Robotics

2023 / 2024

Elvi Mihai Sabau Sabau

Yassine Oueslati

Pietro Miotto

May 29, 2024

Contents

List of Figures

1 Project Overview

For our final project, as the third assignment on this subject, we decided to make something fun with the robomaster, in this case we tought it would be cool to create the logic to control de robomaster s1 using only body gestures, in a similar fashion that a rc car is controlled.

1.1 Objectives

The objectives for our project are as follows:

- Control the Robomaster S1 camera's gimbal.
- Control the Robomaster S1 movement wheels.
- Detect the user and keep track of the user with the camera while the robomaster is moving.
- Identify different body gestures of the user.

With the following implementations, we should be able to implement our idea and make the robomaster move on its own based on what the camera feed is tracking from the users body gestures.

1.2 Resources

For this project we used the following libraries and tools:

- **Mediapipe**: We used the mediapipe python library to detect the body gestures, specifically the shoulders and the wrists.
- **OpenCV**: We used OpenCV2 to handle the image processing part, we also used ros2 opencv bridge to manipulate the image from the Image msg from Ros.
- **Robomaster S1**: We also use the Robomaster S1 as the robo-cart for this project.

1.3 Architecture

For the system that we want to implement, we thought of splitting it in 3 nodes (processes).

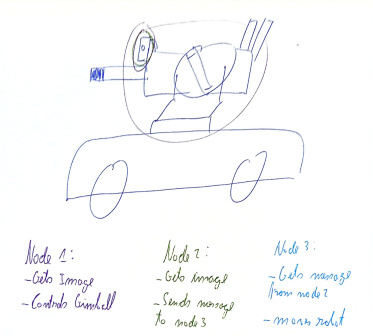

Figure 1: Simple Diagram of the logic distribution between the 3 nodes.

- The first node manages the gimball and tracks the user.
- The second detects the user's gesture and sends a command representing a specific gesture to the third node.
- And the third node moves the robomaster accordingly to the message received from the previous node.

1.4 Methodology

We decided to split the work one node per person since were a 3 person group, where each person tackles a specific part of the system.

Initially, since to develop the logic for each node we didn't use ROS directly, we started developing tiny scripts interacting with opencv2 and mediapipe, creating the base cases for the gestures.

Figure 2: Video capture of the initial gesture tracking script devleoped in Python.

And after having something working, we migrated the python code to ROS2, and integrated the robot services into our code.

2 System Nodes

In this section we describe what specific logic we implemented on each node.

2.1 Node 1: User Tracking

- `Purpose`: This node is designed to control the gimbal movement to locate and track a human using the camera feed as input.
- `__init__(self)`:
  - `Description`: Constructor method for the PrimaryController class.
  - `Functionality`: Initializes the node, subscribes to image and joint state topics, creates a publisher for gimbal commands, and initializes MediaPipe pose solution.
- `gimbal_call(self, msg)`:
  - `Description`: Callback function for processing joint state messages.
  - `Functionality`: Extracts yaw position of the gimbal from the joint state message and updates the `gimball_rads` attribute so that we can use it later to know of the joint reached the limit or not.
- `search_human(self)`:
  - `Description`: Method for searching for a human based on gimbal position.
  - `Functionality`: Turn full right and scan the environment to find a human and if you reached the limit of the joint start turning left.
- `image_callback(self, msg)`:
  - `Description`: Callback function for processing image messages.
  - `Functionality`: Converts ROS image message to OpenCV image, processes the image using MediaPipe pose solution, extracts pose landmarks, determines human position, and adjusts gimbal movement accordingly.
- `get_values(self, landmarks, width)`:
  - `Description`: Method for extracting vertical and horizontal positions from pose landmarks.
  - `Functionality`: Calculates the average vertical and horizontal positions of the shoulders based on pose landmarks using the midpoint formula $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$.
- `get_position(self, vertical_position, horizontal_position, epsilon)`:
  - `Description`: Method for determining the position (up, down, left, right, or center) based on vertical and horizontal positions.
  - `Functionality`: Compares the shoulder positions with a center point to determine the direction (up, down, left, right, or center) with a specified epsilon error.
- `move(self, yaw=0.0, pitch=0.0)`:
  - `Description`: Method for publishing gimbal commands.
  - `Functionality`: Publishes gimbal command messages with specified yaw and pitch speeds to control gimbal movement.
  - `Example`: If the human is right Then Move right until the pos of the human becomes center

2.2 Node 2: Body Gestures

This node focuses on the deployment of a body landmark detection system, which takes as input the video flow from the topic `/camera/image_color` and publishes the inferred gestures to the `/cmd_command` topic. To detect body landmarks, we implement Google's `mediapipe` library. This library is deployed as follows:

```
self.mp_pose = mp.solutions.pose
self.pose = self.mp_pose.Pose()
self.mp_drawing = mp.solutions.drawing_utils
```

We basically import the pose estimation module and the landmarks drawing modules from MediaPipe's solutions and then istantiate the `Pose()` class from mediapipe. This node is divided in two main functions, delved in the following paragraphs.

2.2.1 `classify_gesture(self, landmarks, width)` function

This function takes as input the landmarks of the person's body computed by mediapipe modules. From all the landmarks computed by mediapipe, we consider only the one related to the two shoulders (left and right) and the two wrists (left and right). We store in appropriate variables the visibility of the considered landmarks. Each of the landmarks has

an x and y value, which we use to detect the vertical and horizontal position of both wrists with repsect to the vertical and horizontal positions of the respective shoulders.To better understand, I provide an infographics which explains how the vertical and horizontal position of the writsts w.r.t. the shoulder ones are used to compute a specific state.

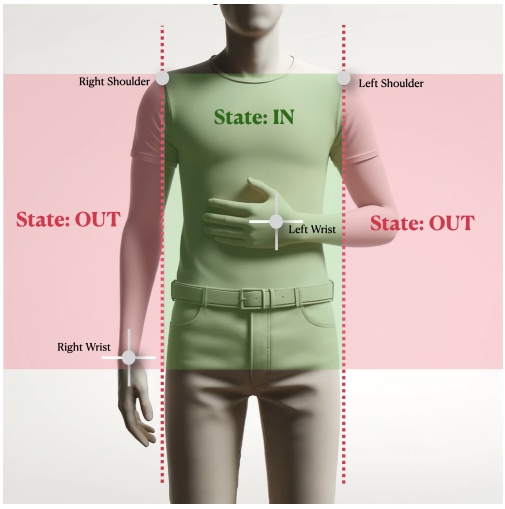

Figure 3: x values of wrists w.r.t. x values of shoulders. If the x value of the wrist is higher than the one of the shoulder the state is Outside, if the x value of the wrist is lower than the one of the shoulder the state is Outside. Consider the red lines as thresholds

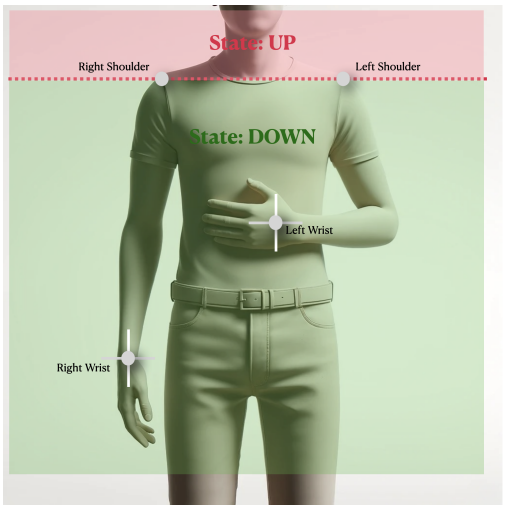

Figure 4: y values of wrists w.r.t. y values of shoulders. If the y value of the wrist is higher than the one of the shoulder the state is Up, if the y value of the wrist is lower than the one of the shoulder the state is Down. Consider the red lines as thresholds
These states, with the visibility values of both wrists and shoulders landmarks, are returned from this function, and are later used to infer the gestures.

2.2.2 `image_callback(self,msg)` function

This function is called each time a new image is published on the topic `/camera/image_color`. Note that, by default, the topic subscription is applied with a `QoS profile`(i.e., history depth) of size `10`, so that, if any new message is received from the topic while the previous one is still being elaborated, this new message is stored in a queue which can handle up to 10 entries. This is extremely useful given how time consuming image processing tasks are. The function works as follows:

- Once the image is taken from the topic, we need to process it to make it usable by the mediapipe modules. To perform this, we use a bridge for `Cv2 library`, provided by ROS documentation [1]. The bridge is istantiated and used as follows:

```
def __init__(self):
    [...]
    self.bridge = CvBridge()

def image_callback(self, msg):
    [...]
    frame = self.bridge.imgmsg_to_cv2(
        msg)
```

This is necessary because the msg of the `/camera/image_color Image()` topic is a 1D

array of bites with some metadata. By combining metadata with the 1D array of bites, this bridge/interface is able to return an image suitable for `CV2` library.

- Once the image is transformed to a valid Cv2 format, it is processed by the mediapipe modules. After that, we check if any landmark is detected. If so, we show them on the screen, if not, we write on the screen `NO POSE LANDMARKS`.
- Once landmarks are detected and plot on the frame, we call the `classify_gesture()` function, which returns the detected states for the hands positions and visibility values for each of the landmarks used to compute the states. We then check if the individual returned visibility values reach a specific treshold. If the treshold is not crossed, we plot on the screen `NOT FULL BODY VISIBILITY`
- If the treshold is crossed, we use two switches to match the states with a direction value and a mobement value. This is done as follows:

```
#direction
match (left_state, left_vertical):
    case ('Out', 'Up'):
        direction =  '1' #'LEFT'
    case ('Out', 'Down'):
        direction = '-1' #'RIGHT'
    case ('In', _):
        direction = '0' #'STOP'

#movement
match (right_state, right_vertical):
    case ('Out', 'Up'):
        movement = '1' #'FORWARD'
    case ('Out', 'Down'):
        movement = '-1' #'BACKWARD'
    case ('In', _):
        movement = '0' #'STOP'
```

We then publish to the topic `/cmd_command` a `String()` message containing `DIRECTION` and `MOVEMENT`. This message is going to be evaluated by the 3d Node to infer the movement and direction (which can be combined) for the robot.

2.3 Node 3: Movement and Control

This node's logic can be broken down into three main phases: command reception, command processing, and action execution.

2.3.1 Command Reception

The system subscribes to a command topic `cmd_command`, where it receives movement instructions as strings in format direction, movement.

Each command string contains directional and movement data as 1, 0 or -1, which represents the increase, stop or decrease in speed for direction (spin left / right) and movement (move forward or backwards), which are parsed and stored in separate arrays for movement and direction.

msg	Direction	Movement
{1,1}	left	F
{1,0}	left	STOP
{1,-1}	left	B
{0,1}	STOP	F
{0,0}	STOP	STOP
{0,-1}	STOP	B
{-1,1}	right	F
{-1,0}	right	STOP
{-1,-1}	right	B

Figure 5: Table representing all 9 posible movement commands.

2.3.2 Command Processing

The stored commands are processed at regular intervals, set to every 0.5 seconds.

During each processing cycle, the node calculates the average values of the movement and direction commands received in the previous interval. This averaging process smooths out any inconsistencies or noise in the commands, resulting in a stable and predictable set of instructions for the robot.

After calculating these averages, the command arrays are cleared to prepare for the next set of incoming commands.

2.3.3 Action Execution

The averaged commands are then translated into velocity instructions. The system scales the movement and direction commands by predefined speed

factors (velocity_speed and direction_speed) to ensure the robot moves at appropriate speeds.

```
def perform_action(self):
    """ Perform the action based on the
        movement and direction """
    # If there is no command, do nothing
    if len(self.movement) == 0
    or len(self.direction) == 0:
        self.move() # Default values are 0.0
        return

    # Get the most common movement and
        direction, only integers
    avg_mvm = np.mean(self.movement)
    avg_dir = np.mean(self.direction)

    # Reset the movement and direction
    self.movement = []
    self.direction = []

    # Perform the action
    speed = avg_mvm * self.velocity_speed
    rads = avg_dir * self.direction_speed

    self.move(speed, rads)
```

These scaled values are used to create a Twist message, which is published to the robot's velocity topic `cmd_vel`. The robot uses these messages to adjust its linear and angular velocities, thus moving according to the user's gestures.

```
def move(self, speed=0.0, rads=0.0):
    # Move the robot in a certain direction
    velocity = Twist()
    velocity.linear.x = float(speed)
    velocity.angular.z = float(rads)
    self.velocity_publisher.publish(velocity)
```

Additionally, the system includes a safety mechanism to stop the robot if no valid commands are present. If either the movement or direction arrays are empty during a processing cycle, the system sends a stop command, ensuring the robot remains stationary in the absence of new instructions.

2.4 Execution

To run the package, follow these steps:

2.5 Building the Package

First, copy the source code `usi_final` into our development folder, and then run `colcon build`.

2.6 Running the Package

To start the program, once built, source the terminal, and then in two terminals, run the following nodes:

- Start the bridge node, which allows interaction with the Robomaster from ROS2. More documentation can be found robomaster_ros docs.
- Launch the launch file in the package we developed using the command: `ros2 launch usi_final primary.launch.xml`.

3 Testing

After integrating all the nodes, we performed tests in two environments:

- Inside the simulator, using the simulated camera and Ubuntu's broken camera.
- In real life, using two different Robomasters.

3.1 Simulator

For the simulator, we used CoppeliaSim with the Robomaster S1 model.

Initially, we tried to use Bill, the human model from CoppeliaSim, and also the orange mannequin.

However, after several trials and errors, we noticed that Mediapipe doesn't detect the human model well in the simulator, so we opted for a different approach.

Using a table flipped on its long side to match the height of a human, we took pictures of ourselves performing various poses. We then printed these pictures as textures on the table for the robot to detect, and this method worked perfectly.

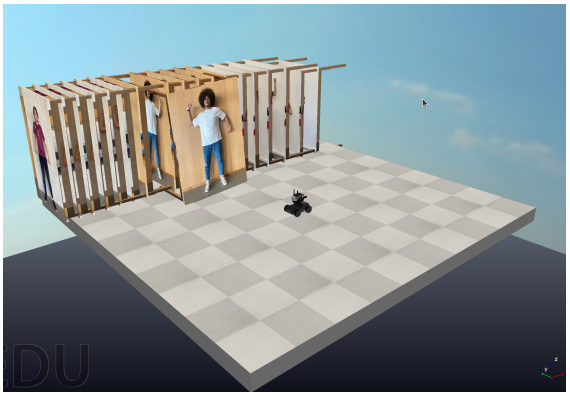

Figure 6: Tables with pictures of ourselves performing the 7 gestures.

We took a total of 21 pictures and mapped them to the tables. In total, we have 9 gestures. This

setup ensured that gestures could be detected independently of background, clothing, or other factors.

3.1.1 In-Simulator Camera

Initially, we noticed that Mediapipe didn't detect our friend Bill[1] very well.

Figure 7: Bill being wrongfully tracked by the Robomaster.

So, we switched to using the tables, and this worked perfectly.

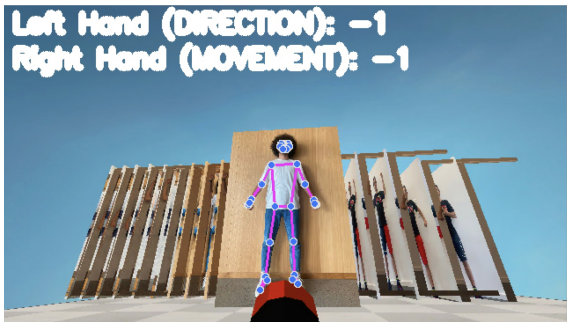

Figure 8: Yassine Table being tracked by the Robomaster.

3.1.2 Real-Life Camera

We also attempted to use the actual camera from our laptops to control the simulated Robomaster robot. However, due to certain issues beyond our control, the camera only worked at 0.5 fps and displayed significant artifacts that offset the camera frame to the right.

3.2 Deployment

We wanted to ensure our system works effectively with the Robomaster, so we conducted tests on two different units. During these tests, we encountered some issues beyond our control.

[1]Bill is the human model

3.2.1 Pink Robomaster

After launching the bridge and connecting to this Robomaster, we experienced significant issues with handling the gimbal.

The gimbal only worked 1 out of 7 times after restarting the Robomaster. Furthermore, after approximately 13 minutes, the gimbal began moving on its own.

Figure 9: Pietro attempting to tame the pink Robomaster

After consulting with our colleagues from the robotics lab, we learned that the pink Robomaster was the first one they assembled, and it had a firmware issue due to them missinstalling the S2 version in the S1 robomaster, and that that could be the issue that were experiencing with gimbal only working sometimes.

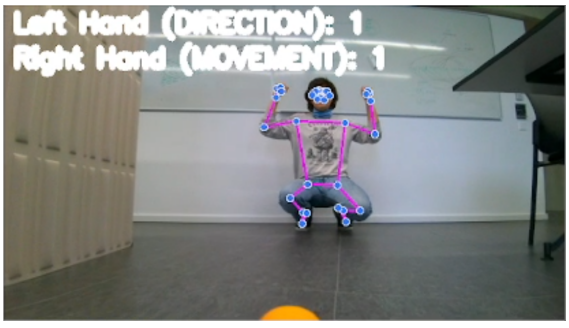

Figure 10: Pietro being tracked by the pink robomaster.

The TAs advised us to use the grey Robomaster but cautioned that the issue of the gimbal moving on

its own is a known problem still being addressed by the bridge's developer.

3.2.2 Grey Robomaster

As for the grey robomaster, we didn't had any strange issues, excepting the gimbal moving on its own after 13 minutes, everything worked smoothly.

Figure 11: Elvi controlling the Grey Robo Master

4 Uninplemented improvements and ideas

Here we describe the ideas and improvements we tought about but didn't had the time to implement:

Currently, the Robomaster detects when the user is out of sight and begins searching for them. However, we also wanted to add a feature where, if the Robomaster turns too far in one direction causing the gimbal to lose track of the user, it would automatically and instantly rotate the gimbal to keep the user in view.

We also planned to implement a "following" feature. This would allow the Robomaster to follow the user based on a specific gesture and to remain in place ("orbit") when the user turns around.

4.1 Demo

We recorded a demo where we guide the robomaster trough a simple maze: https://www.youtube.com/watch?v=KHzfW1WePFc

5 Code

The whole code for the project can be found in the following repository: https://github.com/Frenzoid/USI_RB

References

[1] ROS Wiki. *cv_bridge: Converting Between ROS Images and OpenCV Images (Python)*. Accessed: 2024-05-17. 2024. URL: http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython.