

Advanced Software Development for Robotics Assignment set 1 — ROS2

Group10: Tongli Zhu(s3159396), Thijs Hof (s3099571)

Statement regarding use of AI tools

This report was initially drafted by Microsoft Word and then converted into a LaTeX formatted document. During the conversion process, ChatGPT was utilized to transcribe mathematical formulas to meet the coding requirements of the LaTeX format.

Assignment 1.1: Image processing with ROS2

1.1.1 Camera input with standard ROS2 tools

1.1.1.1 rqt graph RosGraph

Use the specific image_tools ROS2 package (Appendix I), and the standard showimage node of the standard image_tools ROS2 package to capture webcam footage, publish to a channel and view the channel as figure 1.



Figure 1: rqt graph RosGraph

1.1.1.2 Show that the system works

1. Open the first terminal:

```
%cd ASDFR/cam2image_ws  
%colcon build  
%. install/setup.bash  
ros2 run image_tools_sdfc cam2image --ros-args --params-file cam2image.yaml
```

2. In the second terminal:

```
ros2 run image_tools showimage
```

3. In the third terminal, start `rqt_graph` to visualize the ROS graph:

```
rqt_graph
```

4. Return to the first terminal and run the `cam2image` node with the additional Quality of Service parameters:

```
ros2 run image_tools_sdfv cam2image --ros-args --params-file cam2image.yaml --ros-args
-p depth:=1 -p history:=keep_last
```

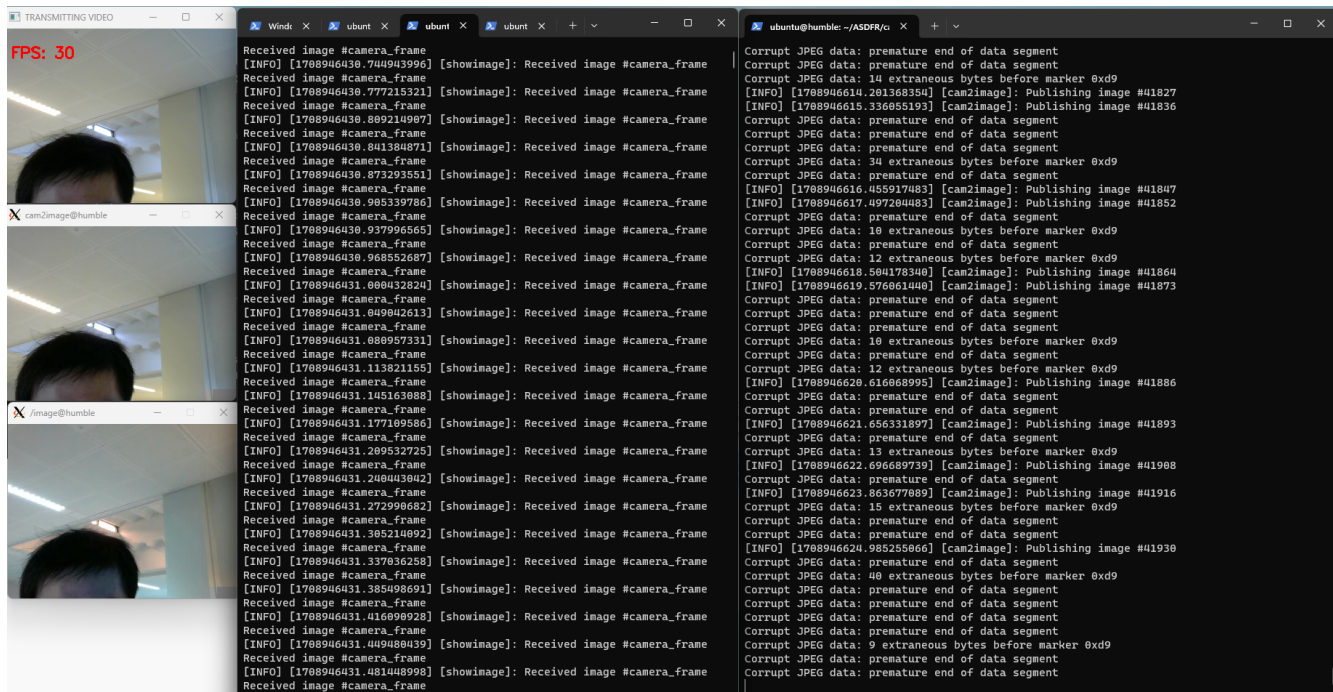


Figure 2: The screenshot for how system works(1.1.1)

1.1.1.3 Questions

1. Describe in your own words what the parameters `depth` and `history` do and how they might influence the responsiveness of the system. Are the given parameter values good choices for usage in a sequence controller? Motivate your answer.

The parameters `depth` and `history` are part of the message-passing mechanism in ROS and are used to define how the message queue is processed.

- (a) **History:** defining which messages should be retained when the `depth` limit is reached. The `keep_last` option means to keep only the most recent `N` messages, where `N` is defined by the `depth` value. In addition, there are `keep_first` (opposite of `keep_last`), `keep_all` and other options.
- (b) **Depth:** specifying the upper limit of the number of messages that the queue can store. The given parameter is set to 1, which means that the queue will only hold the latest message. If a new message arrives, it will replace the old message. When `keep_last` is used in combination with `depth:=1`, it will ensure that the queue always contains only the latest message.

For a sequential controller, the usual goal is to make decisions quickly based on the latest information. For example, to adjust the robot's movement direction in real-time, using `depth:=1` and `history=keep.last` is a good choice. After this configuration, the sequence controller will always respond based on the latest data, thereby improving the real-time performance and responsiveness of the system.

2. **What is the frame rate of the generated messages? How can you influence the frame rate? Is there a maximum? What determines the maximum?**

The frame rate of message generation refers to the number of image messages published per unit time, usually measured in frames per second (FPS). In the assignment, we got the result through `ros2 topic hz /image`. The average rate: 24.128, which indicates that the image messages published to the `/image` topic have an average of 24.128 frames per second. Several aspects that affect frame rate:

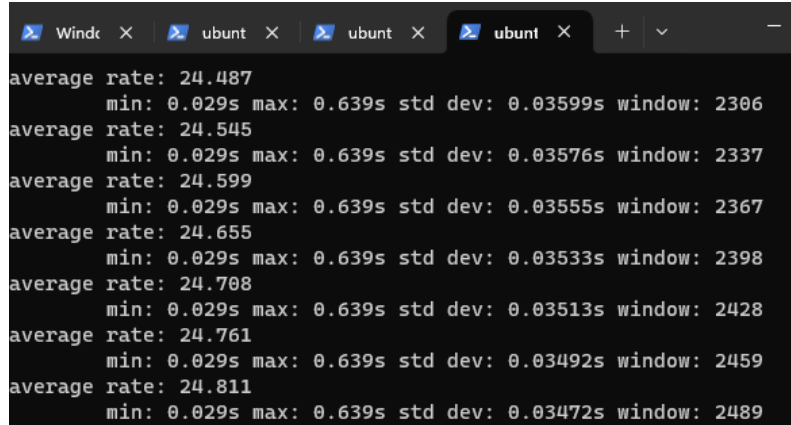


Figure 3: The frame rate

- The publisher's sending frequency: In the code, we can control how often messages are published by setting a timer in the node.
- System resources: Because ROS nodes run in virtual machines, not only the resources allocated to the virtual machine, but also the CPU and memory of the host will also affect the achievable frame rate. The higher the resource usage, the lower the actual frame rate will be.
- Network quality: The bandwidth and latency of the network can also limit frame rates.
- Camera hardware capabilities: A camera's hardware specifications directly determine how fast it can capture images. For example, a camera might have a maximum capture capability of 30 FPS. Even if system resources and network conditions allow for higher frame rates, the camera itself cannot provide frame rates beyond its hardware limitations.

The maximum frame rate that can be achieved is limited by the weakest of the above factors. No matter how ideal other conditions may be, the overall performance of the system will be determined by this most limiting factor. This means that any performance bottleneck in any part of the system will directly affect the maximum frame rate that the entire system can achieve.

1.1.2 Adding a brightness node(show the system works)

1.1.2.1 Design Steps

- The node subscribes to a topic named `image` and receives messages of type `sensor_msgs::msg::Image`.
- Within the callback of the subscriber, the incoming ROS image message is converted into the OpenCV format, `cv::Mat`, using the `cv_bridge` package. This package provides an interface between ROS image messages and OpenCV images.
- The image frame is then converted to a grayscale image using the OpenCV function `cvtColor`. This conversion to grayscale simplifies the computation of brightness, as each pixel in a grayscale image corresponds to the brightness level at that point.

4. After calculating the average brightness of the grayscale image, the node compares this value with a predefined brightness threshold, which is set by default to 127.5.
5. Depending on whether the average brightness is below or above the threshold, the node publishes a message of type `std_msgs::msg::String` to a topic named `brightness_topic`. If the average brightness is below the threshold, the message contains "It is dark"; if above, the message contains "Light is ON".

Thus, based on the average brightness of the image, the Brightness Node can determine and communicate whether the environment is bright or dark.

1.1.2.2 rqt graph



Figure 4: rqt graph(brightness node and topic)

1.1.2.3 Exception debugging

After compilation, the code did not report an error, and there were no exceptions after running. However, when viewing rqt, `brightness_topic` could not be found. We spent a long time checking the code, and finally found that the code was correct, but we did not use the `echo`. Although the node published the message to the topic, it would not appear as long as there was no `echo`.

1.1.3 Adding ROS2 parameters

1.1.3.1 Design Steps

1. The node declares a new parameter named `brightness_threshold` with a default value of 127.5. This is done in the constructor of the node using the `declare_parameter` function provided by the ROS2 `rclcpp` API.
2. When the node starts, it retrieves the value of the `brightness_threshold` parameter using the `get_parameter` function. This allows the node to use the latest value set for this parameter.
3. To start the node with a custom brightness threshold from the command line, the following syntax is used:

```
ros2 run brightness brightness_node --ros-args -p brightness_threshold:=<value>
```

4. During runtime, the parameter can be adjusted without restarting the node by using the following command:

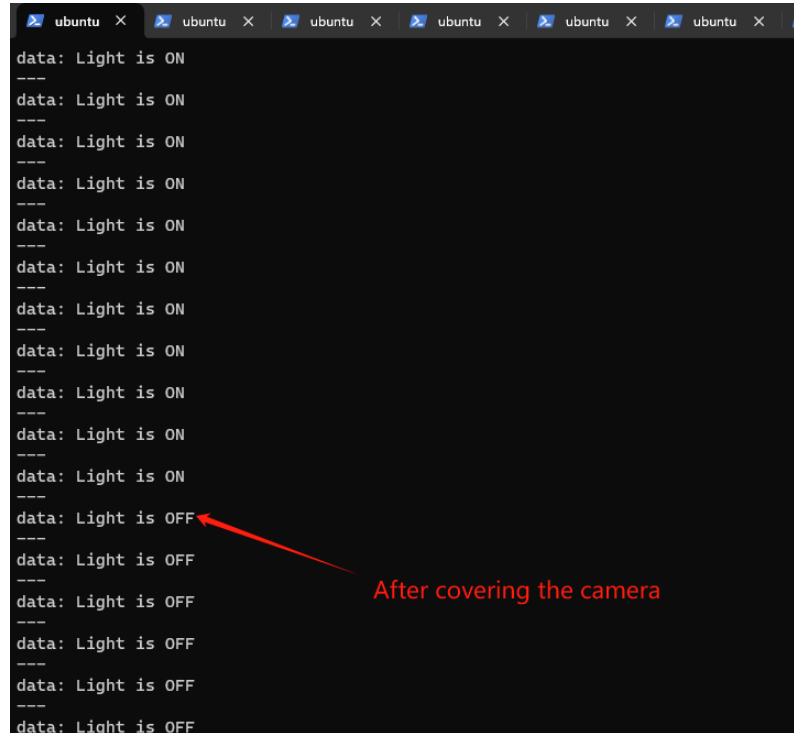
```
ros2 param set /brightness_node brightness_threshold <value>
```

5. The node's subscription callback function, which processes the incoming images, uses the current value of the `brightness_threshold` parameter each time it is called.
6. To demonstrate that the system works with the ROS2 parameters, tests are conducted by varying the `brightness_threshold` parameter and observing the output messages. The node should consistently publish "Light is ON" or "Light is OFF" based on the current threshold and the image's average brightness.

1.1.3.2 Screen shots

```
ubuntu@humble:~/ASDFR/cam2image_ws$ ros2 param set /brightness_node brightness_threshold 10.0
Set parameter successful
ubuntu@humble:~/ASDFR/cam2image_ws$ ros2 param set /brightness_node brightness_threshold 200.0
Set parameter successful
ubuntu@humble:~/ASDFR/cam2image_ws$ |
```

Figure 5: Configurability during run time



```
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is ON
---
data: Light is OFF
---
data: Light is OFF
---
data: Light is OFF
---
data: Light is OFF
---
data: Light is OFF
---
data: Light is OFF
---
data: Light is OFF
```

After covering the camera

Figure 6: The result

1.1.4 Simple light position indicator

1.1.4.1 Design Steps

1. Node Initialization:

- Create a node named "lp_indicator_node".

2. Parameter Handling:

- Declare and retrieve a parameter named "brightness_threshold" for setting the brightness threshold, with a default value of 130.0.
- Register a parameter callback to update the `brightness_threshold_` member variable and log the change when the "brightness_threshold" parameter is updated.

3. Subscription Handling:

- Create a subscriber `image_subscription_` to subscribe to the "image" topic, receiving messages of type `sensor_msgs::msg::Image` and specifying `image_callback` as the callback function.

4. Publishing Handling:

- Create a publisher `position_publisher_` to publish to the "light_position_topic" topic, with message type `std_msgs::msg::String`, to publish the position of the light source.
- Create another publisher `processed_image_publisher_` to publish to the "processed_image_topic" topic, with message type `sensor_msgs::msg::Image`, to publish the processed image.

5. Image Processing Callback (`image_callback`):

- Use `cv_bridge` to convert the received ROS image message to an OpenCV image (grayscale).
- Apply thresholding to generate a binary image `binary_image`.
- Compute the center of gravity of the white pixels in the binary image.
- Draw a circle at the center point in the binary image.
- Convert the processed image back to a ROS message format and publish it using `processed_image_publisher_`.

6. Publishing Light Source Position:

- Create a string message containing the light source position information and publish it using `position_publisher_`.

Through these steps, the `LPIndicatorNode` is capable of receiving image data, processing the image to locate the bright spot, and then publishing both the processed image and the location of the bright spot. This node serves as both a subscriber (receiving image data) and a publisher (publishing the light source position and the processed image).

1.1.4.2 rqt graph

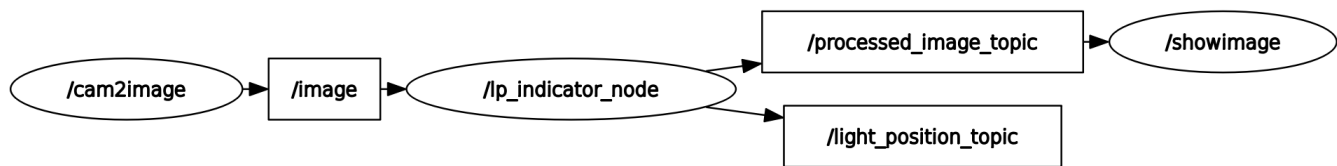


Figure 7: rqt graph

1.1.4.3 Use and Test

To use and test the `LPIndicatorNode`, follow these three steps:

1. **Displaying the Processed Image:** Utilize the `image_tools` package to display the processed image by running the command:

```
ros2 run image_tools showimage --ros-args -r image:=/processed_image_topic
```

This command subscribes to the `/processed_image_topic` topic and displays the images using the `showimage` node.

2. **Monitoring the Light Position:** To observe the position of the light source, execute the following command in a terminal:

```
ros2 topic echo /light_position_topic
```

This command will display the light position updates published to the `/light_position_topic` in real-time on the terminal.

3. **Adjusting the Brightness Threshold:** To test the effect of different brightness threshold values, use the `ros2 param set` command. For example, to change the threshold, execute:

```
ros2 param set /lp_indicator_node brightness_threshold <value>
```

Adjusting this parameter to observe how the node's behavior changes with different threshold settings.

These steps provide a method to test and demonstrate the functionality of the `LPIndicatorNode`, enabling the visualization of the processed image and the dynamic adjustment of parameters to affect the node's processing behavior.

1.1.4.4 Screen shots

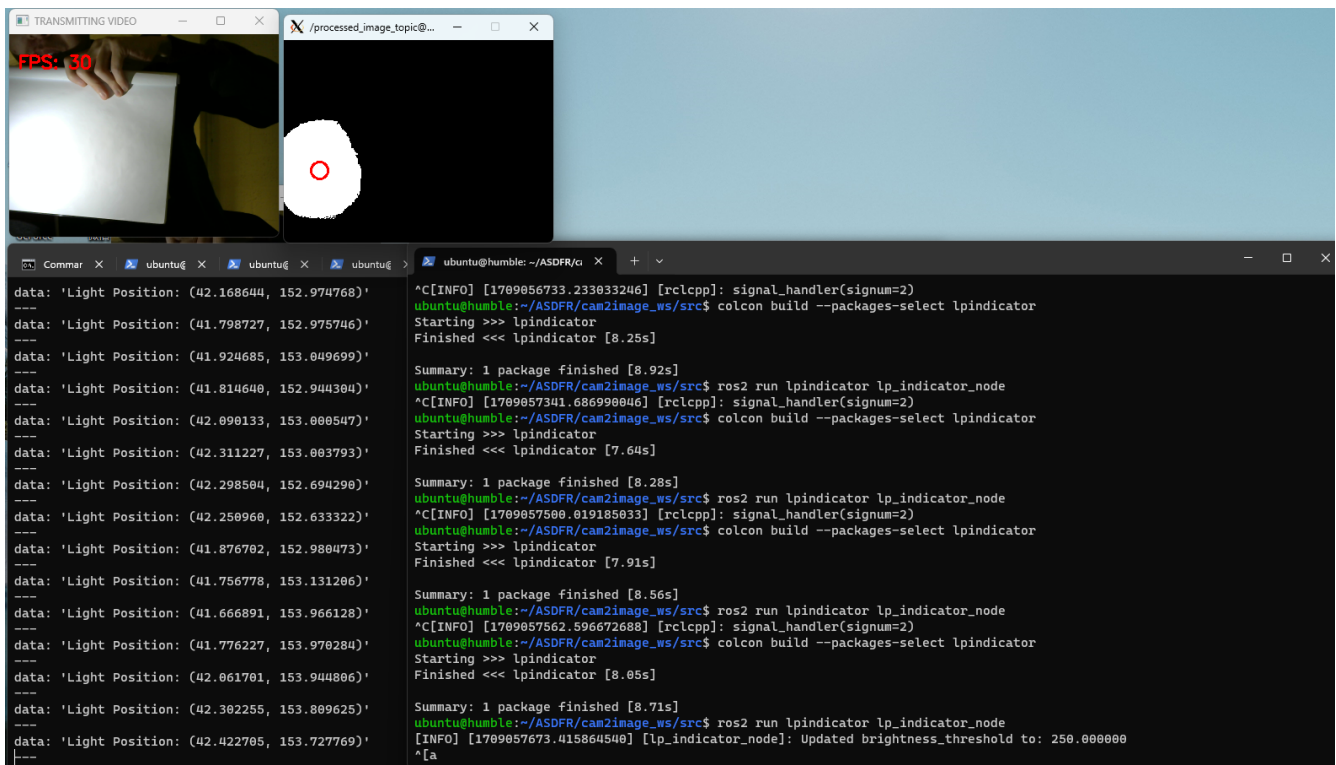


Figure 8: The result

Assignment 1.2: Sequence controller

1.2.1 Unit test the sequence controller using the RELbot Simulator

1.2.1.1 Show the system works:

We consider tasks 1.2.1 and 1.2.2 as a whole for the design process. This section details the operation and interaction of the nodes within the system, focusing on the WheelControlNode's behavior and its interaction with other components through a ROS 2 launch file.

(1) General Design Elements:

• Initialization and Node Setup:

- The node retrieves initial motor speeds from parameter server, initializing the motors with `initial_left_motor_vel` and `initial_right_motor_vel`.
- It subscribes to `light_position_data_topic` and `output/camera_position` to receive updates on the light source and camera positions.
- Two publishers, `left_motor_vel_pub_` and `right_motor_vel_pub_`, are established to publish speeds to the left and right motors.

• Subscriber Setup:

- `light_pose_sub_` receives light position updates and processes them in `light_pose_callback`.
- `camera_pose_sub_` receives camera position updates and processes them in `camera_pose_callback`.

• Publisher Setup:

- The node controls the motors via `left_motor_vel_pub_` and `right_motor_vel_pub_` by publishing the calculated speeds.

(2) Specialized Design for Preset Speed Sequence (1.2.1):

- **Speed Setpoints:** Predefined speed values in `speed_sequence` dictate the motors' speeds, guiding the robot's movement pattern. Specifically, the speed sequence is set as `{0.0, 1.0, 2.0, 3.0}`, which means the motor speeds will iterate through these values in sequence.
- **Timer:** A timer is configured to periodically invoke `generate_and_publish_setpoint`, which fetches the current speed setpoint for the motors from `speed_sequence`. The timer is set to trigger every 5000 milliseconds (5 seconds), ensuring that the speed setpoint is updated at this interval.

(3) Detailed Specialized Design for Light Following (1.2.2):

- **Light Position Callback:** `light_pose_callback` updates `light_pose_` and sets `light_pose_received_` to true upon receiving light source data.
- **Camera Position Callback and Proportional Control Strategy:**
 - `camera_pose_callback` is invoked upon camera position updates.
 - If `light_pose_received_` is true, it calculates the x-axis difference between light source and camera, adjusting motor speeds with a proportional control strategy, where left motor speed is $-k_p \times x_difference$ and right motor speed is $k_p \times x_difference$.

(4) Launch File Configuration:

The launch file manages several nodes for the system's operation:

- **cam2image:** From the `image_tools_sdf` package, provides camera image data.
- **lp_indicator_node:** Manages the light position indicator node.
- **showimage:** From the `image_tools` package, displays the processed image.
- **RELbot_simulator:** Initiates the RELbot simulator.

- **wheel_control_node:** Activates the WheelControlNode, with parameters to set initial motor speeds.

(5) Mode Switching via Launch File:

- **To switch to Preset Speed Sequence (1.2.1):** Commenting out `lp_indicator_node` in the launch file ceases light position updates, making the robot follow the preset speed sequence.
- **To switch to Light Following (1.2.2):** Uncommenting `lp_indicator_node` enables light position updates, allowing the robot to follow the light source.

(6) rqt graph:

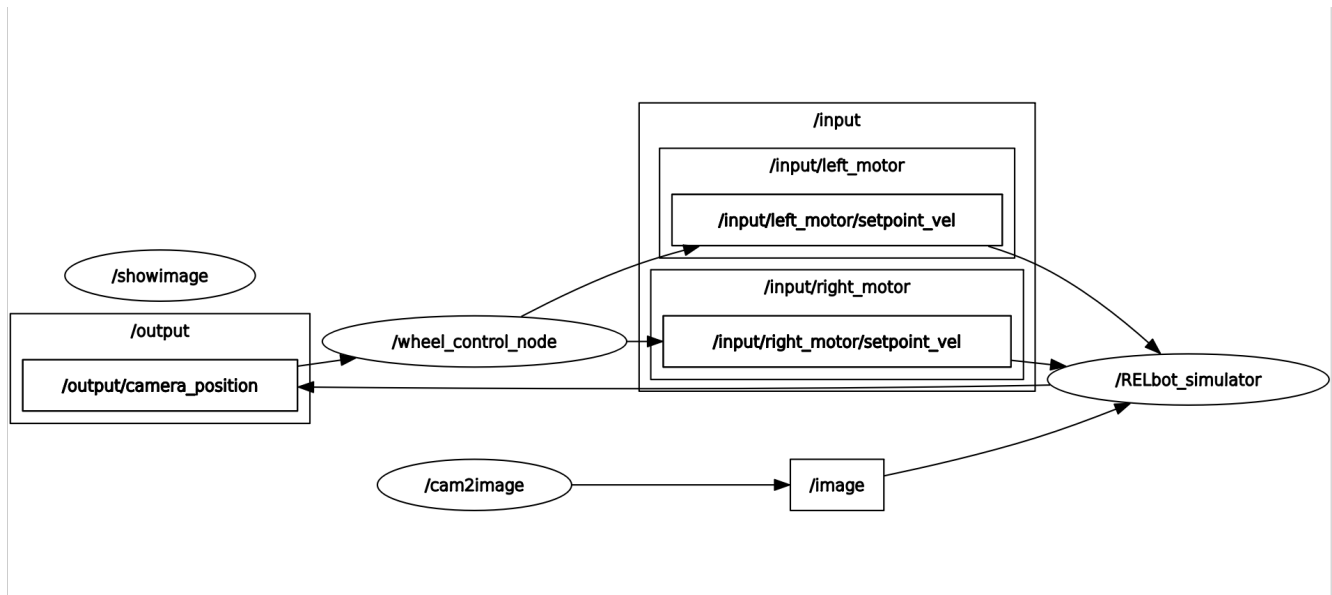


Figure 9: rqt graph

(7) The results: We can see from the moving camera that when the speed in the setpoint is 0, the size of the spray bottle displayed is consistent with the reference. When the speed is positive, the spray bottle appears to be enlarged, simulating the forward progress of RELbot_simulator.

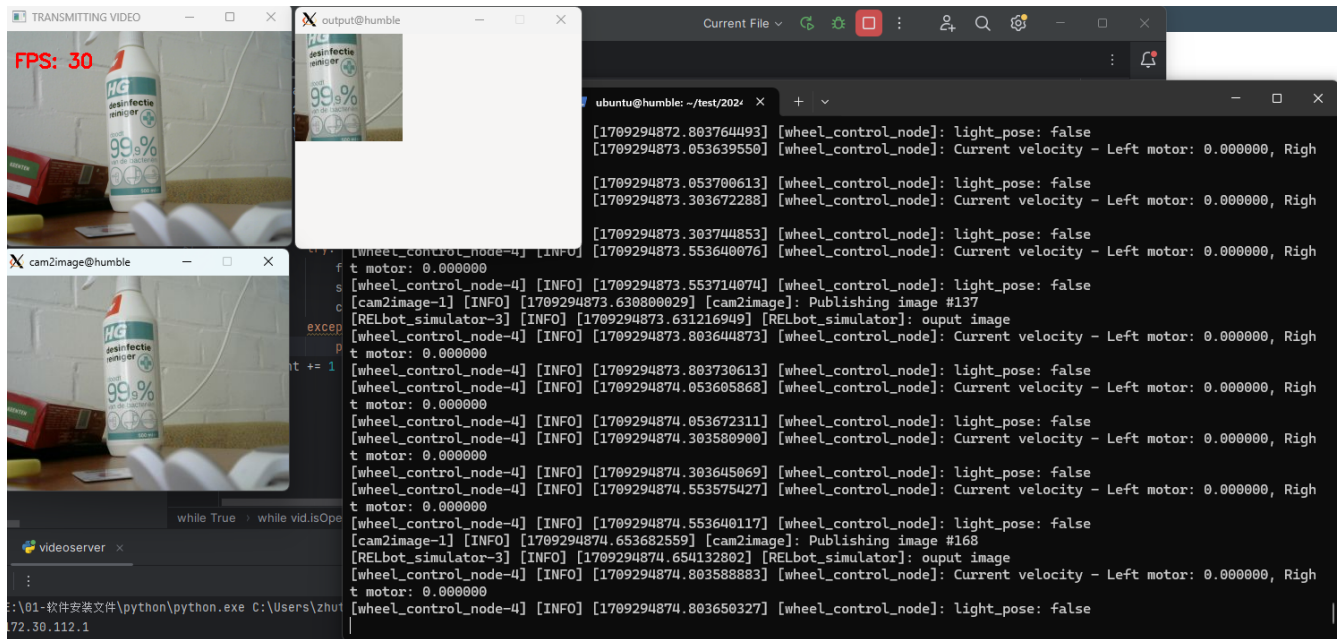


Figure 10: The camera image of RELbot_simulator when the speed is 0

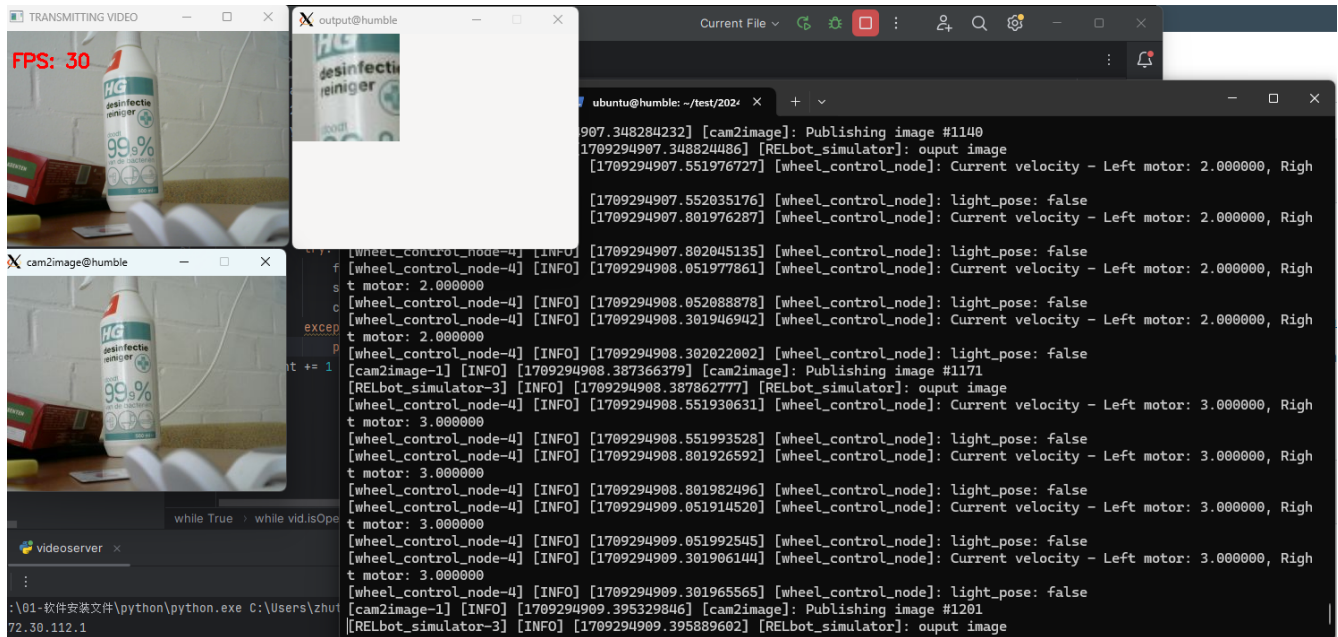


Figure 11: The camera image of RELbot_simulator when the speed is positive

1.2.1.2 Questions

Plot the setpoint and actual position (use of rqt is ok). Explain the results. Don't forget to look at the time constant of the first order system.

Questions

In the rqt_plot chart, the red line represents the setpoints for the speed of the left and right motors. They are presented as a polyline, reflecting sudden changes in speed at specific time points in every 10 seconds. However, it's important to note that although these polylines appear to be continuous changes, they are actually connecting discrete time points — for example, (5,0), (10,1), (20,2), (30,3), etc. This polyline is essentially the result of rqt_plot automatically linking our discrete data points.

The cyan line represents the actual position of RELbot, which corresponds to its movement along the x-axis. In an ideal environment, when the speed jumps to 1 rad/s at 10s, we expect to see a diagonal line from that point, indicating that the robot starts moving at a constant speed. However, the chart does not show a straight line. This discrepancy is where the time constant becomes significant. In our simulator, there is a setting that simulates motor speed behavior as a first-order system dynamic. This means that the speed will not jump instantaneously but will gradually approach the new setpoint. The time constant is the time required for the system to reach 63.2% of its final steady-state value, which is closer to reality.

This effect is particularly evident at 25s. Despite the speed instruction suddenly dropping from 3 rad/s to 0 rad/s, the cyan position line does not immediately reflect this change. Instead, it continues to rise for a while, then gradually slows down and eventually stops. We added a vertical green line as a reference, making this phenomenon more obvious by increasing the `TIME_CONSTANT` in `DynamicsSimulation.hpp` from 0.3 to 5.

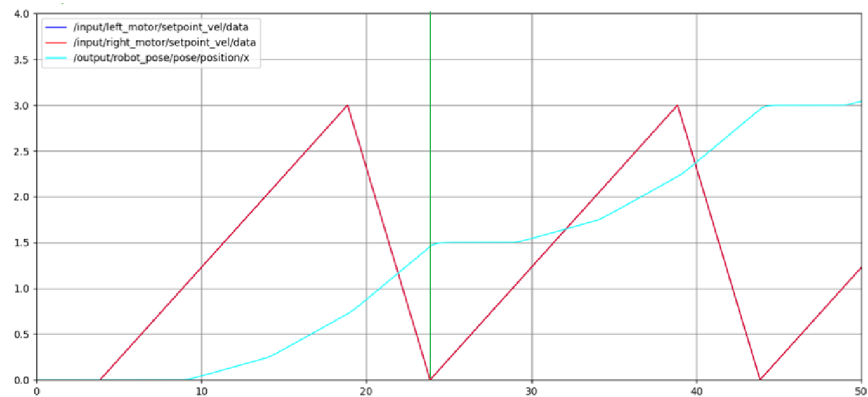


Figure 12: The setpoint and actual position(TIME CONSTANT = 0.3)

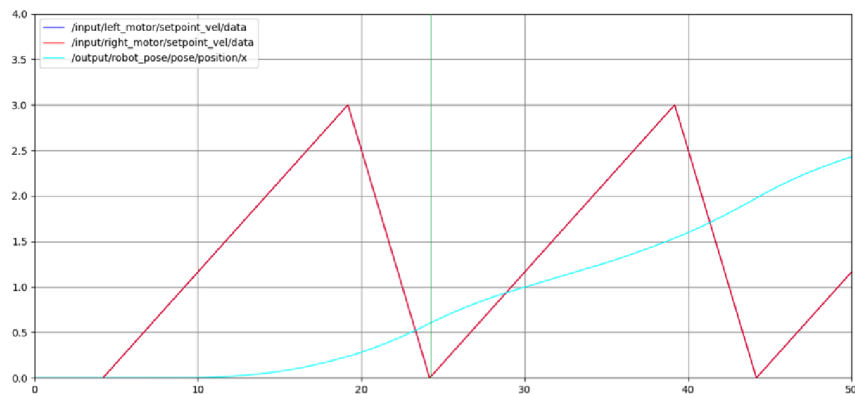


Figure 13: The setpoint and actual position(TIME CONSTANT = 5)

1.2.2 Integration of image processing and RELbot Simulator

Show the system works:

(1) Design steps:

Please refer to 1.2.1.

(2) rqt graph:

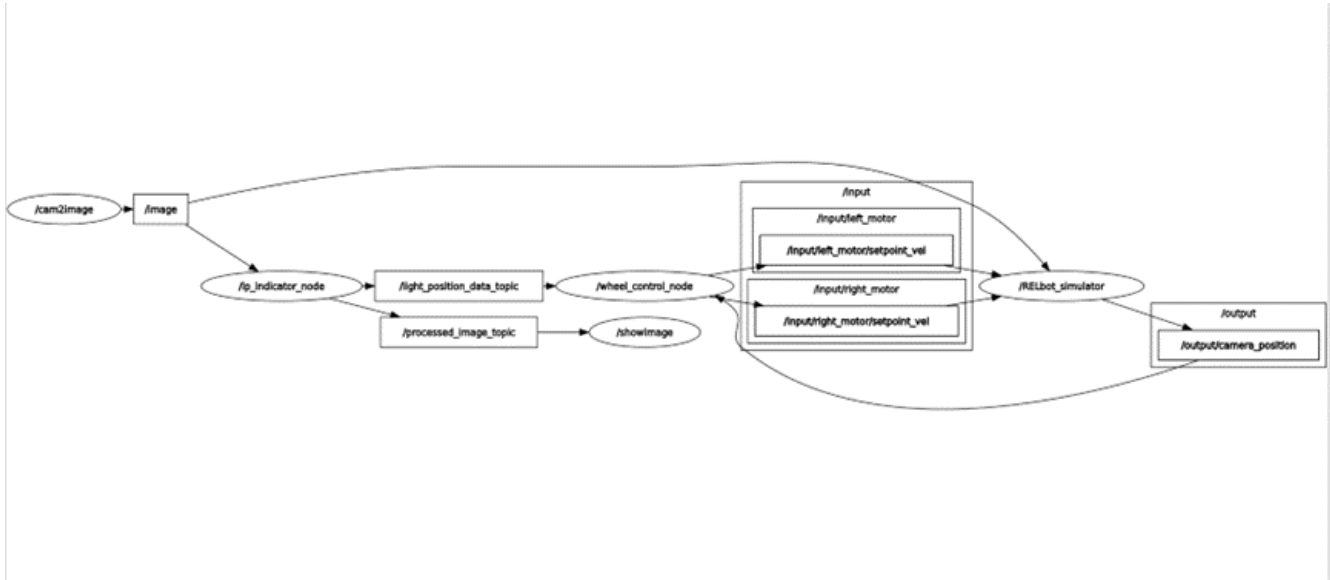


Figure 14: rqt graph

(3) The result:

As can be seen from the screenshot, under the same timestamp, the camera position on the right side will follow the light point position on the left side, which proves the effectiveness of the program.

The figure displays four terminal windows arranged in a 2x2 grid, showing the output of a ROS2 program. Each window has a title bar with the user 'ubuntu@humble' and the current directory. The left column of windows shows data from the home directory (~), while the right column shows data from the directory ~/ASDFR/GI.

Top Left Window (~/): Displays three point cloud messages. Each message includes a 'point' field with x, y, and z coordinates, and a 'header' field with 'stamp' (sec and nanosec) and 'frame_id'.

```

x: 78.13736644928542
y: 110.70223393922575
z: 0.0
---
header:
  stamp:
    sec: 1709142596
    nanosec: 926588530
  frame_id: ''
point:
  x: 78.13736644928542
  y: 110.70223393922575
  z: 0.0
---
header:
  stamp:
    sec: 1709142596
    nanosec: 977150954
  frame_id: ''
point:
  x: 78.43625442869796
  y: 110.52956155890168
  z: 0.0
---
header:
  stamp:
    sec: 1709142597
    nanosec: 7372246
  frame_id: ''
point:

```

Top Right Window (~/ASDFR/GI): Displays three point cloud messages, similar in structure to the left window but with different coordinate values.

```

  frame_id: ''
point:
  x: 79.0
  y: 120.0
  z: 0.0
---
header:
  stamp:
    sec: 1709142596
    nanosec: 928503379
  frame_id: ''
point:
  x: 79.0
  y: 120.0
  z: 0.0
---
header:
  stamp:
    sec: 1709142596
    nanosec: 978870192
  frame_id: ''
point:
  x: 79.0
  y: 120.0
  z: 0.0
---
header:
  stamp:
    sec: 1709142597
    nanosec: 8348453

```

Bottom Left Window (~/): Displays three point cloud messages with different coordinates.

```

header:
  stamp:
    sec: 1709142580
    nanosec: 689797141
  frame_id: ''
point:
  x: 236.65088622069754
  y: 77.09746521821994
  z: 0.0
---
header:
  stamp:
    sec: 1709142580
    nanosec: 720548985
  frame_id: ''
point:
  x: 236.65088622069754
  y: 77.09746521821994
  z: 0.0
---
header:
  stamp:
    sec: 1709142580
    nanosec: 769279713
  frame_id: ''
point:
  x: 236.50193301435408
  y: 76.61990430622009
  z: 0.0
---

```

Bottom Right Window (~/ASDFR/GI): Displays three point cloud messages with different coordinates.

```

---
header:
  stamp:
    sec: 1709142580
    nanosec: 659606557
  frame_id: ''
point:
  x: 229.0
  y: 120.0
  z: 0.0
---
header:
  stamp:
    sec: 1709142580
    nanosec: 691684173
  frame_id: ''
point:
  x: 229.0
  y: 120.0
  z: 0.0
---
header:
  stamp:
    sec: 1709142580
    nanosec: 721661477
  frame_id: ''
point:
  x: 229.0
  y: 120.0
  z: 0.0

```

Figure 15: The results for 1.2

Questions

1. Is the system you created an open-loop or closed-loop system (why)?

The system we created is a closed loop control system. In a closed-loop system, the output of the system (in this case the motion of the RELbot) is adjusted based on the input (the position of the light source) and the current state (the position detected by the camera).

2. Is ROS2 (which is a non-realtime platform) in general, appropriate for this type of systems? Why? Are there any limitations (i.e., when do things go wrong)?

ROS2 is generally suitable for such systems for the following reasons:

- **Modularity and Flexibility:** ROS2 supports a modular architecture, allowing developers to create reusable and configurable nodes, which is ideal for various robotic systems.
- **Communication Mechanisms:** ROS2 offers robust communication mechanisms including topics, services, and actions, facilitating effective data exchange between nodes.

However, the limitations of ROS2 are primarily:

- **Non-Realtime Nature:** ROS2 cannot guarantee strict execution or response times, which might affect applications that require high real-time performance.
- **Resource Competition:** In systems with limited resources, multiple processes and nodes might compete for CPU and memory, potentially leading to performance bottlenecks.

In summary, this approach is appropriate unless strict timing control with high rate and low latency is required.