av2 and the world coordinate system, providing an accurate spatial reference benchmark for other nodes.

The plots_publisher_node acts as a data processing and relay node. It receives coordinate system transformation information in real time by subscribing to the /tf topic, and calculates the corresponding motion trajectories based on these spatial data. After processing is completed, this node publishes the trajectories to the /trajectories topic, which specifically carries the trajectory marking data for visualization. This design separates data acquisition from data processing, ensuring the modularization and scalability of the system.

As the visualization terminal of the system, rviz acquires all the necessary information for scene rendering by simultaneously subscribing to the /tf and /trajectories topics. From the /tf topic, rviz obtains the transformation relationships between coordinate systems, thereby determining the position and orientation of the drone grid model in the three-dimensional space. From the /trajectories topic, it receives the pre-calculated trajectory path data and presents it in the visualization interface with specific colors and styles. This collaborative architecture enables each node to focus on its core function, and through the topic communication mechanism, it organically combines into a complete drone visualization system.

```
<launch>
  <arg name="static" default="false"/>

  <!-- Transform publishers !-->
  <group if="$(arg static)">
    <node pkg="tf2_ros" type="static_transform_publisher" name="av1broadcaster" args="1 0 0 0 0 0 1 world av1"/>
    <node pkg="tf2_ros" type="static_transform_publisher" name="av2broadcaster" args="0 0 1 0 0 0 1 world av2"/>
  </group>

  <node name="frames_publisher_node" pkg="two_drones_pkg" type="frames_publisher_node" unless="$(arg static)"/>

  <!-- Marker publisher -->
  <node name="plots_publisher_node" pkg="two_drones_pkg" type="plots_publisher_node"/>

  <!-- Visualizer -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find two_drones_pkg)/config/default.rviz"/>
</launch>
```

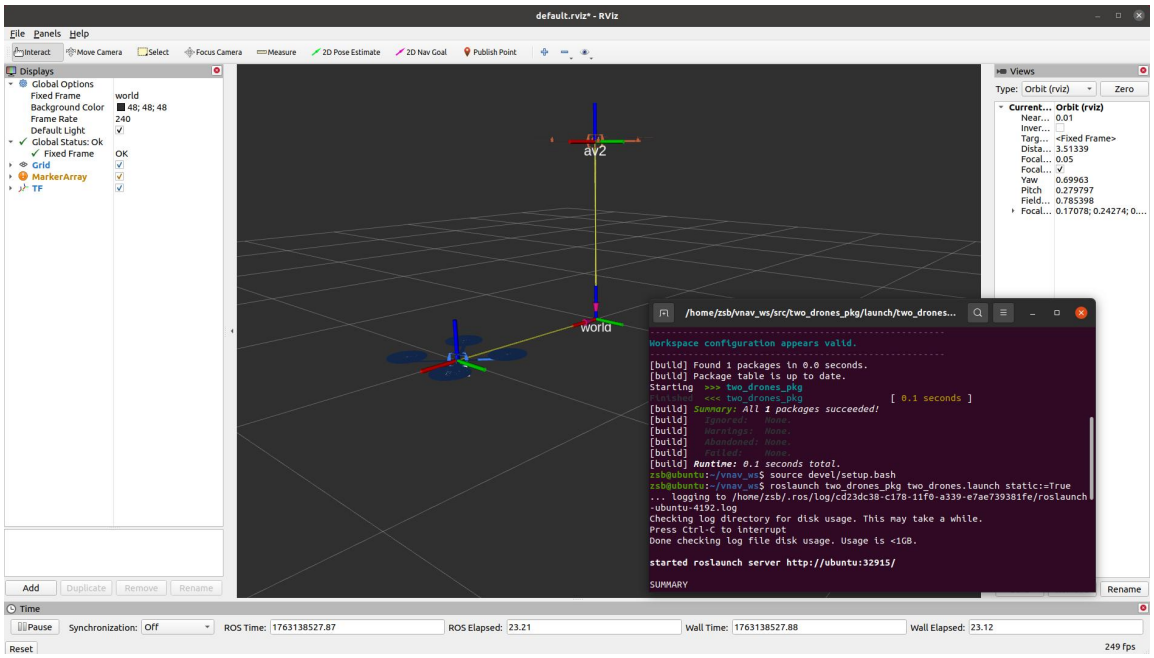
After reviewing the launch file, I realized that the above response was not entirely satisfactory. Before the launch, it will first be determined whether it is static. If so, the transform publisher used is a dedicated static one. Its pkg is "tf2_ros" and type is "static_transform_publisher". This means that when my code is not yet complete, the topic it sends is actually "static_transform_publisher". Later, plots_publisher_node will subscribe to the just-sent topic and send the topic for visualization, as well as rviz subscribing to the topic for visualization. However, after I complete all the code, I can make a complete judgment on whether the static is true or false, and then proceed with the subsequent topic sending and subscription.

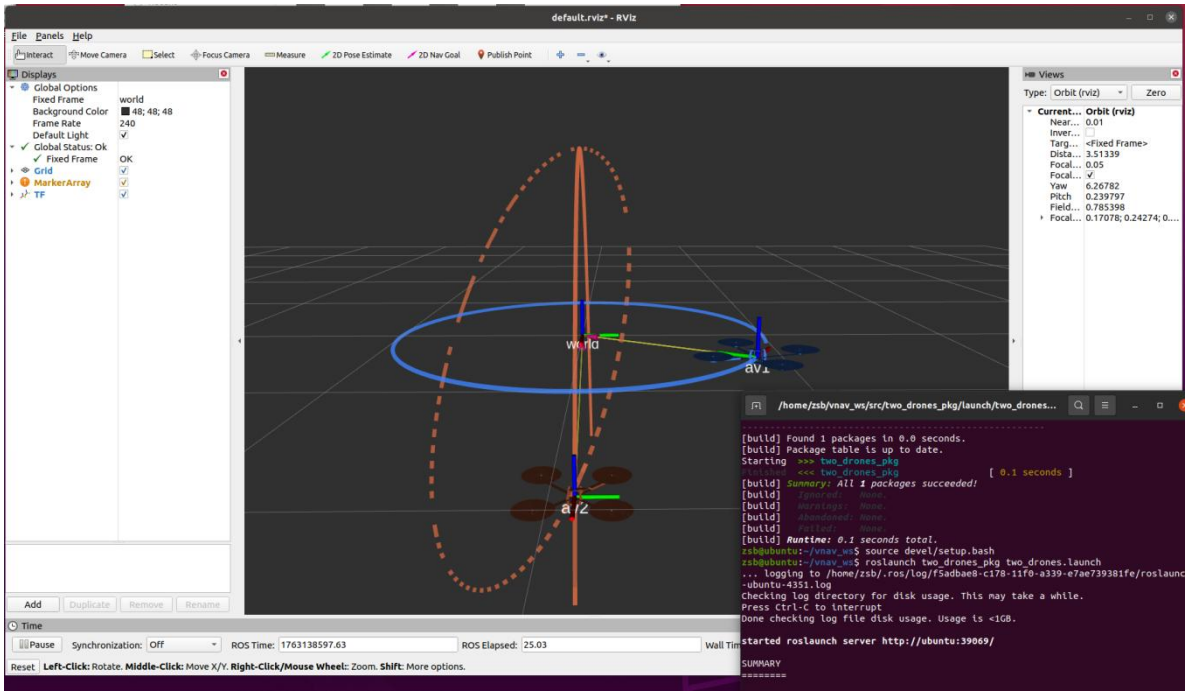
4. What changes would occur if we omitted the "static:=True" part? Why?

In the static mode, the position of the unmanned aerial vehicle is fixed. The system uses the /tf_static topic to publish one-time coordinate system data, with low resource consumption. It is suitable for system debugging and teaching demonstrations. The unmanned aerial vehicle maintains its initial position unchanged, and the trajectory is displayed as a short line or a single point.

In the dynamic mode, the unmanned aerial vehicles (UAVs) move continuously along the pre-determined trajectories: AV1 performs circular motion, while AV2 exhibits complex curvilinear motion. The system continuously updates the coordinate system data at a frequency of 50Hz through the /tf topic, which consumes relatively high resources, but it can fully display the motion process and trajectory shape, and is suitable for motion analysis and algorithm verification.

The two modes can be switched by setting the "static:=True" parameter, providing flexible options for different application scenarios.





IV. Using Transform to Control Movement

The code is as follows:

frames_publisher_node.cpp:

```
class FramesPublisherNode {
 private:
  ros::NodeHandle nh;
  ros::Time startup_time;

  ros::Timer heartbeat;
  // *** FILL IN *** instantiate a transform broadcaster...
  tf2_ros::TransformBroadcaster tf_broadcaster;
```

ros::NodeHandle nh: The core interface of the ROS node, responsible for communicating with the ROS master node (roscore), creating timers, publishers / subscribers.

tf2_ros::TransformBroadcaster: The transformation publishing tool of the tf2 library, used to publish the coordinate transformations (position + orientation) of the drone to the /tf topic.

```
public:
 FramesPublisherNode() {
   // NOTE: This method is run once, when the node is launched.
   startup_time = ros::Time::now();
   heartbeat =
       nh.createTimer(ros::Duration(0.02), &FramesPublisherNode::onPublish, this);
   heartbeat.start();
 }
```

Initialize the starting point of the movement time and set up a timer to ensure that the position and posture of the drone are updated every 0.02 seconds, guaranteeing the smoothness of the movement.

```
 void onPublish(const ros::TimerEvent&) {
   // NOTE: This method is called at 50Hz, due to the timer created on line 16.

   // 1. Compute time elapsed in seconds since the node has been started
   double time = (ros::Time::now() - startup_time).toSec();

   // Here we declare two geometry_msgs::TransformStamped objects, which need to be
   // populated
   geometry_msgs::TransformStamped AV1World;
   geometry_msgs::TransformStamped AV2World;

   // Set header information
   AV1World.header.stamp = ros::Time::now();
   AV1World.header.frame_id = "world";
   AV1World.child_frame_id = "av1";

   AV2World.header.stamp = ros::Time::now();
   AV2World.header.frame_id = "world";
   AV2World.child_frame_id = "av2";
```

Based on the start time, obtain the current duration of movement, which is used as the independent variable for trajectory calculation (ensuring that the movement changes continuously over time).

geometry_msgs::TransformStamped: A standard ROS message type used to store the transformation ("parent coordinate system → child coordinate system") between two coordinate systems (including position translation and orientation rotation).

The role of header information: It enables the receiving party (such as RViz, other nodes) to know the time and reference coordinate system of the transformation, ensuring the consistency of the data.

```
   // 2. Populate the two transforms for the AVs

   // AV1: position [cos(time), sin(time), 0.0]
   AV1World.transform.translation.x = cos(time);
   AV1World.transform.translation.y = sin(time);
   AV1World.transform.translation.z = 0.0;

   // AV1: orientation - y axis tangent to trajectory, z parallel to world
   // The tangent direction is [-sin(time), cos(time), 0]
   // We want y axis to point in this direction
   double yaw = atan2(cos(time), -sin(time)); // Calculate yaw angle
   tf2::Quaternion q_av1;
   q_av1.setRPY(0.0, 0.0, yaw); // Only yaw rotation
   AV1World.transform.rotation.x = q_av1.x();
   AV1World.transform.rotation.y = q_av1.y();
   AV1World.transform.rotation.z = q_av1.z();
   AV1World.transform.rotation.w = q_av1.w();

   // AV2: position [sin(time), 0.0, cos(2*time)]
   AV2World.transform.translation.x = sin(time);
   AV2World.transform.translation.y = 0.0;
   AV2World.transform.translation.z = cos(2 * time);

   // AV2: orientation is irrelevant, set to identity
   AV2World.transform.rotation.x = 0.0;
   AV2World.transform.rotation.y = 0.0;
   AV2World.transform.rotation.z = 0.0;
   AV2World.transform.rotation.w = 1.0;

   // 3. Publish the transforms using a tf2_ros::TransformBroadcaster
   std::vector<geometry_msgs::TransformStamped> transforms;
   transforms.push_back(AV1World);
   transforms.push_back(AV2World);
   tf_broadcaster.sendTransform(transforms);
 }
```

Trajectory logic: x = cos(time), y = sin(time) are the standard parametric equations for a unit circle. As time (time) changes, AV1 will perform uniform circular motion along the x-y plane (angular velocity 1 rad/s).

Attitude logic: The yaw angle of the tangent direction is calculated using atan2(cos(time), -sin(time)), ensuring that the y-axis (movement direction) of AV1 is always consistent with the tangent of the trajectory, meeting the requirement of "movement direction following the trajectory". Trajectory logic: x = sin(time), z = cos(2time) form a periodic spatial curve (x-axis period 2π, z-axis period π). AV2 will perform "wave-like" curve motion in the x-z plane.

Attitude logic: The quaternion is (0, 0, 0, 1) (unit quaternion), indicating that the body coordinate system of AV2 (x2, y2, z2) is completely parallel to the world coordinate system (xw, yw, zw), and only undergoes translational motion. The coordinates of the two drones are sent to the /tf topic in one go using tf_broadcaster.sendTransform.

Other nodes (such as RViz, trajectory drawing node) subscribe to the /tf topic, and then can obtain the real-time position and attitude of the drones, enabling visualization or subsequent calculations.

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "frames_publisher_node");
  FramesPublisherNode node;
```

Initialize the ROS node, create the FramesPublisherNode object (automatically start the motion control), and let the node run continuously by using ros::spin(), responding to the timer callback.

plots_publisher_nodes.cpp:

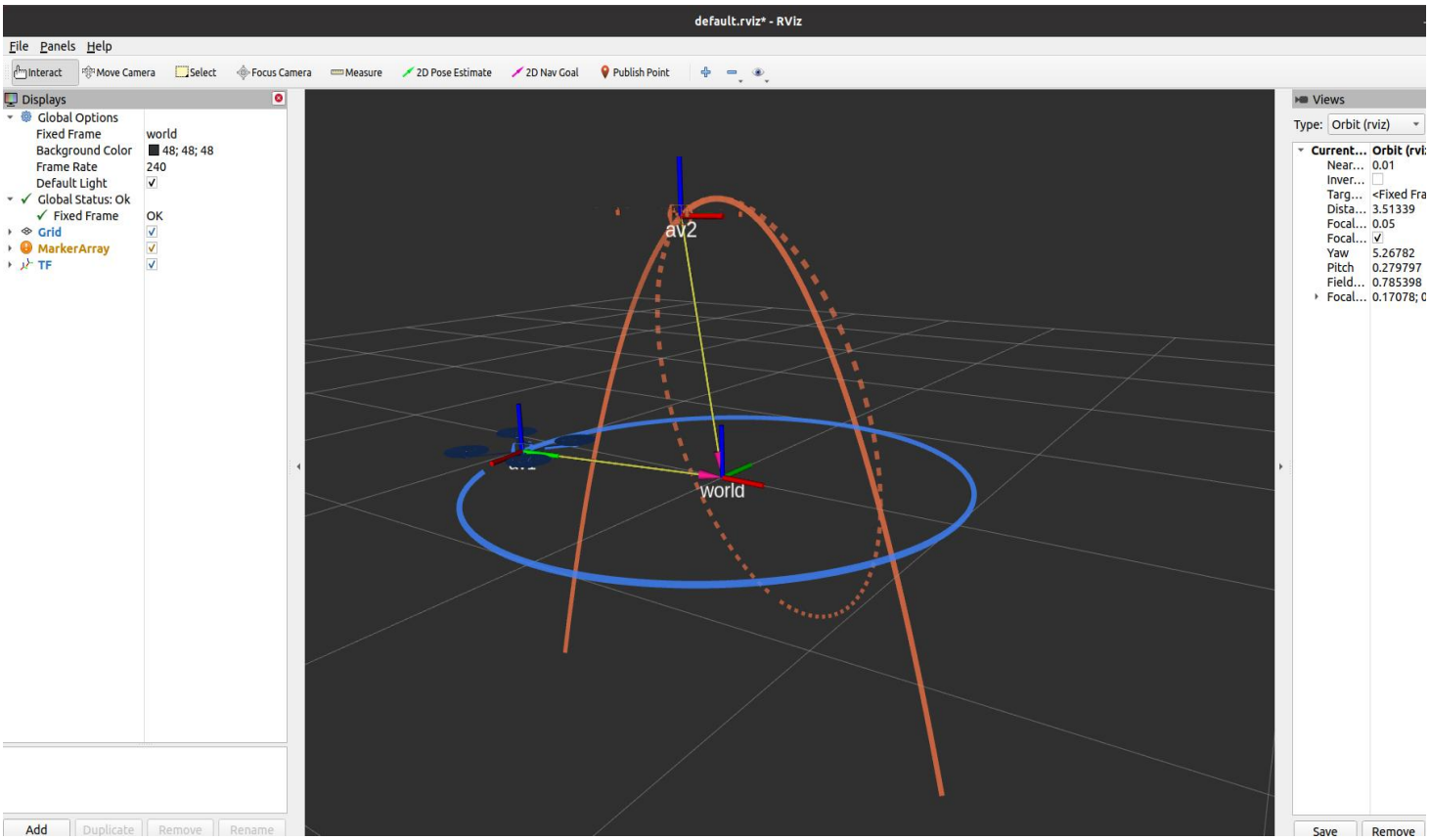
```
void update() {
  // NOTE: you need to populate this transform
  geometry_msgs::TransformStamped transform;
  try {
    // ~~~~~~~~~~~~~~~~~~~~~~  BEGIN OF EDIT SECTION  ~~~~~~~~~~~~~~~~~~~~~~~~

    /* The transform object needs to be populated with the most recent
     * transform from ref_frame to dest_frame as provided by tf.

     * Relevant variables in this scope:
     *   - ref_frame, the frame of reference relative to which the trajectory
     *                needs to be plotted (given)
     *   - dest_frame, the frame of reference of the object whose trajectory
     *                 needs to be plotted (given)
     *   - parent->tf_buffer, a tf2_ros::Buffer object (given)
     *   - transform, the geometry_msgs::TransformStamped object that needs to be populated
     *
     * HINT: use "lookupTransform", see
     * https://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20listener%20%28C%2B%2B%29#TheCode
     */

    // *** FILL IN ***
     transform = parent->tf_buffer.lookupTransform(ref_frame, dest_frame, ros::Time(0) );
    // ~~~~~~~~~~~~~~~~~~~~~~  END OF EDIT SECTION  ~~~~~~~~~~~~~~~~~~~~~~~~
    while (poses.size() >= buffer_size) {
      poses.pop_front();
    }

    geometry_msgs::Point tmp;
    tmp.x = transform.transform.translation.x;
    tmp.y = transform.transform.translation.y;
    tmp.z = transform.transform.translation.z;
    poses.push_back(tmp);
  } catch (const tf2::TransformException& ex) {
    ROS_ERROR_STREAM("transform lookup failed: " << ex.what());
  }
}
```

The update function serves as a crucial bridge connecting "coordinate transformation publishing" and "visualization trajectory generation". Its main task is to accurately extract the real-time position data of the drone from the tf2 cache of ROS, form a continuous trajectory through orderly cache management, and provide a sequence of points that can be plotted for visualization tools such as RViz, ultimately achieving the intuitive presentation of the drone's movement trajectory.

The most core operation is to query the coordinate transformation through the line of code "transform = parent->tf_buffer.lookupTransform(ref_frame, dest_frame, ros::Time(0))". This line of code implements the core function of extracting target data from the cache. The three parameters each have their own significance: ref_frame and dest_frame jointly define the transformation relationship "from which coordinate system to which coordinate system", for example, when ref_frame is "world" and dest_frame is "av1", the query is for the position and posture of AV1 relative to the world coordinate system; while ros::Time(0) is the key design of the timestamp parameter, it indicates "obtaining the latest available transformation data in the cache", rather than specifying a fixed time point - this choice is mainly to avoid data delay issues, because there is a microsecond-level delay from the publication of the coordinate transformation to its storage in the cache. If using ros::Time::now() (current time), it may not be able to query the latest data that has not been cached, while ros::Time(0) will automatically match the most recent data with the closest timestamp in the cache, significantly improving the query success rate. After the query is successful, the returned geometry_msgs::TransformStamped object contains the core information required for trajectory drawing - transform.translation (position x, y, z), and the subsequent trajectory generation is based on this data.

After modifying the codes of the two parts and running them automatically using roslaunch as before, you can see the complete running trajectory image.

I found that there is inevitably an error occurring during the operation.This error is a time synchronization issue that occurs during the TF transformation query in the ROS system. The root cause lies in the plots_publisher_node node, which, when querying the coordinate system transformation from av2 to av1, has a slight time discrepancy between the requested timestamp and the actual timestamp of the TF data being published. The ROS TF2 library is meticulously



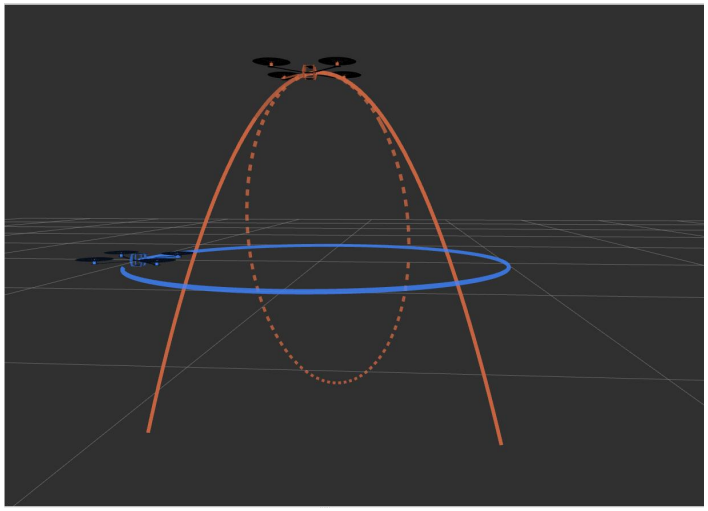
designed and strictly prohibits time extrapolation of transformation data. When the query time point has a very small difference from the latest available data time point, even if this difference is only at the microsecond level, the system will throw an extrapolation error instead of automatically interpolating.

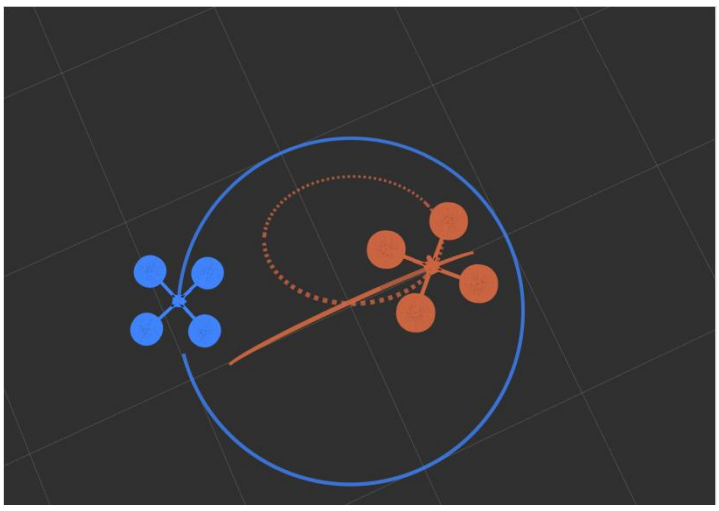
The main measure to solve this problem is to modify the TF query logic in the TrajTrail::update() method of plots_publisher_node.cpp. It is recommended to adopt a query method with a timeout mechanism. Replace the original call of lookupTransform(ref_frame, dest_frame, ros::Time(0)) with lookupTransform(ref_frame, dest_frame, ros::Time(0), ros::Duration(0.1)). This modification instructs the TF system: if there is no precise matching transformation data at the requested time, it can wait for a maximum of 0.1 seconds to obtain available data, thereby bypassing the small time difference caused by node scheduling during the initial startup.

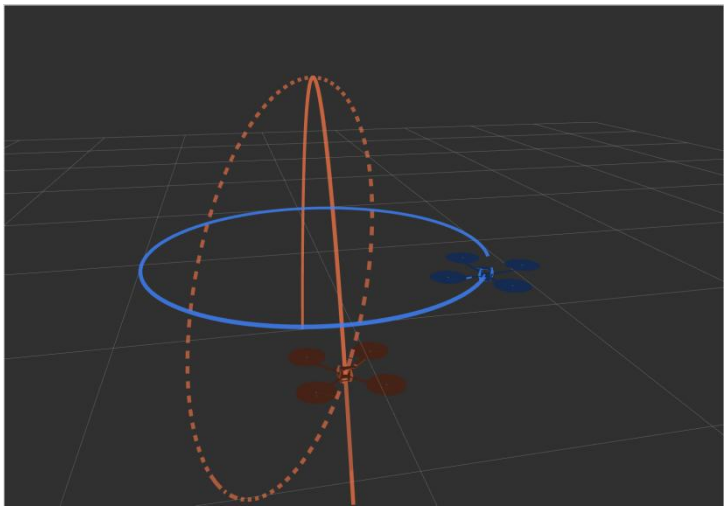
PART 3. Control the Trajectory Movements of the Two Unmanned Aircraft as Per the Requirements

I.Launch a Campaign

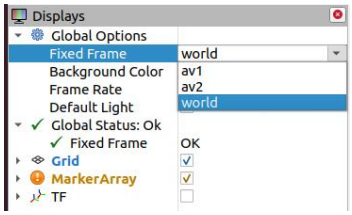
This part requires me to fill in the code for the frame_publisher_node.cpp and plots_publisher_nodes.cpp sections. I have completed this part in PART 2 IV. Using Transform to Control Movement.The obtained RViz image is as follows:



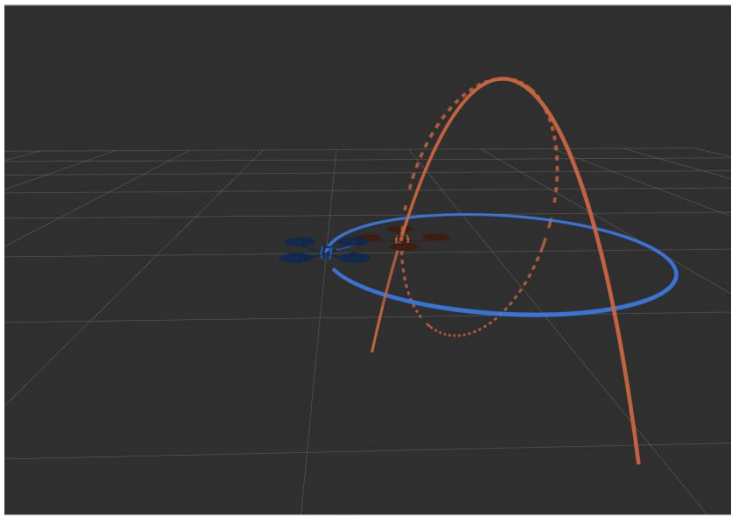


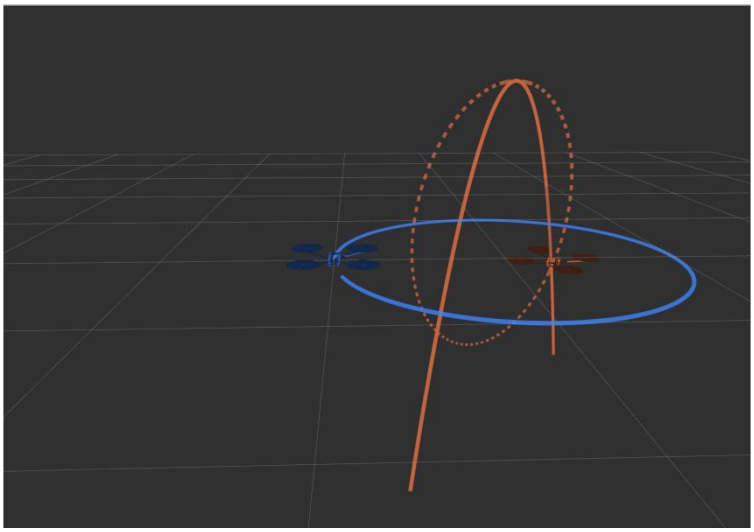


The image in the figure is based on the world frame as the fixed coordinate system. Then change the "Fixed Frame" option of the sidebar to "av1", that is, take the "av1 frame" as the fixed coordinate system. At this point, observe the movement image of "av2" relative to "av1".

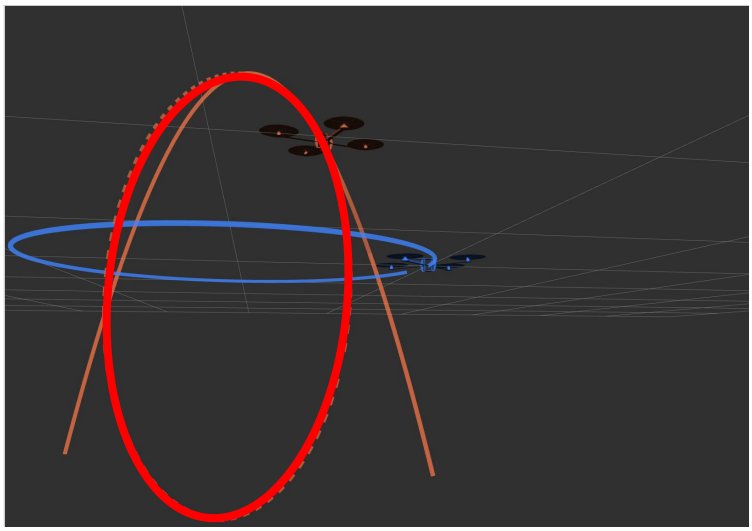


I obtained the following image:



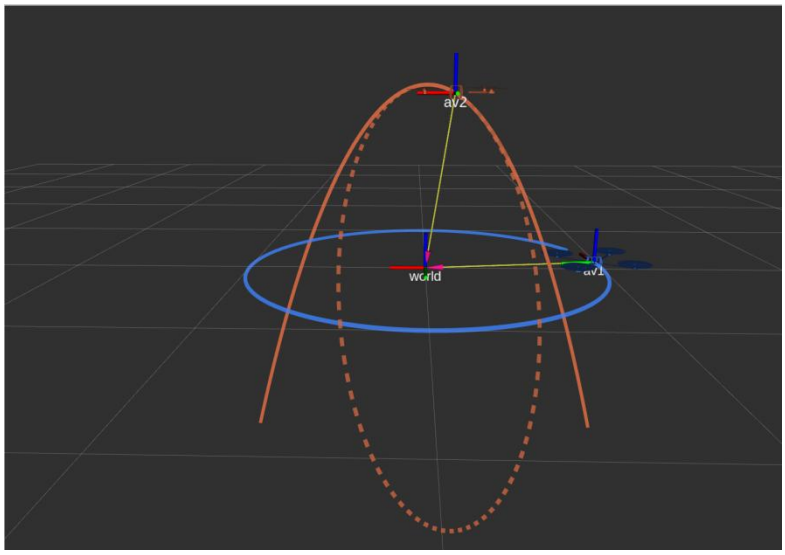


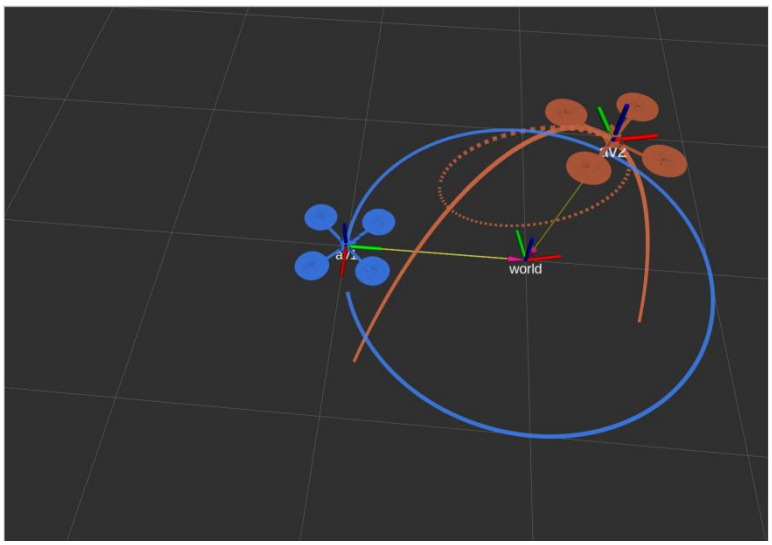
I found that the parabolic trajectory of av2 is rotating around a certain point on the circular trajectory of av1. In fact, the trajectory of av2 is an ellipse, as shown in the figure below:



II.Looking Up a Transform

The experiment requires me to display the trajectory route. The trajectory route has already been achieved in the previous experimental content. So next, I added the TF coordinate system to RViz, which enables a more intuitive display of the changes in the coordinate axes.





When the fixed coordinate system is set to the world coordinate system, the observer is in an absolute God's-eye view. In this mode, the blue solid line clearly delineates the absolute motion trajectory of AV1 in the world coordinate system, which is a perfect circle strictly confined within the horizontal plane, intuitively reflecting the mathematical laws described by its position function. At the same time, the red solid line shows the absolute motion path of AV2 in the world coordinate system, presenting a unique spatial curve feature, and its movement is constrained within the vertical plane. These two trajectories respectively present the motion characteristics of the unmanned aircraft in the fixed reference system from different dimensions, providing a direct visual basis for analyzing the absolute motion.

After switching the fixed coordinate system to the AV1 aircraft coordinate system, the entire observation perspective underwent a fundamental transformation. At this point, what was presented was the relative motion scene as seen from the AV1 pilot's perspective. The most significant change was that the world coordinate system began to rotate and move around AV1. This visual phenomenon was not that the world was moving, but rather it reflected the movement state of the observation platform itself. This phenomenon strongly verified the correctness of the coordinate system transformation. From this perspective, the AV2 relative motion trajectory depicted by the red dotted line exhibited complex spatial characteristics. This trajectory was neither a simple circle nor a regular parabola; rather, it was the composite result formed by the absolute motion of the two drones after the coordinate system transformation.

PART 4. Mathematical Derivations

"Let's do some calculations."

So far, I have used ROS and tf to gain an intuitive understanding of the movement of AV2 relative to the body coordinate system of AV1. In this section, I need to apply my knowledge of homogeneous transformations to clearly study its relative trajectory. Since this exercise is designed to help you become familiar with the mathematical principles of three-dimensional transformations, I need to write out in detail all the homogeneous transformation matrices used throughout the process and accurately explain the logic and algebraic steps employed.

1. In the problem formulation ,we mentioned that AV2's trajectory is an arc of parabola in the $x-z$ plane of the world frame. Can you prove this statement?

$$\because\ O_2^w(t)=[x(t),\ y(t),\ z(t)]^T=[\sin t,\ 0,\ \cos(2t)]^T$$

$$\cos(2t)=1-2\sin^2 t$$

$$\therefore\ z(t)=1-2(x(t))^2\quad \therefore\ z=-2x^2+1$$

$$\therefore\ \text{This is the equation of a parabola in the } x\text{-}z \text{ plane.}$$

2. Compute $o_2^1(t)$, i.e., the position of AV2 relative to AV1's body frame as a function of t .

$$\because\ O_1^w(t)=\begin{bmatrix}\cos t\\ \sin t\\ 0\end{bmatrix}\quad O_2^w(t)=\begin{bmatrix}\sin t\\ 0\\ \cos(2t)\end{bmatrix}\quad R_w^1(t)=\begin{bmatrix}\cos t & -\sin t & 0\\ \sin t & \cos t & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$\therefore\ T_w^1(t)=\begin{bmatrix}R_w^1(t) & O_1^w(t)\\ 0^T & 1\end{bmatrix}\quad T_w^2(t)=\begin{bmatrix}I & O_2^w(t)\\ 0^T & 1\end{bmatrix}$$

$$\therefore\ T_1^2(t)=(T_w^1(t))^{-1}\cdot T_w^2(t)=\begin{bmatrix}R_1^2(t) & O_2^1(t)\\ 0^T & 1\end{bmatrix}$$

$$\therefore\ O_2^1(T)=R_w^1(t)^T\cdot(O_2^w-O_1^w)=\begin{bmatrix}\cos t & \sin t & 0\\ -\sin t & \cos t & 0\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}\sin t-\cos t\\ -\sin t\\ \cos 2t\end{bmatrix}=\begin{bmatrix}\frac{1}{2}\sin 2t-1\\ -\sin^2 t\\ \cos 2t\end{bmatrix}$$

3. Show that $o_2^1(t)$ describes a planar curve and find the equation of its plane Π.

Let $x=\frac{1}{2}\sin 2t-1$ $y=-\sin^2 t$. $z=\cos 2t$.

$\because y=-\sin^2 t=\frac{\cos 2t-1}{2}=\frac{z-1}{2}$ $\therefore z=2y+1$

$\because x=\frac{1}{2}\sin 2t-1$ $\therefore \sin 2t=2(x+1)$

$\because \sin^2 2t+\cos^2 2t=1$ $\therefore [2(x+1)]^2+z^2=1$

$\therefore 4(x+1)^2+(2y+1)^2=1$ This is a equation of an ellipse

And. $\because z=2y+1$. There is no t and x.

$\therefore$ It's a linerequation and it's parallel to the x-axis

4. Rewrite the above trajectory explicitly using a 2D frame of reference (x_p, y_p) on the plane found before. Try to ensure that the curve is centered at the origin of this 2D frame and that x_p , y_p are axes of symmetry for the curve.

$\because \Pi=2y-z+1=0$. $P'=(-1,-\frac{1}{2},0)$. $\therefore P'$ is in Π

$\because \vec{n}=(0,2,-1)$ $\therefore \hat{z_p}=\frac{(0,2,-1)}{\sqrt{0^2+2^2+(-1)^2}}=(0,\frac{2}{\sqrt{5}},-\frac{1}{\sqrt{5}})$.

$\because (1,0,0)\cdot(0,2,-1)=0$. $\therefore (1,0,0)$ is perpendicular to $\vec{n}$

$\therefore \hat{x_p}=(1,0,0)$ $\therefore \hat{y_p}=\hat{z_p}\times\hat{x_p}=(0,-\frac{1}{\sqrt{5}},-\frac{2}{\sqrt{5}})$

$\therefore o_2^p(t)=\begin{bmatrix}(O_2^1(t)-P')\cdot\hat{x_p}\\(O_2^1(t)-P')\cdot\hat{y_p}\\(O_2^1(t)-P')\cdot\hat{z_p}\end{bmatrix}=\begin{bmatrix}\frac{1}{2}\sin 2t\\-\frac{\sqrt{5}}{2}\cos 2t\\0\end{bmatrix}$ $\therefore (x_p,y_p)=(\frac{1}{2}\sin 2t,-\frac{\sqrt{5}}{2}\cos 2t)$

5. Using the expression of $o_2^p(t)$, prove that the trajectory of AV2 relative to AV1 is an ellipse and compute the lengths of its semi-axes.

$\because x_p=\frac{1}{2}\sin 2t$. $y_p=-\frac{\sqrt{5}}{2}\cos 2t$ $\therefore (2x_p)^2+(-\frac{2}{\sqrt{5}}y_p)^2=1$ $\therefore \frac{x_p^2}{(\frac{1}{2})^2}+\frac{y_p^2}{(\frac{\sqrt{5}}{2})^2}=1$

So the major semi-axis is $\frac{\sqrt{5}}{2}$. the minor semi-axis is $\frac{1}{2}$

(y) (x).

PART 5. More Properies of Quaternions

In the lecture notes, we have defined two linear maps $\Omega_1: \mathbb{R}^4 \to \mathbb{R}^{4\times4}$ and $\Omega_2: \mathbb{R}^4 \to \mathbb{R}^{4\times4}$, such that for any $\mathbf{q} \in \mathbb{R}^4$, we have:

$$\boldsymbol{\Omega_1(q)} = \begin{bmatrix} \mathbf{q_4} & \mathbf{-q_3} & \mathbf{q_2} & \mathbf{q_1} \\ \mathbf{q_3} & \mathbf{q_4} & \mathbf{-q_1} & \mathbf{q_2} \\ \mathbf{-q_2} & \mathbf{q_1} & \mathbf{q_4} & \mathbf{q_3} \\ \mathbf{-q_1} & \mathbf{-q_2} & \mathbf{-q_3} & \mathbf{q_4} \end{bmatrix}, \quad \boldsymbol{\Omega_2(q)} = \begin{bmatrix} \mathbf{q_4} & \mathbf{q_3} & \mathbf{-q_2} & \mathbf{q_1} \\ \mathbf{-q_3} & \mathbf{q_4} & \mathbf{q_1} & \mathbf{q_2} \\ \mathbf{q_2} & \mathbf{-q_1} & \mathbf{q_4} & \mathbf{q_3} \\ \mathbf{-q_1} & \mathbf{-q_2} & \mathbf{-q_3} & \mathbf{q_4} \end{bmatrix}.$$

The product between any two unit quaternions can then be explicitly computed as: $\mathbf{q_a} \otimes \mathbf{q_b} = \boldsymbol{\Omega_1(q_a)q_b} = \boldsymbol{\Omega_2(q_b)q_a}$

In fact, the two linear maps $\boldsymbol{\Omega_1}$ and $\boldsymbol{\Omega_2}$ have more interesting properties, and you are asked to prove the following equalities:

1.For any unit quaternion q , both $\Omega_1(q)$ and $\Omega_2(q)$ are *orthogonal* matrices, *i. e.*,

$$\Omega_1(q)^T\Omega_1(q) = \Omega_1(q)\Omega_1(q)^T = I_4,$$
$$\Omega_2(q)^T\Omega_2(q) = \Omega_2(q)\Omega_2(q)^T = I_4.$$

Intuitively, what is the reason that both $\Omega_1(q)$ and $\Omega_2(q)$ must be orthogonal?

$$\Omega_1(q)^T\Omega_1(q) = \begin{bmatrix} q_4 & q_3 & -q_2 & -q_1 \\ -q_3 & q_4 & q_1 & -q_2 \\ q_2 & -q_1 & q_4 & -q_3 \\ q_1 & q_2 & q_3 & q_4 \end{bmatrix} \cdot \begin{bmatrix} q_4 & -q_3 & q_2 & q_1 \\ q_3 & q_4 & -q_1 & q_2 \\ -q_2 & q_1 & q_4 & q_3 \\ -q_1 & -q_2 & -q_3 & q_4 \end{bmatrix}$$

$(\Omega_1(q)^T\Omega_1(q))_{11} = q_4^2 + q_3^2 + q_2^2 + q_1^2 = 1$ Similarly: $(\Omega_1(q)^T\Omega_1(q))_{22} = (\Omega_1(q)^T\Omega_1(q))_{33} = (\Omega_1(q)^T\Omega_1(q))_{44} = 1$

$(\Omega_1(q)^T\Omega_1(q))_{12} = -q_4q_3 + q_4q_3 - q_2q_1 + q_2q_1 = 0$ Similarly: $(\Omega_1(q)^T\Omega_1(q))_{13} = \cdots = (\Omega_1(q)^T\Omega_1(q))_{43} = 0$

$\therefore \Omega_1(q)^T\Omega_1(q) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I_4$. From the commutative law. we have $\Omega_1(q)^T\Omega_1(q) = \Omega_1(q)\Omega_1(q)^T = I_4$

$$\Omega_2(q)^T\Omega_2(q) = \begin{bmatrix} q_4 & -q_3 & q_2 & -q_1 \\ q_3 & q_4 & -q_1 & -q_2 \\ -q_2 & q_1 & q_4 & -q_3 \\ q_1 & q_2 & q_3 & q_4 \end{bmatrix} \cdot \begin{bmatrix} q_4 & q_3 & -q_2 & q_1 \\ -q_3 & q_4 & q_1 & q_2 \\ q_2 & -q_1 & q_4 & q_3 \\ -q_1 & -q_2 & -q_3 & q_4 \end{bmatrix}$$

$(\Omega_2(q)^T\Omega_2(q))_{11} = q_4^2 + q_3^2 + q_2^2 + q_1^2 = 1$. Similarly: $(\Omega_2(q)^T\Omega_2(q))_{22} = (\Omega_2(q)^T\Omega_2(q))_{33} = (\Omega_2(q)^T\Omega_2(q))_{44} = 1$

$(\Omega_2(q)^T\Omega_2(q))_{12} = q_4q_3 - q_4q_3 + q_1q_2 - q_1q_2 = 0$. Similarly: $(\Omega_2(q)^T\Omega_2(q))_{12} = \cdots = (\Omega_2(q)^T\Omega_2(q))_{43} = 0$

$\therefore \Omega_2(q)^T\Omega_2(q) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I_4$. From the commutative law, we have $\Omega_2(q)^T\Omega_2(q) = \Omega_2(q)\Omega_2(q)^T = I_4$

Reason: When a unit quaternion is otimes with p.

$\|q \otimes p\| = \|q\| \cdot \|p\| = \|p\|$, $\|p \otimes q\| = \|p\| \cdot \|q\| = \|p\|$

$\therefore \|q\| = 1$. When: $q \otimes p$: $\Omega_1(q) \cdot p = q \otimes p$. When $p \otimes q$, $\Omega_2(q) \cdot p = p \otimes q$.

$\therefore \Omega_1(q)$ and $\Omega_2(q)$ must be orthogonal.

2. For any unit quaternion q, both $\Omega_1(q)$ and $\Omega_2(q)$ convert q to be the unit quaternion that corresponds to the 3D identity rotation, *i. e.*

$$\Omega_1(q)^T q = \Omega_2(q)^T q = [0,0,0,1]^T$$

$$\Omega_1(q)^T \cdot q = \begin{bmatrix} q_4 & q_3 & -q_2 & -q_1 \\ -q_3 & q_4 & q_1 & -q_2 \\ q_2 & -q_1 & q_4 & -q_3 \\ q_1 & q_2 & q_3 & q_4 \end{bmatrix} \cdot \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = \begin{bmatrix} q_1q_4 + q_2q_3 - q_2q_3 - q_1q_4 \\ -q_1q_3 + q_2q_4 + q_1q_3 - q_2q_4 \\ q_1q_2 - q_1q_2 + q_3q_4 - q_3q_4 \\ q_1^2 + q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \therefore \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\Omega_2(q)^T \cdot q = \begin{bmatrix} q_4 & -q_3 & q_2 & -q_1 \\ q_3 & q_4 & -q_1 & -q_2 \\ -q_2 & q_1 & q_4 & -q_3 \\ q_1 & q_2 & q_3 & q_4 \end{bmatrix} \cdot \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = \begin{bmatrix} q_1q_4 - q_2q_3 + q_2q_3 - q_1q_4 \\ q_1q_3 + q_2q_4 - q_1q_3 - q_2q_4 \\ -q_1q_2 + q_1q_2 + q_3q_4 - q_3q_4 \\ q_1^2 + q_2^2 + q_3^2 + q_4^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

3. For any two vectors $x, y \in \mathbb{R}^4$ **, show the two linear operators commute,** *i. e.*

$$\Omega_1(x)\Omega_2(y) = \Omega_2(y)\Omega_1(x)$$

$$\Omega_1(x)\Omega_2(y)^T = \Omega_2(y)^T\Omega_1(x).$$

$\because$ For $\forall p$: $\Omega_1(x)p = x \otimes p$, $\Omega_2(y)p = p \otimes y$

$\therefore \Omega_1(x)\Omega_2(y)\cdot p = \Omega_1(x)\cdot(p \otimes y) = x \otimes (p \otimes y)$

$\Omega_2(y)\Omega_1(x)p = \Omega_2(y)\cdot(x \otimes p) = (x \otimes p) \otimes y$

$\because x \otimes (p \otimes y) = (x \otimes p) \otimes y \quad \therefore \Omega_1(x)\Omega_2(y)\cdot p = \Omega_2(y)\Omega_1(x)\cdot p$

$\therefore \Omega_1(x)\Omega_2(y) = \Omega_2(y)\Omega_1(x)$

$\because \Omega_1(x)\Omega_2(y)^T \cdot \Omega_2(y) = \Omega_1(x)\cdot I_4 = I_4 \cdot \Omega_1(x)$

$= \Omega_2(y)^T \cdot \Omega_2(y)\cdot\Omega_1(x) = \Omega_2(y)^T \cdot \Omega_1(x)\cdot\Omega_2(y)$

$\therefore \Omega_1(x)\Omega_2(y)^T = \Omega_2(y)^T \cdot \Omega_1(x)$