

# Multimodal visual understanding + SLAM navigation

## 1. Course Content

1. Learn to use the robot's visual understanding and SLAM navigation functions
2. Learn new key source codes

## 2. Preparation

### 2.1 Content Description

This course uses the Jetson Orin NX as an example. For Raspberry Pi and Jetson Nano boards, you need to open a terminal on the host 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 host computer, refer to the **[Configuration and Operation Guide] -- [Enter the Docker (Jetson Nano and Raspberry Pi 5 users see here)]** section of this product tutorial. For Orin and NX boards, simply open a terminal and enter the commands mentioned in this course.

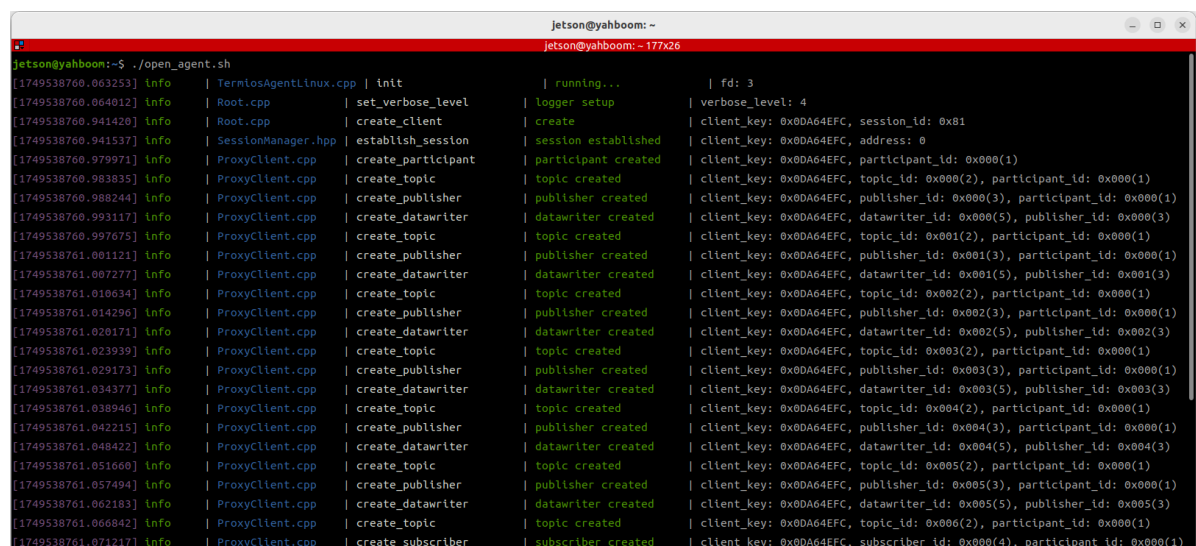
### 2.2 Start the Agent

**Note:** To test all cases, you must start the docker agent first. If it has already been started, you do not need to start it again.

Enter the command in the vehicle terminal:

```
sh start_agent.sh
```

The terminal prints the following information, indicating that the connection is successful



```
Jetson@yahboom: ~  
Jetson@yahboom:~$ ./open_agent.sh  
[1749538760.063253] Info | TermiosAgentLinux.cpp | Init | running... | fd: 3  
[1749538760.064012] Info | Root.cpp | set_verbose_level | logger setup | verbose_level: 4  
[1749538760.941420] Info | Root.cpp | create_client | create | client_key: 0x0DA64EFC, session_id: 0x81  
[1749538760.941537] Info | SessionManager.hpp | establish_session | session established | client_key: 0x0DA64EFC, address: 0  
[1749538760.979971] Info | ProxyClient.cpp | create_participant | participant created | client_key: 0x0DA64EFC, participant_id: 0x000(1)  
[1749538760.983835] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x000(2), participant_id: 0x000(1)  
[1749538760.988244] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x0DA64EFC, publisher_id: 0x000(3), participant_id: 0x000(1)  
[1749538760.993117] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x0DA64EFC, datawriter_id: 0x000(5), publisher_id: 0x000(3)  
[1749538760.997675] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x001(2), participant_id: 0x000(1)  
[1749538761.001121] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x0DA64EFC, publisher_id: 0x001(3), participant_id: 0x000(1)  
[1749538761.007277] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x0DA64EFC, datawriter_id: 0x001(5), publisher_id: 0x001(3)  
[1749538761.010634] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x002(2), participant_id: 0x000(1)  
[1749538761.014296] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x0DA64EFC, publisher_id: 0x002(3), participant_id: 0x000(1)  
[1749538761.020171] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x0DA64EFC, datawriter_id: 0x002(5), publisher_id: 0x002(3)  
[1749538761.023939] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x003(2), participant_id: 0x000(1)  
[1749538761.029173] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x0DA64EFC, publisher_id: 0x003(3), participant_id: 0x000(1)  
[1749538761.034377] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x0DA64EFC, datawriter_id: 0x003(5), publisher_id: 0x003(3)  
[1749538761.038946] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x004(2), participant_id: 0x000(1)  
[1749538761.042215] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x0DA64EFC, publisher_id: 0x004(3), participant_id: 0x000(1)  
[1749538761.048422] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x0DA64EFC, datawriter_id: 0x004(5), publisher_id: 0x004(3)  
[1749538761.051660] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x005(2), participant_id: 0x000(1)  
[1749538761.057494] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x0DA64EFC, publisher_id: 0x005(3), participant_id: 0x000(1)  
[1749538761.062183] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x0DA64EFC, datawriter_id: 0x005(5), publisher_id: 0x005(3)  
[1749538761.066842] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x0DA64EFC, topic_id: 0x006(2), participant_id: 0x000(1)  
[1749538761.071217] Info | ProxyClient.cpp | create_subscriber | subscriber created | client_key: 0x0DA64EFC, subscriber_id: 0x000(4), participant_id: 0x000(1)
```

**Note:** This chapter requires you to complete at least one SLAM map according to the LiDAR chapter course.

## 2.3 Configuring map mapping files

For more details on the principles and concepts of map mapping, please refer to the course content of [2. Basic Knowledge of AI Large Model - 3. Embodied Intelligent Robot System Architecture]

Connect to the robot desktop via VNC, create two new terminals, and start the first terminal first.

```
ros2 launch M3Pro_navigation base_bringup.launch.py
```

Then start in the second terminal

```
ros2 launch M3Pro_navigation navigation2.launch.py
```

Create a new terminal on the virtual side and start

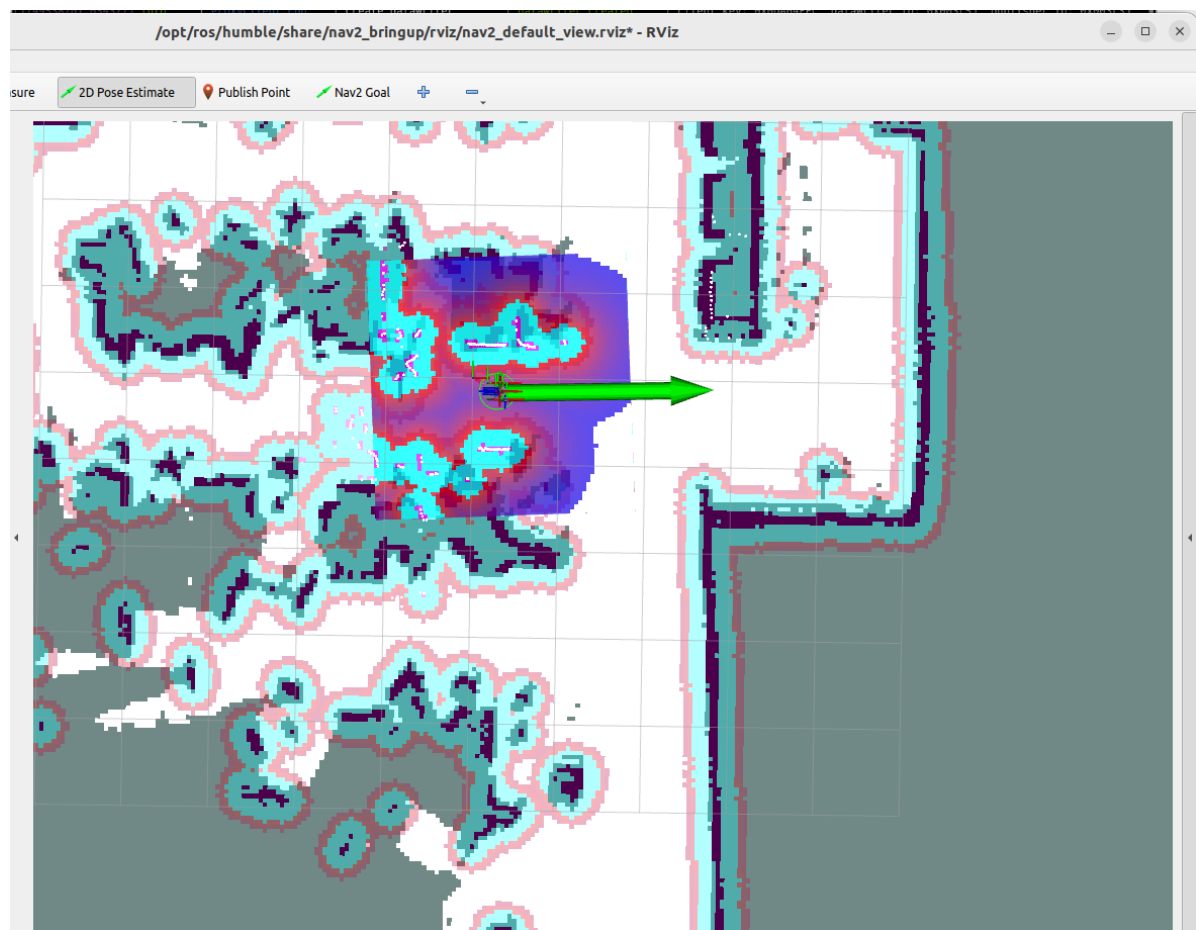
```
ros2 launch slam_view nav_rviz.launch.py
```

Or display it in rviz on the car side

```
ros2 launch M3Pro_navigation nav_rviz.launch.py
```

The rviz2 visualization interface will then open. Click **2D Pose Estimate** in the upper toolbar to enter the selection state and roughly mark the robot's position and orientation on the map.

The robot's model will be displayed on the map as shown below:



Create a new terminal in the virtual machine again and enter the command

```
ros2 topic list
```

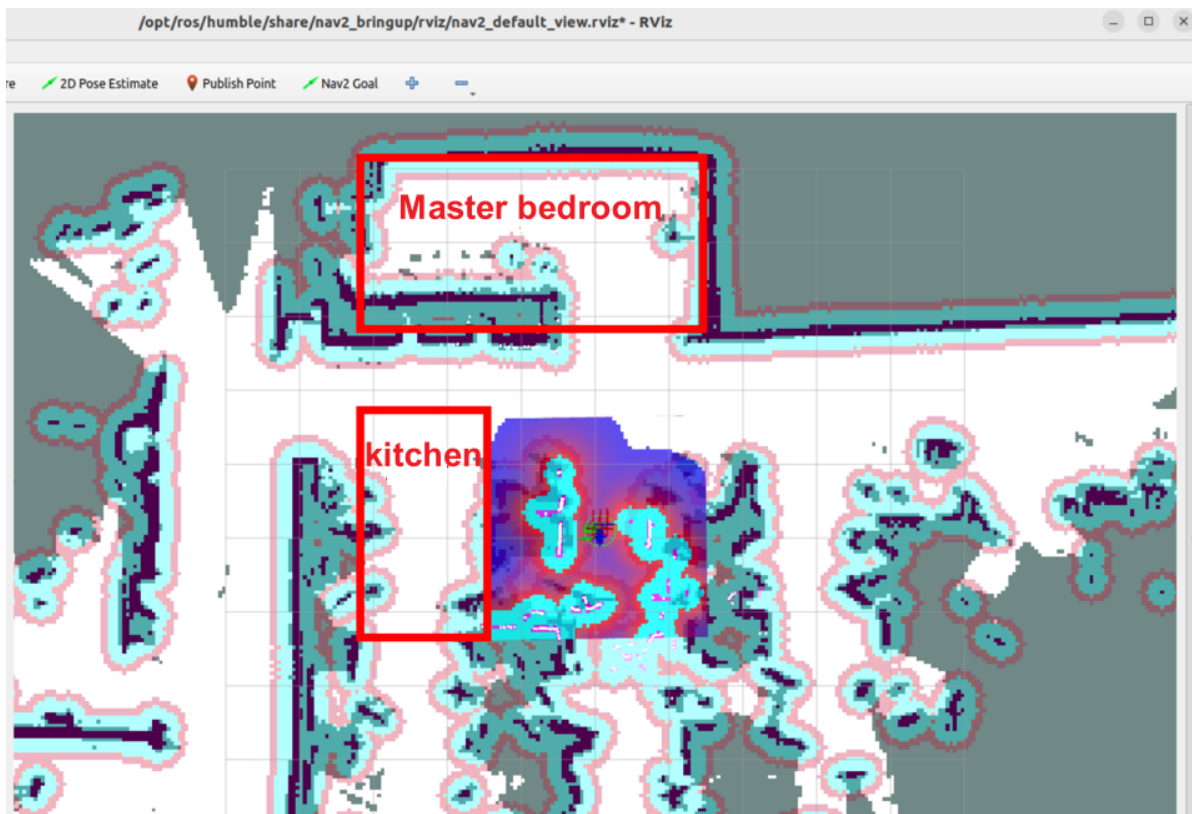
```
yahboom@yahboom-virtual-machine: ~  
yahboom@yahboom-virtual-machine: ~ 104x26  
/bt_navigator/transition_event  
/clicked_point  
/cmd_vel  
/cmd_vel_nav  
/controller_server/transition_event  
/cost_cloud  
/diagnostics  
/downsampled_costmap  
/downsampled_costmap_updates  
/evaluation  
/global_costmap/clearing_endpoints  
/global_costmap/costmap  
/global_costmap/costmap_raw  
/global_costmap/costmap_updates  
/global_costmap/footprint  
/global_costmap/global_costmap/transition_event  
/global_costmap/published_footprint  
/global_costmap/voxel_grid  
/global_costmap/voxel_marked_cloud  
/goal_pose  
/imu/data  
/imu/data_raw  
/imu/mag  
/initialpose  
/joint_states  
/local_costmap/clearing_endpoints
```

You can see the topic **/initialpose** , which is the topic for using the **2D Pose Estimate** tool in rviz2 in the previous step . We can use this tool to publish coordinates to view the data of the **/initialpose** topic to obtain the coordinates and orientation angle of a certain point on the map.

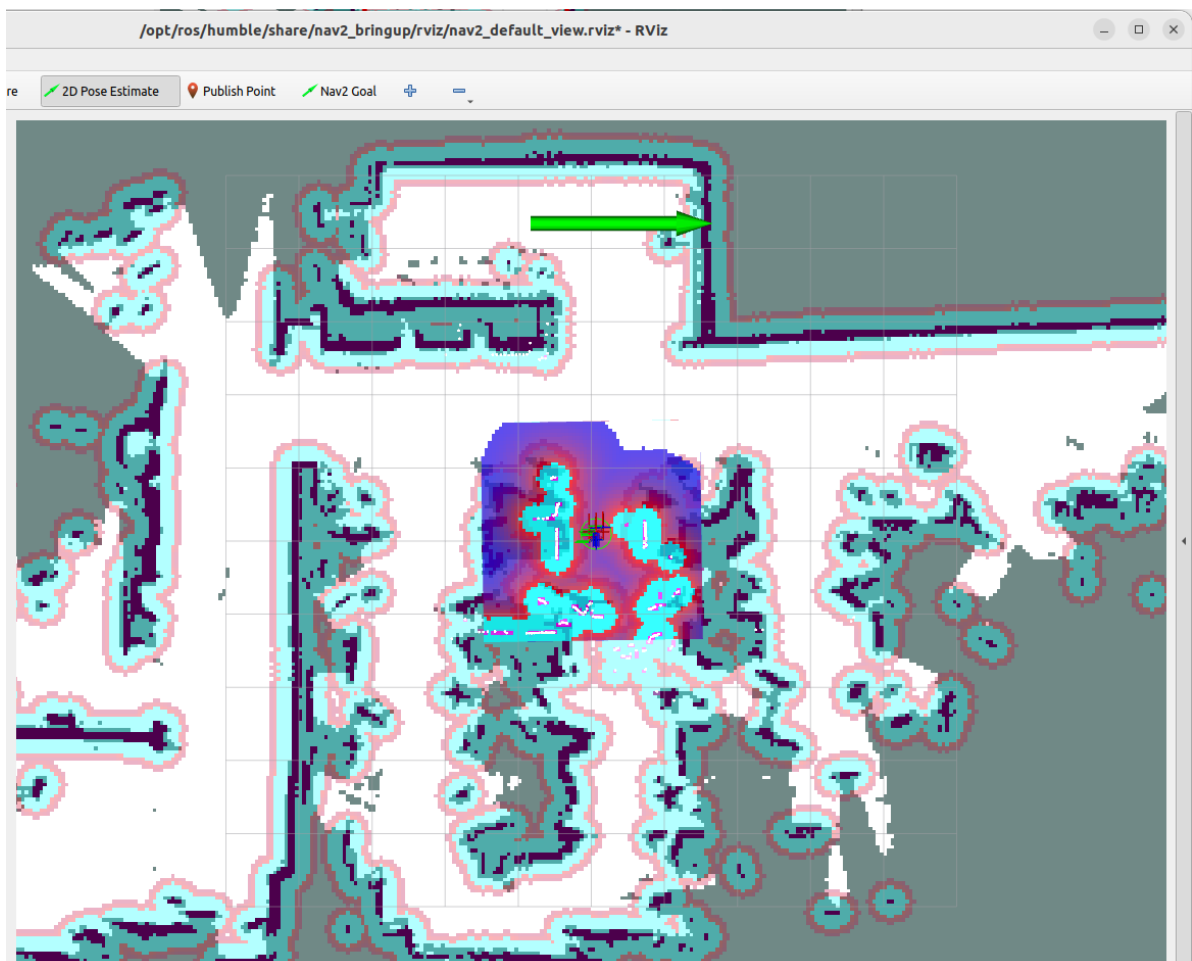
Enter the following command in the terminal to observe the data on the **/initialpose** topic:

```
ros2 topic echo /initialpose
```

We can name any point on the map. Here we take "Master Bedroom" and "Kitchen" as examples.



As shown in the figure below, we first click the **2D Pose Estimate** tool, then click and hold the left button of the mouse anywhere in the "Master Bedroom" area to adjust the direction, and release it after confirming.



The robot position will be adjusted to the position and orientation just selected. You can preview the robot's expected position and orientation in rviz



Jetson Nano, Raspberry Pi host, you need to enter Docker first:

```
root/M3Pro_ws/src/largemodel/config/map_mapping.yaml
```

Replace the "name" in A with "master bedroom", and the "position" and "orientation" data with the data we just saw in the terminal **/initialpose topic**

```
#根据实际的场景环境，自定义地图中的区域，可以添加任意个区域，注意和大模型的地图映射保持一致即可
#According to the actual scene environment, customize the areas in the map. You
can add any number of areas, just make sure they are consistent with the map
mapping of the large model
```

```
#地图映射Map mapping
```

```
A:
```

```
  name: 'Master bedroom'
  position:
    x: 1.6878514289855957
    y: -0.32615160942077637
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.4503881236302555
    w: 0.8928328724306793
```

```
#此处可新增添加区域对应的栅格地图坐标点，注意和上面格式保持一致
```

```
#Here, you can add the raster map coordinate points corresponding to the added
area. Please note that the format should be consistent with the above
```

In the same way, we can add a map of the kitchen

```
#根据实际的场景环境，自定义地图中的区域，可以添加任意个区域，注意和大模型的地图映射保持一致即可
#According to the actual scene environment, customize the areas in the map. You
can add any number of areas, just make sure they are consistent with the map
mapping of the large model
```

```
#地图映射Map mapping
```

```
A:
```

```
  name: 'Master bedroom'
  position:
    x: 1.6878514289855957
    y: -0.32615160942077637
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.4503881236302555
    w: 0.8928328724306793
```

```
G:
```

```
  name: 'kitchen'
  position:
    x: 2.367471694946289
    y: -1.568994164466858
    z: 0.0
  orientation:
```

```
x: 0.0
y: 0.0
z: -0.3106425066197889
w: 0.950526818706855
```

#此处可新增添加区域对应的栅格地图坐标点，注意和上面格式保持一致

#Here, you can add the raster map coordinate points corresponding to the added area. Please note that the format should be consistent with the above

Then switch to the workspace in the vehicle terminal `M3Pro_ws/`: recompile the largemodel function package to take effect

jetson orin nano, jetson orin NX host:

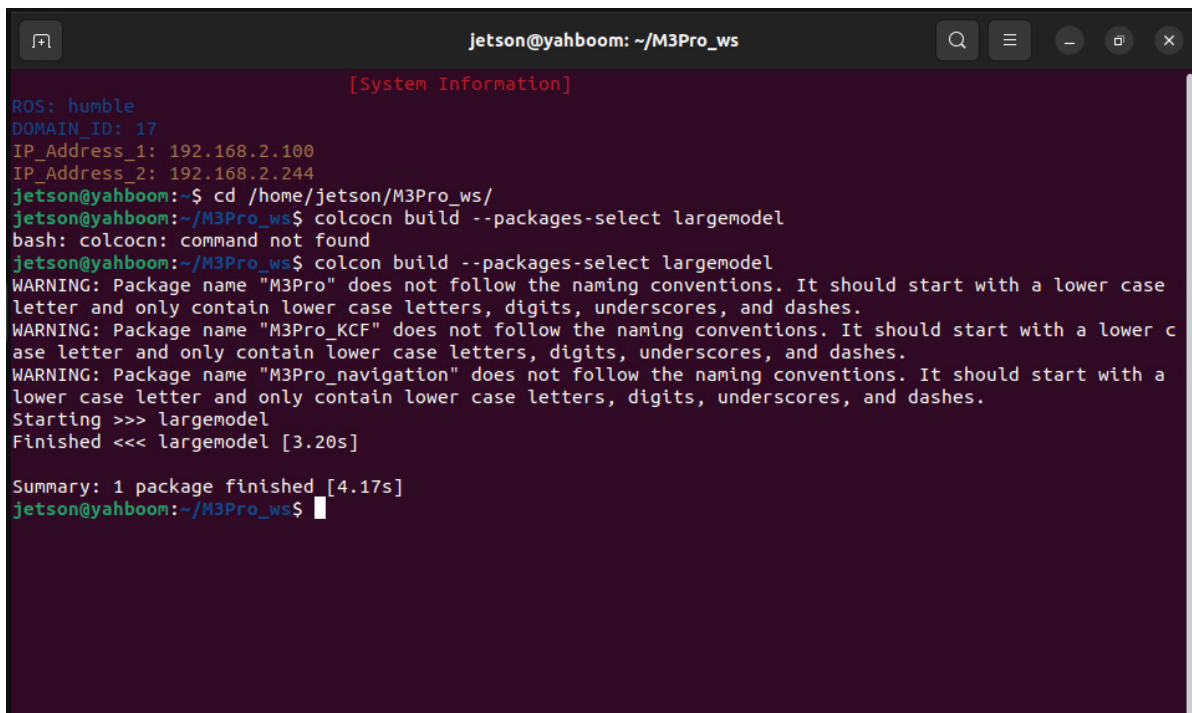
```
cd /home/jetson/M3Pro_ws/
```

Jetson Nano, Raspberry Pi host, you need to enter Docker first:

```
cd root/M3Pro_ws/
```

Recompile the feature package:

```
colcon build --packages-select largemodel
```

A terminal window titled 'Jetson@yahboom: ~/M3Pro\_ws' with standard Ubuntu window controls. It displays ROS system information including ROS version (humble), domain ID (17), and IP addresses. The user runs 'cd /home/jetson/M3Pro\_ws/' and then 'colcon build --packages-select largemodel'. The terminal shows a warning about package naming conventions for 'M3Pro', 'M3Pro\_KCF', and 'M3Pro\_navigation'. The build process for 'largemodel' starts and finishes successfully within 3.20 seconds. A summary shows 1 package finished in 4.17 seconds. The prompt returns to 'jetson@yahboom:~/M3Pro\_ws\$' with a cursor.

```
Jetson@yahboom: ~/M3Pro_ws
[System Information]
ROS: humble
DOMAIN_ID: 17
IP_Address_1: 192.168.2.100
IP_Address_2: 192.168.2.244
jetson@yahboom:~$ cd /home/jetson/M3Pro_ws/
jetson@yahboom:~/M3Pro_ws$ colcon build --packages-select largemodel
bash: colcon: command not found
jetson@yahboom:~/M3Pro_ws$ colcon build --packages-select largemodel
WARNING: Package name "M3Pro" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits, underscores, and dashes.
WARNING: Package name "M3Pro_KCF" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits, underscores, and dashes.
WARNING: Package name "M3Pro_navigation" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits, underscores, and dashes.
Starting >>> largemodel
Finished <<< largemodel [3.20s]

Summary: 1 package finished [4.17s]
jetson@yahboom:~/M3Pro_ws$
```

### 3. Run the case

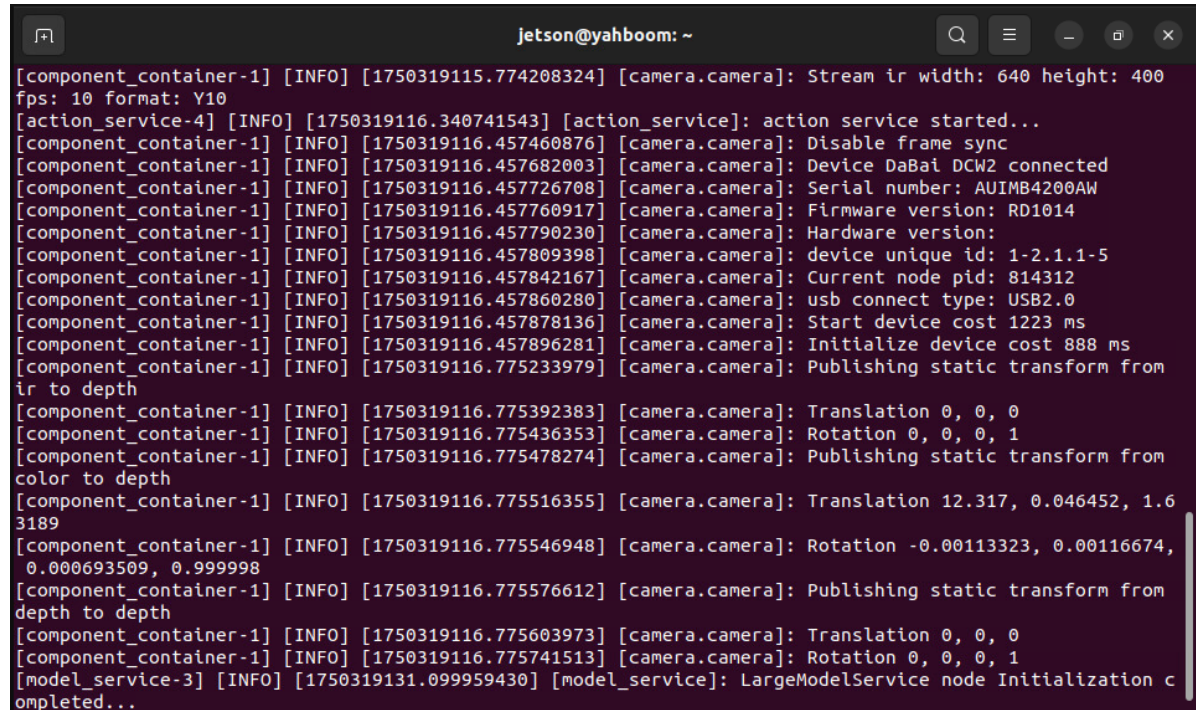


## 3.1 Startup Program

Connect to the robot desktop via VNC, open a terminal, and start the command

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

After initialization is complete, the following content will be displayed

A terminal window titled 'jetson@yahboom: ~' displays the output of the command 'ros2 launch largemodel largemodel\_control.launch.py text\_chat\_mode:=True'. The output consists of multiple INFO messages from various components. The 'camera.camera' component logs details about the DaBai DCW2 camera, including stream resolution (640x400), frame sync settings, device serial number (AUIMB4200AW), firmware version (RD1014), hardware version, device unique ID (1-2.1.1-5), current node PID (814312), USB connect type (USB2.0), start device cost (1223 ms), and initialization device cost (888 ms). It also publishes static transforms from 'ir to depth' and 'color to depth'. The 'action\_service' component logs that the action service started. The 'model\_service' component logs that the LargeModelService node initialization is completed.

```
[component_container-1] [INFO] [1750319115.774208324] [camera.camera]: Stream ir width: 640 height: 400
fps: 10 format: Y10
[action_service-4] [INFO] [1750319116.340741543] [action_service]: action service started...
[component_container-1] [INFO] [1750319116.457460876] [camera.camera]: Disable frame sync
[component_container-1] [INFO] [1750319116.457682003] [camera.camera]: Device DaBai DCW2 connected
[component_container-1] [INFO] [1750319116.457726708] [camera.camera]: Serial number: AUIMB4200AW
[component_container-1] [INFO] [1750319116.457760917] [camera.camera]: Firmware version: RD1014
[component_container-1] [INFO] [1750319116.457790230] [camera.camera]: Hardware version:
[component_container-1] [INFO] [1750319116.457809398] [camera.camera]: device unique id: 1-2.1.1-5
[component_container-1] [INFO] [1750319116.457842167] [camera.camera]: Current node pid: 814312
[component_container-1] [INFO] [1750319116.457860280] [camera.camera]: usb connect type: USB2.0
[component_container-1] [INFO] [1750319116.457878136] [camera.camera]: Start device cost 1223 ms
[component_container-1] [INFO] [1750319116.457896281] [camera.camera]: Initialize device cost 888 ms
[component_container-1] [INFO] [1750319116.775233979] [camera.camera]: Publishing static transform from
ir to depth
[component_container-1] [INFO] [1750319116.775392383] [camera.camera]: Translation 0, 0, 0
[component_container-1] [INFO] [1750319116.775436353] [camera.camera]: Rotation 0, 0, 0, 1
[component_container-1] [INFO] [1750319116.775478274] [camera.camera]: Publishing static transform from
color to depth
[component_container-1] [INFO] [1750319116.775516355] [camera.camera]: Translation 12.317, 0.046452, 1.6
3189
[component_container-1] [INFO] [1750319116.775546948] [camera.camera]: Rotation -0.00113323, 0.00116674,
0.000693509, 0.999998
[component_container-1] [INFO] [1750319116.775576612] [camera.camera]: Publishing static transform from
depth to depth
[component_container-1] [INFO] [1750319116.775603973] [camera.camera]: Translation 0, 0, 0
[component_container-1] [INFO] [1750319116.775741513] [camera.camera]: Rotation 0, 0, 0, 1
[model_service-3] [INFO] [1750319131.099959430] [model_service]: LargeModelService node Initialization c
ompleted...
```

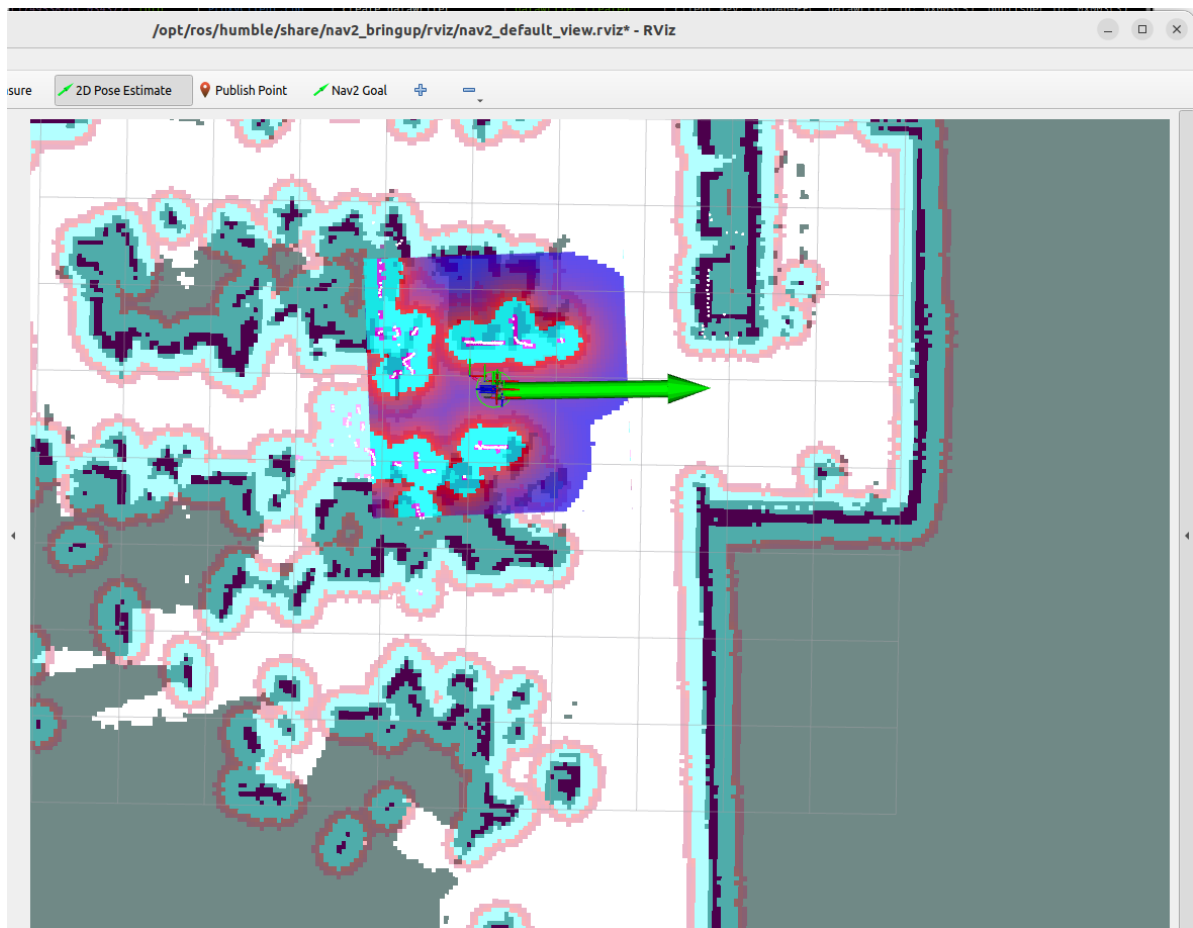
Open two terminals on the vehicle and enter the following commands to start the navigation function:

```
ros2 launch M3Pro_navigation base_bringup.launch.py
ros2 launch M3Pro_navigation navigation2.launch.py
```

Create a new terminal on the virtual end and enter the following command to start the rviz visualization window

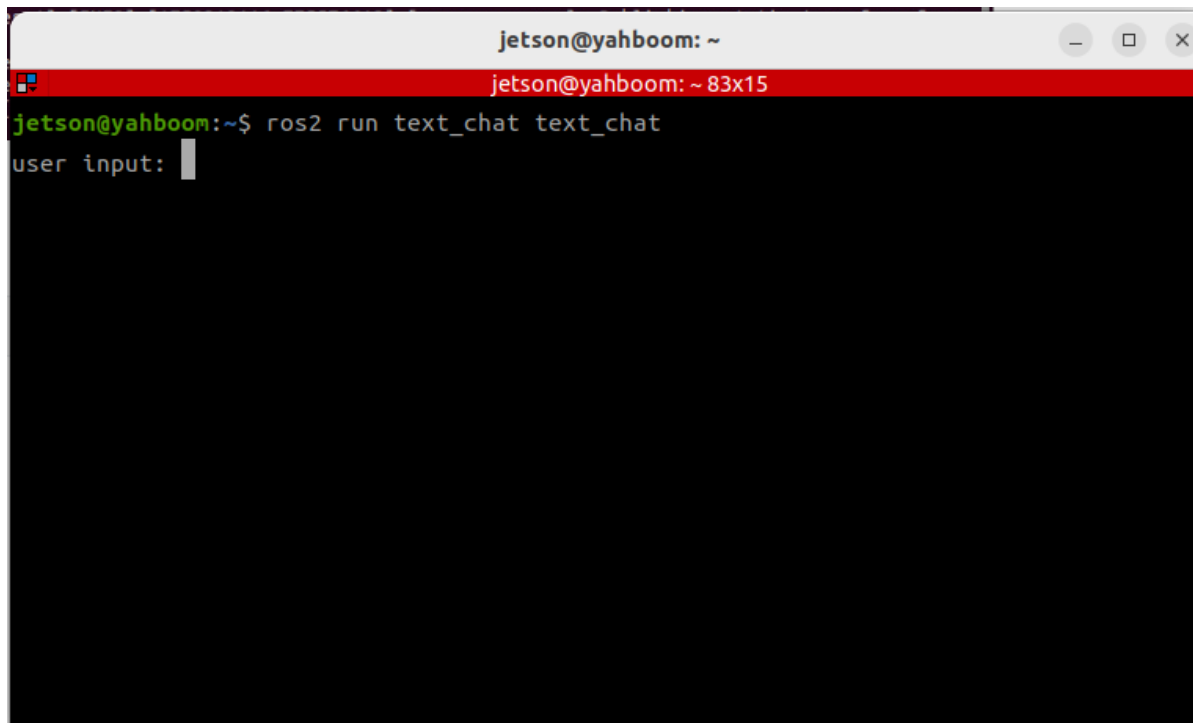
```
ros2 launch slam_view nav_rviz.launch.py
```

Then follow the process of starting the navigation function to initialize the positioning. The rviz2 visualization interface will open. Click **2D Pose Estimate** in the upper toolbar to enter the selection state. Roughly mark the position and orientation of the robot on the map. After initialization positioning, the preparation work is completed.



The virtual machine opens a terminal and starts the text interaction program:

```
ros2 run text_chat text_chat
```



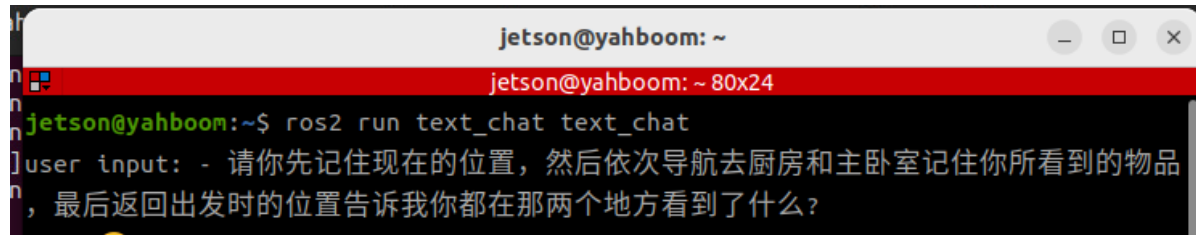
## 3.2 Test Cases

Here are some test cases for reference. Users can write their own dialogue commands.

- Please remember your current location first, then navigate to the kitchen and master bedroom in turn and remember what you see. Finally, return to your starting location and tell me what you see in each of these two places.

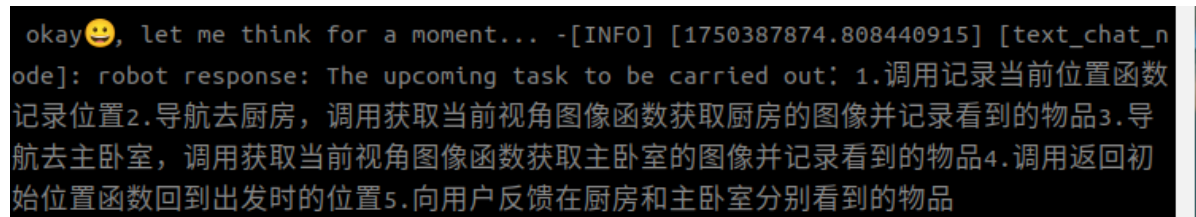
### 3.2.1 Case 1

Copy and enter the above test case in the text interactive terminal:



```
jetson@yahboom: ~  
jetson@yahboom: ~ 80x24  
jetson@yahboom:~$ ros2 run text_chat text_chat  
user input: - 请你先记住现在的位置，然后依次导航去厨房和主卧室记住你所看到的物品，最后返回出发时的位置告诉我你都在那两个地方看到了什么？
```

The decision-making model outputs the planned task steps:



```
okay😊, let me think for a moment... -[INFO] [1750387874.808440915] [text_chat_n  
ode]: robot response: The upcoming task to be carried out: 1.调用记录当前位置函数  
记录位置2.导航去厨房，调用获取当前视角图像函数获取厨房的图像并记录看到的物品3.导  
航去主卧室，调用获取当前视角图像函数获取主卧室的图像并记录看到的物品4.调用返回初  
始位置函数回到出发时的位置5.向用户反馈在厨房和主卧室分别看到的物品
```

The execution layer model will then execute the task steps:

```
user input: [INFO] [1750387878.350106583] [text_chat_node]: robot response: "action": ['get_current_pose()'], "response": 好的，我现在就记住现在的位置，就像记住了一个重要的宝藏地点一样。
[INFO] [1750387882.255020679] [text_chat_node]: robot response: "action": ['navigation(G)'], "response": 我已经记住了当前位置，现在准备导航去厨房啦，就像一个勇敢的探险家一样！
[INFO] [1750387948.665910436] [text_chat_node]: robot response: "action": ['navigation(G)'], "response": 哎呀，刚才去厨房的路上遇到了点小麻烦，不过别担心，我再试一次！
[INFO] [1750387957.637532129] [text_chat_node]: robot response: "action": ['seewh at()'], "response": 我已经成功到达厨房啦！现在我来观察一下这里都有些什么物品。
[INFO] [1750387963.313431519] [text_chat_node]: robot response: "action": [], "response": 在厨房里，我看到的是光滑的地板和一条绿色的线条，不过没有看到具体的物品呢。可能是视角问题，我们继续前进吧！现在我要去主卧室看看了。
[INFO] [1750387966.771073792] [text_chat_node]: robot response: "action": ['navigation(A)'], "response": 我已经告诉了你在厨房看到的情况，现在我将导航去主卧室，看看那里有什么有趣的发现！
[INFO] [1750388031.398516666] [text_chat_node]: robot response: "action": ['navigation(A)'], "response": 哎呀，去主卧室的路上似乎遇到了一点小问题，不过别担心，我再试一次！
[INFO] [1750388037.382963440] [text_chat_node]: robot response: "action": ['seewh at()'], "response": 我已经成功到达主卧室啦！现在我来观察一下这里都有些什么物品。
[INFO] [1750388045.402025419] [text_chat_node]: robot response: "action": [], "response": 在主卧室里，我看到的同样是光滑的地板，地板上有天然的纹理和一些反光的灯光。这里也没有看到具体的物品呢。可能是视角问题或者是房间比较空旷。现在，我将返回出发时的位置，并告诉你我在厨房和主卧室分别看到了什么。
[INFO] [1750388050.039975630] [text_chat_node]: robot response: "action": ['navigation(zero)'], "response": 我已经告诉了你在主卧室看到的情况，现在我将返回出发时的位置。
[INFO] [1750388091.405150056] [text_chat_node]: robot response: "action": [], "response": 我已经回到了出发时的位置。总结一下我的发现：在厨房，我看到的是光滑的地板和一条绿色的线条；而在主卧室，我看到的同样是光滑的地板，地板上有天然的纹理和一些反光的灯光。两个地方都没有看到具体的物品，可能是视角问题或者是房间比较空旷。这就是我的观察结果，希望对你有帮助！
[INFO] [1750388104.844933245] [text_chat_node]: robot response: "action": ['finishtask()'], "response": 我已经完成了全部任务，如果你还有其他需要帮忙的，请随时告诉我哦！
□
```

## 4. Source code analysis

The source code is located at:

jetson orin nano, jetson orin NX host:

```
/home/jetson/M3Pro_ws/src/largemodel/largemodel/src/largemodel/largemodel/action_service.py
```

Jetson Nano, Raspberry Pi host, you need to enter Docker first:

```
root/M3Pro_ws/src/largemodel/largemodel/src/largemodel/largemodel/action_service
.py
```

## 4.1 Case 1

action\_service.py program:

In Case 1, the **seewhat**, **navigation**, **load\_target\_points**, and **get\_current\_pose** methods in the **CustomActionServer** class are used. **Seewhat** has been explained in the **Multimodal Visual Understanding + Robotic Arm Gripping** section. This section explains the two new functions: **navigation**, **load\_target\_points**, and **get\_current\_pose**.

In the **init\_ros\_communication** initialization function, a nav2 navigation client is created to request the ros2 navigation action server for subsequent sending of navigation target point requests; a TF listener is created to monitor the coordinate transformation between map and basefootprint.

```
#Create a navigation function client and request the navigation action server
self . navclient = ActionClient ( self , NavigateToPose , 'navigate_to_pose' )
#Create tf listener to monitor coordinate transformation
self . tf_buffer = Buffer ()
self . tf_listener = TransformListener ( self . tf_buffer , self )
```

The **load\_target\_points** function is responsible for loading the target point coordinates in the map mapping file `map_mapping.yaml` and creating a navigation dictionary to store characters and corresponding map coordinates, where each point coordinate is `PoseStamped` type data.

```
def load_target_points ( self ):
    with open ( self . map_mapping_config , 'r' ) as file :
        target_points = yaml . safe_load ( file )
    self . navpose_dict = {}
    for name , data in target_points . items ():
        pose = PoseStamped ()
        pose . header . frame_id = 'map'
        pose . pose . position . x = data [ 'position' ] [ 'x' ]
        pose . pose . position . y = data [ 'position' ] [ 'y' ]
        pose . pose . orientation . x = data [ 'orientation' ] [ 'x' ]
        pose . pose . orientation . y = data [ 'orientation' ] [ 'y' ]
        pose . pose . orientation . z = data [ 'orientation' ] [ 'z' ]
        pose . pose . orientation . w = data [ 'orientation' ] [ 'w' ]
        self . navpose_dict [ name ] = pose
```

**Navigation** function: Receives a character parameter (corresponding to the character in the above map mapping), parses the coordinates of the corresponding character from the dictionary, and uses the `self.navclient` navigation client object to request the navigation action server of ros2. When the navigation action server returns a value of 4, it means that the navigation is successful, and other values represent failures (possibly due to obstacles, planning failures, etc.). After the navigation is completed, the results of the action execution are fed back to the large model.

```
def navigation ( self , point_name ):
```

```

#Get the target point coordinates
if point_name == "zero" :
    target_pose = self . record_pose
else :
    target_pose = self . navpose_dict . get ( point_name )

goal_msg = NavigateToPose . Goal ()
goal_msg . pose = target_pose
send_goal_future = self . navclient . send_goal_async ( goal_msg )
rclpy . spin_until_future_complete ( self , send_goal_future )

goal_handle = send_goal_future . result ()
if not goal_handle . accepted :
    self . get_logger () . info ( 'navigation Goal rejected' )
    return False

self . get_logger () . info ( 'navigation Goal accepted' )
get_result_future = goal_handle . get_result_async ()
rclpy . spin_until_future_complete ( self , get_result_future )

result = get_result_future . result ()
if result . status == 4 : # 4 indicates successful navigation
    self . get_logger () . info ( 'Navigation succeeded!' )
    self . action_status_pub ( f'Robot feedback: Execution
navigation({point_name}) completed' )
else :
    self . get_logger () . info ( f'Navigation failed with status:
{result.status}' )
    self . action_status_pub ( f'Robot feedback: Failed to execute
navigation({point_name})' )

```

The **get\_current\_pose** function is used to record the robot's current map coordinates in the global coordinate system map and put the coordinates into the dictionary for subsequent retrieval.

```

def get_current_pose ( self ): #Record the current coordinates in record_pose
    '''
    Get the current position in the global map coordinate system
    '''
    # Get the current target point coordinates
    transform = self . tf_buffer . lookup_transform (
        'map' ,
        'base_footprint' ,
        rclpy . time . Time () )
    # Extract position and posture
    pose = PoseStamped ()
    pose . header . frame_id = 'map'
    pose . pose . position . x = transform . transform . translation . x
    pose . pose . position . y = transform . transform . translation . y
    pose . pose . position . z = 0.0
    pose . pose . orientation = transform . transform . rotation
    self . navpose_dict [ 'zero' ] = pose
    # Print the recorded coordinates
    position = pose . pose . position
    orientation = pose . pose . orientation

```

```
self . get_logger (). info ( f'Recorded Pose - Position: x={position.x}, y={position.y}, z={position.z}' )  
self . get_logger (). info ( f'Recorded Pose - Orientation: x={orientation.x}, y={orientation.y}, z={orientation.z}, w={orientation.w}' )  
self . action_status_pub ( f'Robot feedback: get_current_pose() success' )
```