Multimodal visual understand + PTZ tracking(Text Version)

1. Course Content

1. Learn to use robot visual understanding combined with gimbal tracking.
2. Analyze newly discovered key source code.

2. Preparation

2.1 Content Description

> This course uses the Jetson Orin NANO as an example. For Raspberry Pi and Jetson Nano boards, you need to open a terminal on the computer and enter the command to enter the Docker container. Once inside the Docker container, enter the commands mentioned in this course in the terminal. For instructions on entering the Docker container from the computer, refer to **[01. Robot Configuration and Operation Guide] -- [5.Enter Docker (For JETSON Nano and RPi 5)]**. For Orin boards, simply open a terminal and enter the commands mentioned in this course.

📝 This example uses `model: "qwen/qwen2.5-vl-72b-instruct:free","qwen-vl-latest"`

⚠ The responses from the large model for the same test command may not be exactly the same and may differ slightly from the screenshots in the tutorial. If you need to increase or decrease the diversity of the large model's responses, refer to the section on configuring the decision-layer large model parameters in the **[03.AI Model Basics] -- [5.Configure AI large model]**.

⚡ It's recommended to try the previous visual example first. This example adds voice functionality to the singleton example. The functionality is largely the same, so I won't go into detail about the implementation, code debugging, and results!

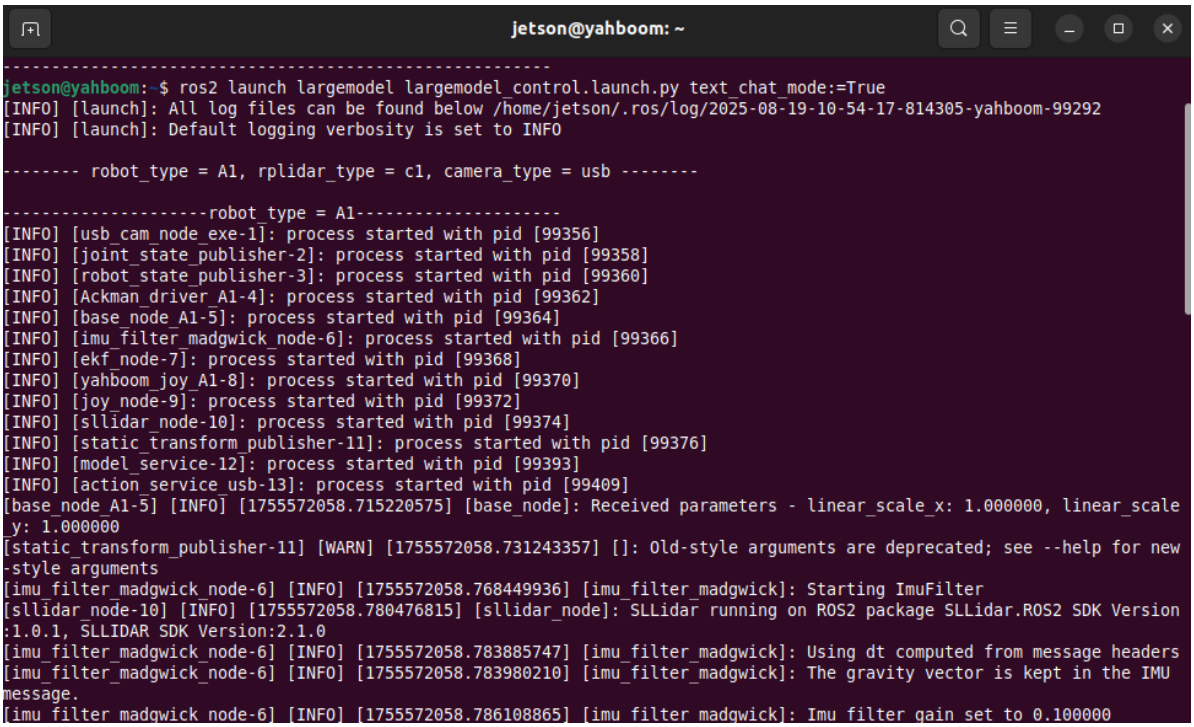
3. Running the Example

3.1 Starting the Program

For Raspberry Pi 5 and Jetson Nano controllers, you need to first enter the Docker container. This is not necessary for the Orin board.

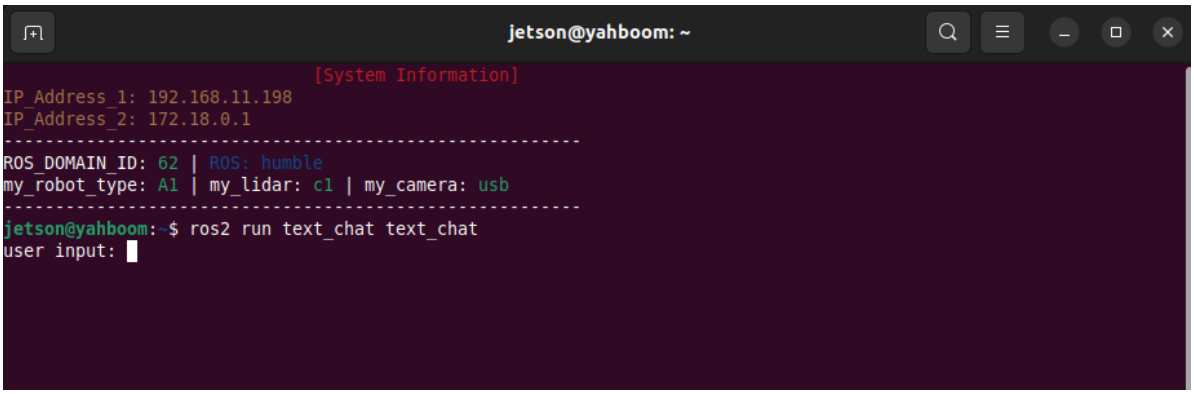
Open a terminal on the car and enter the command:

```
ros2 launch largemodel largemodel_control.launch.py text_chat_mode:=True
```



Open another terminal and start it.

```
ros2 run text_chat text_chat
```



3.2 Test Cases

Here are some reference test cases; users can create their own dialogue commands.

- Start xx tracking

  Color/Face/Object/Machine Code/QR Code/Gesture Recognition/Human Posture

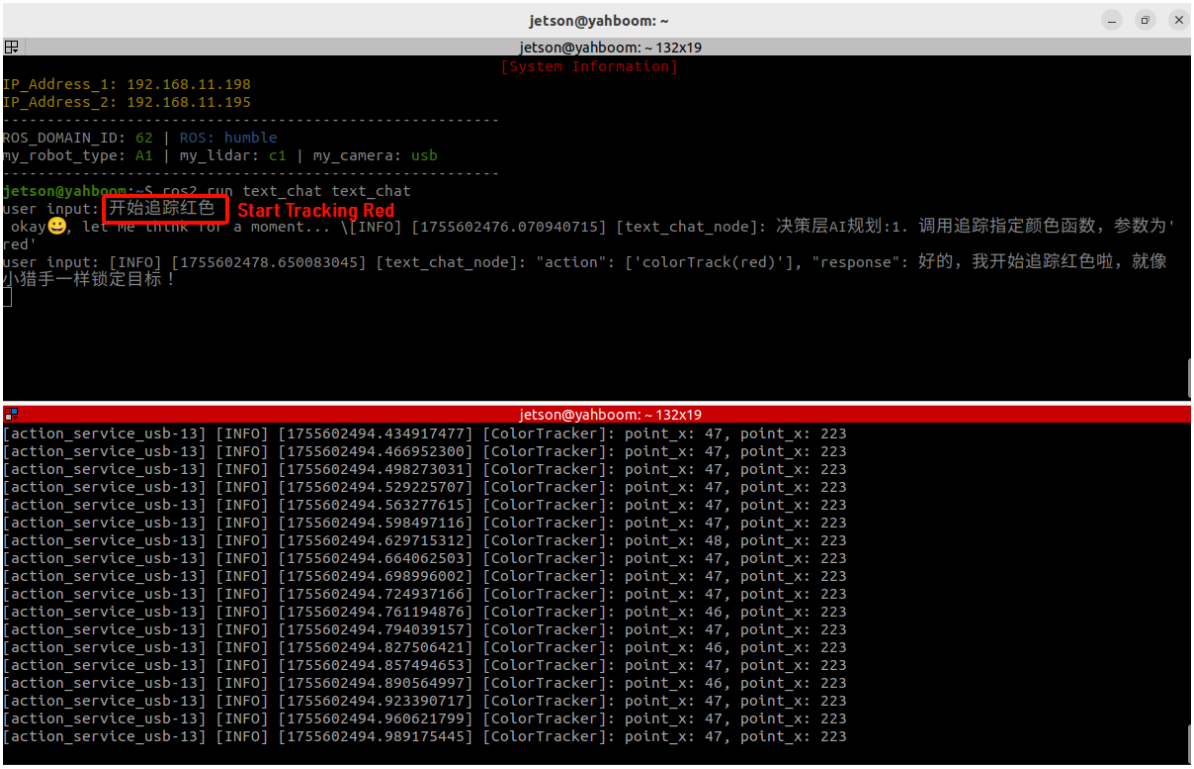
  Color tracking, including: red, green, blue, and yellow (color calibration is required according to the **AI Large Model Preparation** tutorial).

  Object tracking

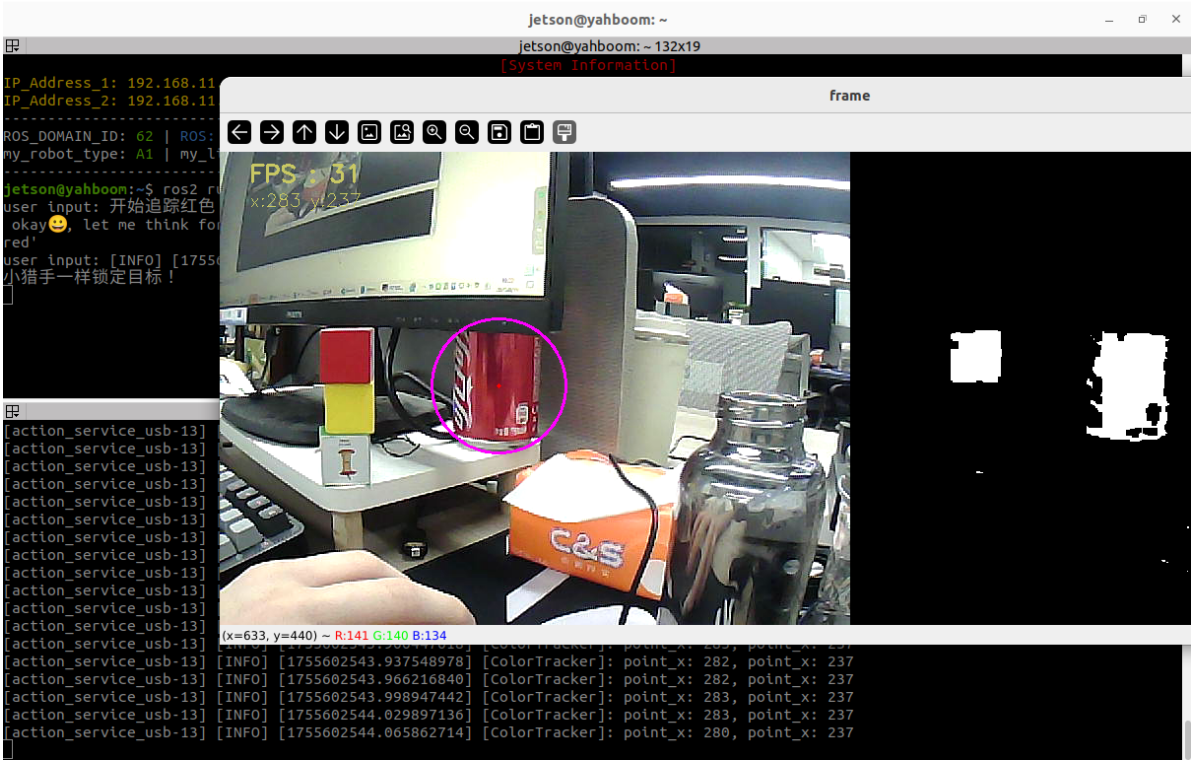
⚠ Please do not end the text with a period or any other characters!

3.2.1 Example 1: "Start Tracking Red"

Type "Start Tracking Red" in the terminal. The terminal will print the following information:

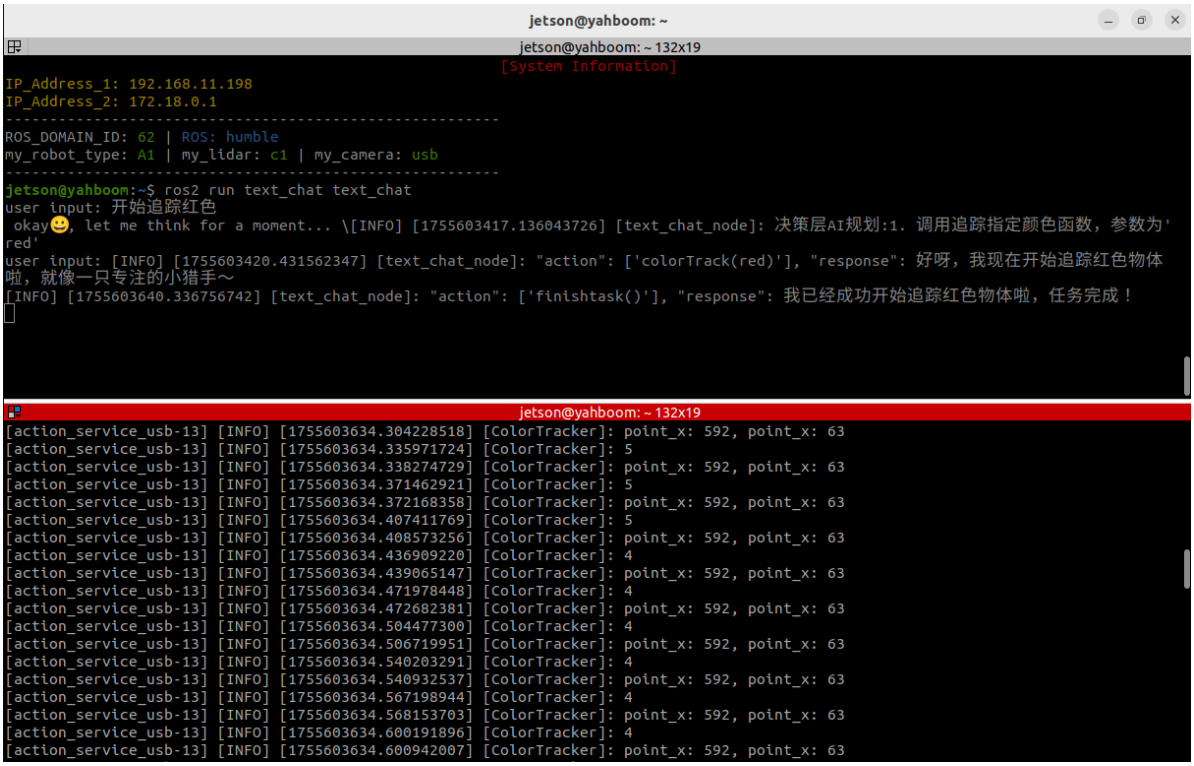


A window titled **frame** will open on the VNC screen, displaying the image from the robot's current perspective.

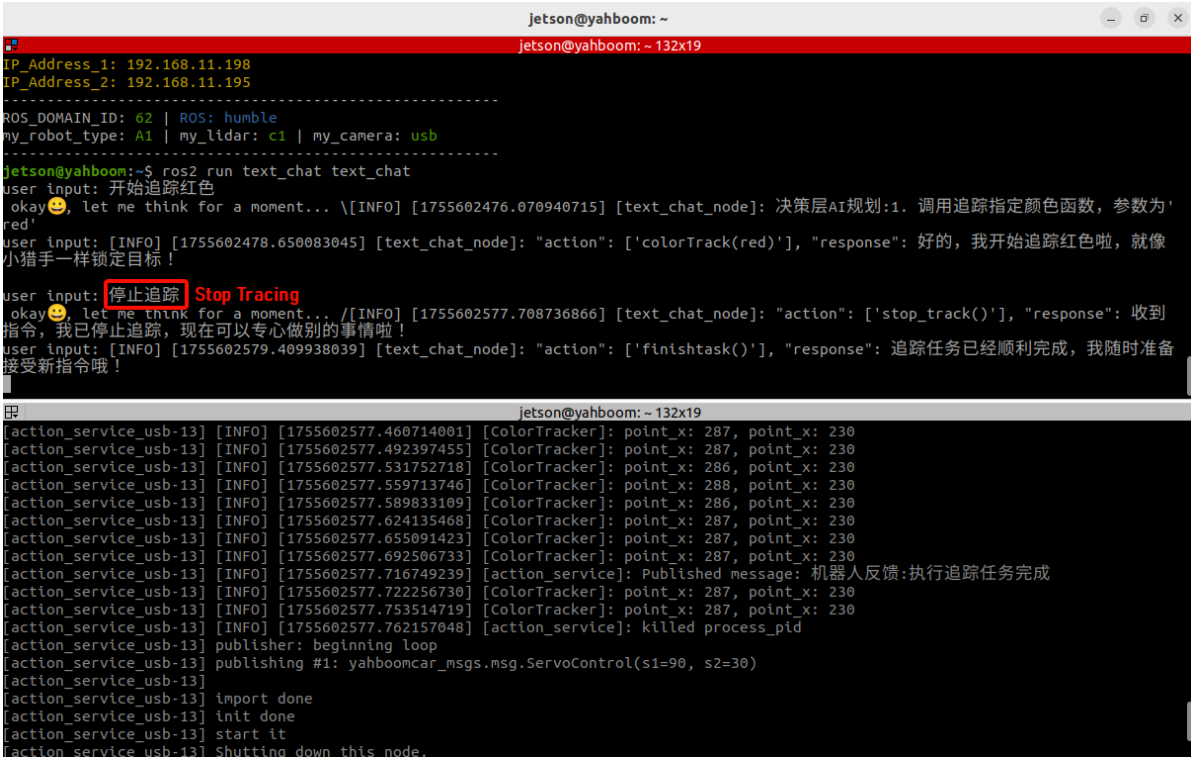


Move the object slowly, and the servo and gimbal will follow.

If there is no target to track in the image, the program will count down for 10 seconds, and the terminal will print a 5-second countdown. The process will automatically end, and the task will be considered complete.



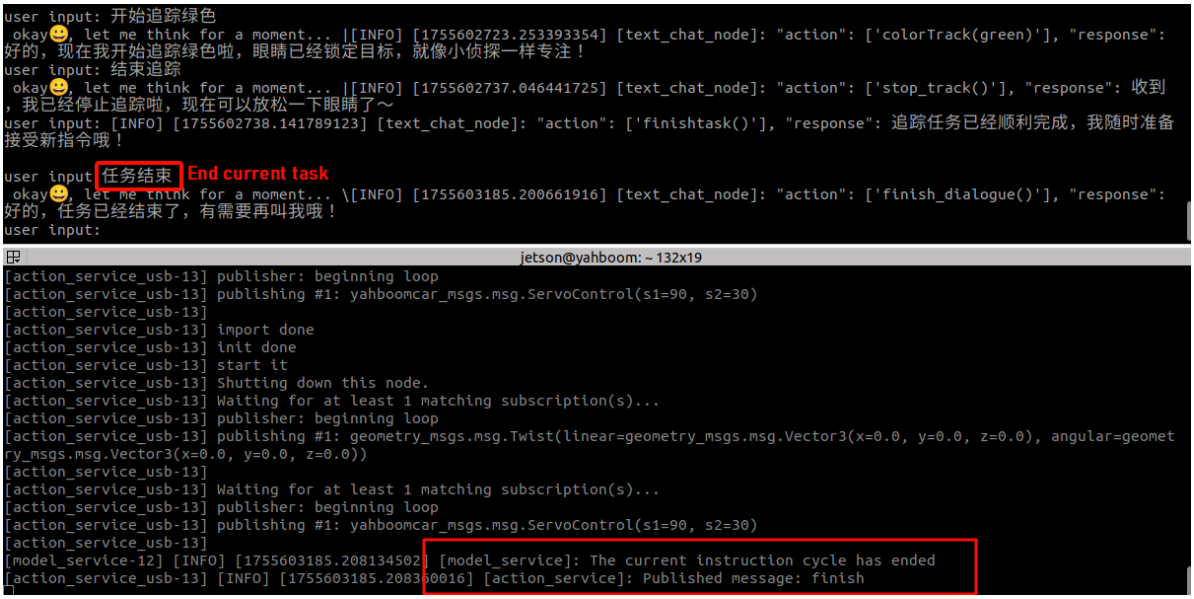
To manually end the task, press `ENTER` in the terminal. Press the key to continue the dialog input, **[Stop Tracing]** or **[End Tracing]**



🔗 If the following warning appears when ending tracing, this is normal. It indicates that the child process has been killed and will not affect normal operation of the program.

```
[action_service_usb-13] Failed to publish log message to rosout: publisher's
context is invalid, at ./src/rcl/publisher.c:389
[action_service_usb-13] Exception in thread Thread-284 (execute):
[action_service_usb-13] rclpy._rclpy_pybind11.RCLError: Failed to publish:
publisher's context is invalid, at ./src/rcl/publisher.c:389
```

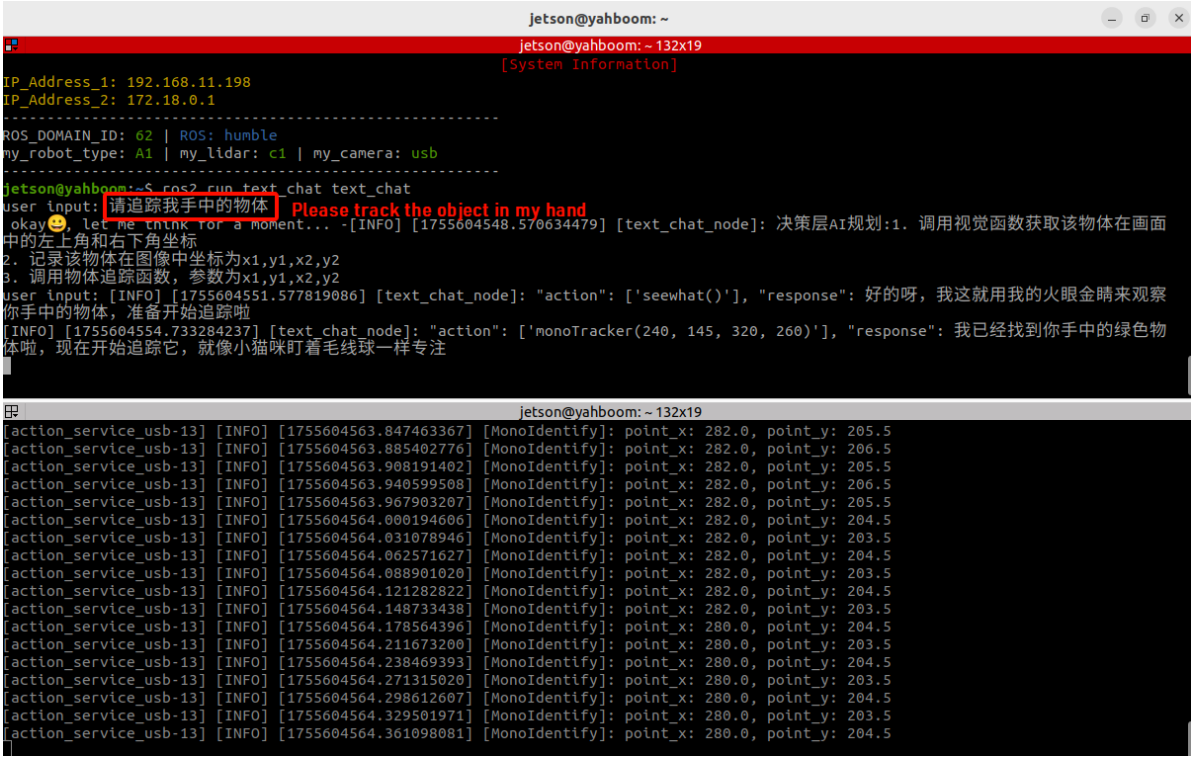
After completing a task, the robot enters a waiting state. Instructions are passed directly to the execution layer model, and all conversation history is retained. You can enter the "**End current task**" command again to end the current task cycle and start a new one.



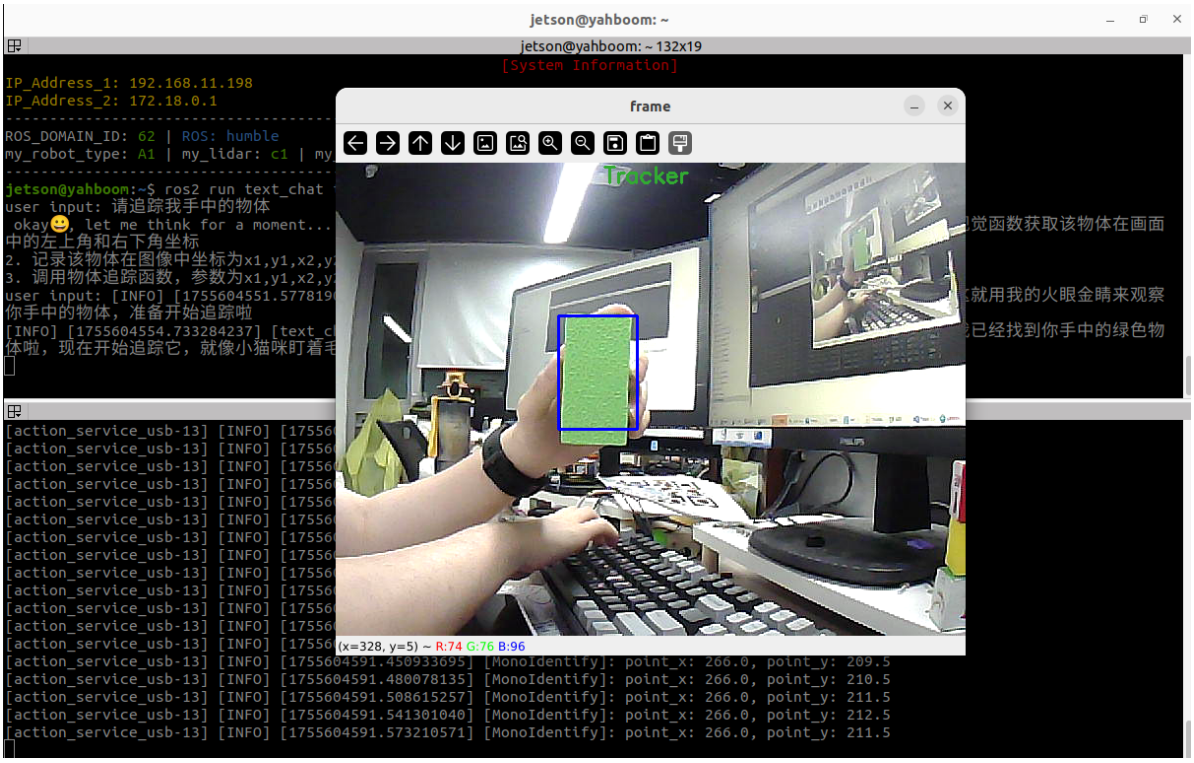
3.2.2 Example 2: "Please track the object in my hand"

⚠ The coordinates obtained in this example are derived entirely from the inference of a large AI model. Therefore, it is recommended to use a newer model for better results!

Type "Please track the object in my hand" in the terminal. The terminal prints the following information:

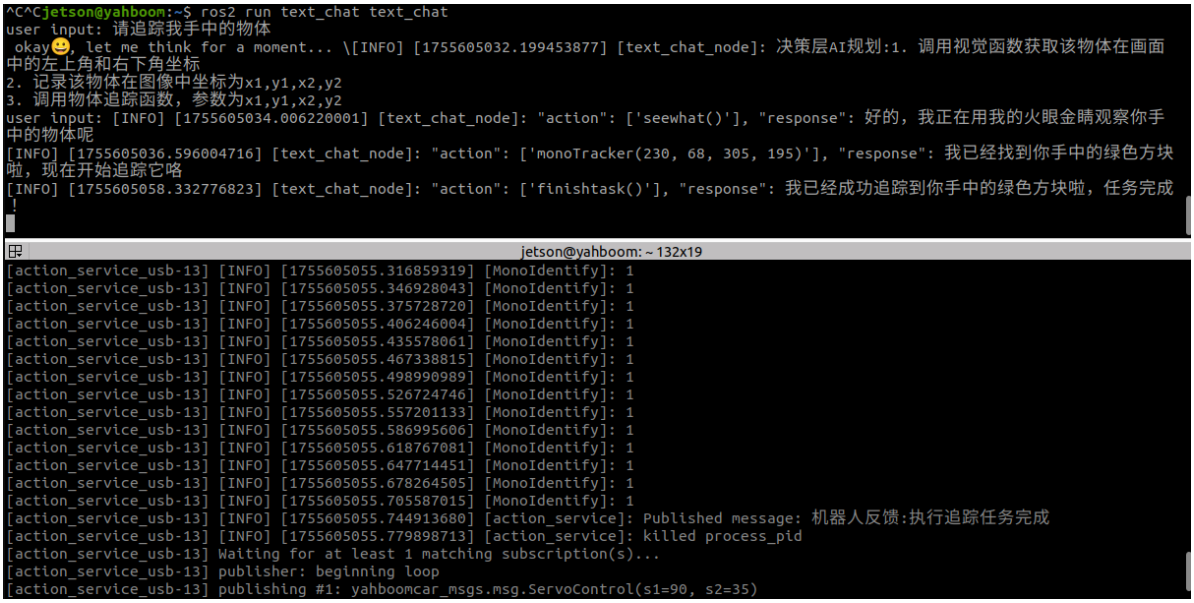


Please hold any object in your hand and place it in the field of view until the tracking frame appears. A window titled **frame** will open on the VNC screen to display the image from the current robot's perspective.

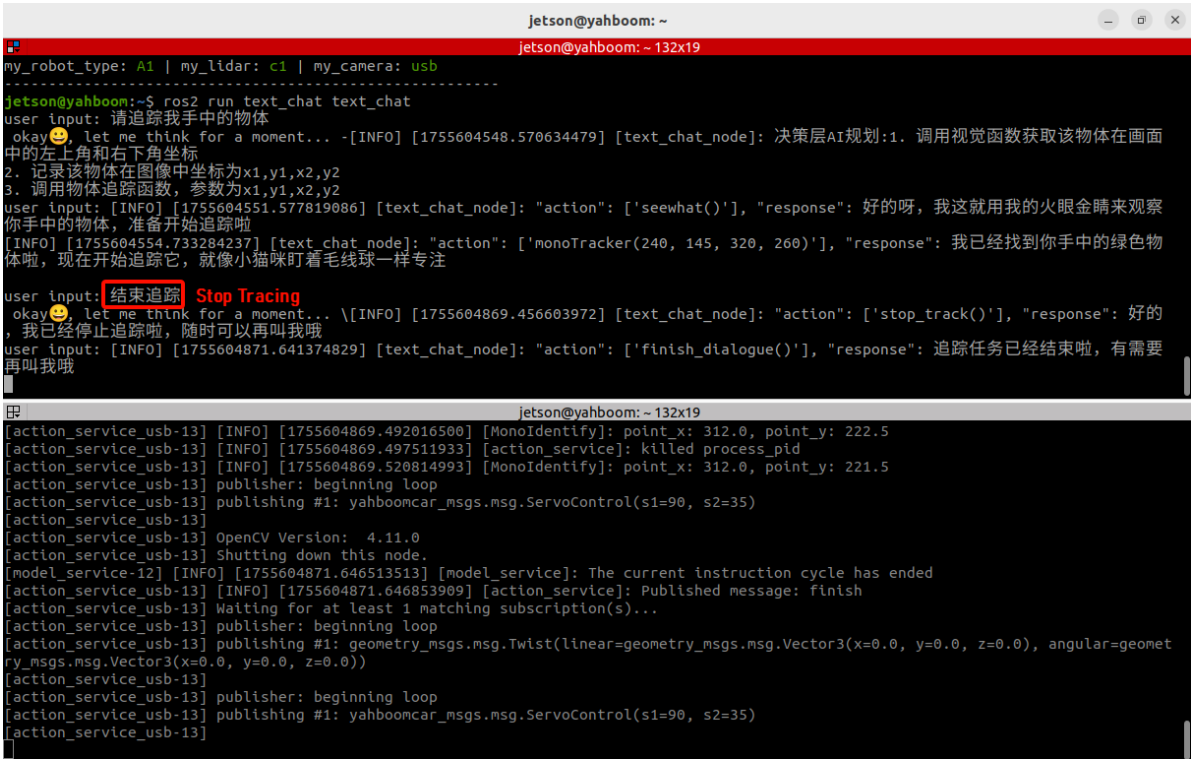


Slowly move the object, and the servo gimbal will follow.

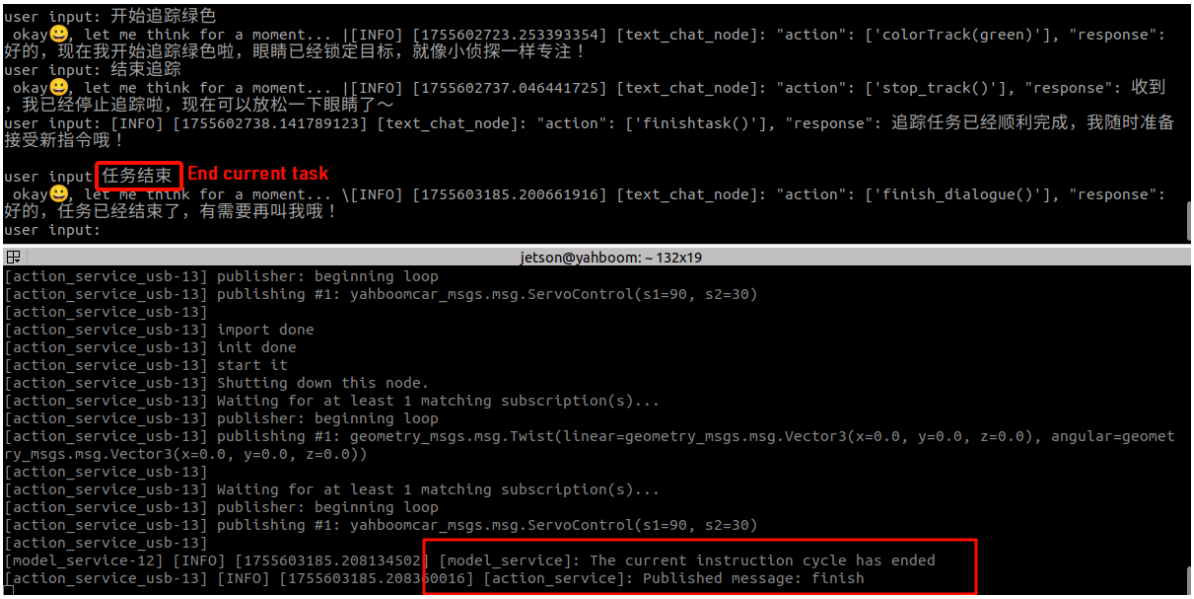
If there's no target to track in the image, the program will count down for 10 seconds, and the terminal will print a 5-second countdown. The process will automatically end, and the mission will be considered complete.



To manually end the task, press `ENTER` in the terminal, continue the dialog input, **[Stop Tracking]** or **[End Tracking]**



After completing a task, the robot enters a waiting state. Instructions are passed directly to the execution layer model, and all conversation history is retained. You can enter the "End current task" command again to end the current task cycle and start a new one.



4.Source code analysis

Source code located at:

Jetson Nano host:

```
/home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_usb.py
```

Jetson Nano, Raspberry Pi host:

You need to first enter Docker.

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_usb.py
```

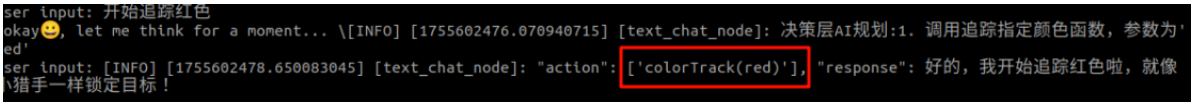
4.1 Example 1

action_service_usb.py:

In Example 1, the **seewhat**, **color_follow**, and **stop_track** methods in the **CustomActionServer** class are used.

- The **seewhat** function primarily retrieves the camera's color image.
- The **colorTrack(self, color)** function performs color tracking.
- The **stop_track()** function issues a stop tracking command.

This section focuses on the **colorTrack(self, color)** function, which requires a color parameter, which can be 'red', 'green', 'blue', or 'yellow'.




```
#Start the color tracking subprocess program
process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl', 'colorTracker','--ros-args','-p',f'target_color:={target_color}'])
```

The startup program source code path is:

`~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_voice_ctrl/yahboomcar_voice_ctrl/colorTracker.py`

```
def colorTrack(self,color):
    try:
        self.colorTracker_future = Future()
        color = color.strip("'\"")
        if color == 'red':
            target_color = int(1)
        elif color == 'green':
            target_color = int(2)
        elif color == 'blue':
            target_color = int(3)
        elif color == 'yellow':
            target_color = int(4)
        else:
            target_color = int(1)
        process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl', 'colorTracker','--ros-args','-p',f'target_color:={target_color}'])
        while not self.colorTracker_future.done():
            if self.interrupt_flag:
                break
            time.sleep(0.1)
        self.get_logger().info(f'killed process_pid')
        self.kill_process_tree(process_1.pid)
        self.cancel()
    except:
        self.get_logger().error('colorTrack Startup failure')
        return
```

When the large model receives the user input of the [Stop Tracking] or [End Tracking] command,

or when the tracking target is lost for more than 10 seconds,

The **stop_track** method will be called to send the future.done signal. After that, `while not self.colorTracker_future.done()` in the **colorTrack** function will exit the blocking state. Then the **kill_process_tree** method will be called to recursively kill the process tree of the child process. Finally, the status of the execution action will be fed back to the execution layer model.

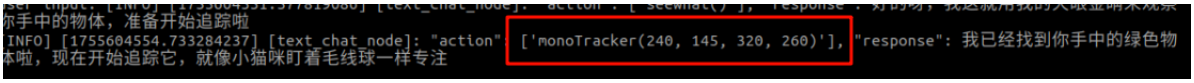
4.2 Example 2

action_service_usb.py Program:

In Example 2, the **seewhat**, **monoTracker**, and **stop_track** methods in the **CustomActionServer** class are used.

- The **seewhat** function primarily obtains the camera's color image.
- The **monoTracker(self,x1,y1,x2,y2)** function performs object tracking.
- The **stop_track()** function issues a stop tracking command.

The **seewhat** function primarily obtains the camera's color image. The **monoTracker(self,x1,y1,x2,y2)** function takes as parameters the coordinates of the upper-left and lower-right vertices of the object's bounding box to be tracked (the upper-left corner of the image is the pixel origin). For example, the coordinates of the outer bounding box of the green square identified in Example 2 can be found in the response from the large model: the upper-left corner is (240,145) and the lower-right corner is (320,260).




```
#Start the object tracking subprocess program
process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl',
'monoTracker','--ros-args','-p',f'x1:={x1}','-p',f'y1:={y1}','-p',f'x2:={x2}','-
p',f'y2:={y2}'])
```

The startup program source code path is:

`~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_voice_ctrl/yahboomcar_voice_ctrl/monoTracker.py`

```
def monoTracker(self,x1,y1,x2,y2):
    try:
        self.monoTracker_future = Future()
        x1 = int(x1)
        x2 = int(x2)
        y1 = int(y1)
        y2 = int(y2)
        process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl',
'monoTracker','--ros-args','-p',f'x1:={x1}','-p',f'y1:={y1}','-p',f'x2:={x2}','-
p',f'y2:={y2}'])
        while not self.monoTracker_future.done():
            if self.interrupt_flag:
                break
            time.sleep(0.1)
        self.get_logger().info(f'killed process_pid')
        self.kill_process_tree(process_1.pid)
```

```
            self.cancel()
        except:
            self.get_logger().error('monoTracker Startup failure')
            return
```

When the large model receives the user input of the [Stop Tracking] or [End Tracking] command,

or when the tracking target is lost for more than 10 seconds,

The **stop_track** method will be called to send the future.done signal. After that, `while not self.monoTracker_future.done()` in the **monoTracker** function will exit the blocking state. Then the **kill_process_tree** method will be called to recursively kill the process tree of the child process. Finally, the status of the execution action will be fed back to the execution layer model.