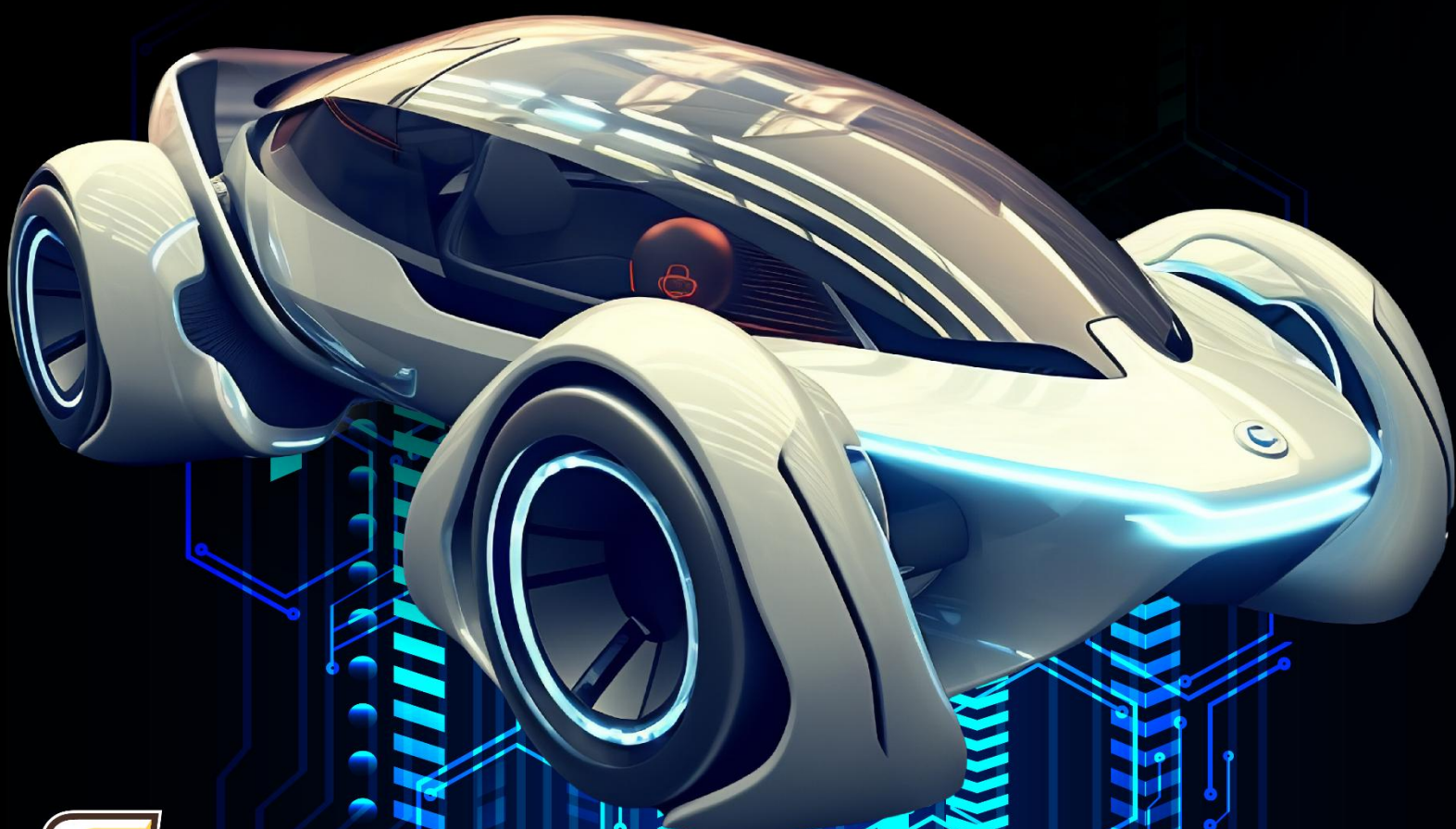
2025 GCCC STEM SUMMER CAMP

TURN RIGHT:

EXPLORING AUTONOMOUS VEHICLES AND ARTIFICIAL INTELLIGENCE

Instruction Manual and Project Guide



Student Activity 1: Custom Dance Routine Instructions

1) Upload code: ***_4a_robot_dance_CONTROLLER*** as is, don't modify.

This code is needed in order to synchronize the LED's, music, and movements, as well as to simplify a few actions.

2) Upload code ***4a_robot_dance_MAIN***

3) Go to the section of the code that has the ***mario_theme*** variable.

You can replace this song with any other song in the ***4b_music_for_robot_dance*** program. Erase everything related to the current song and copy the new song from 4b into 4a.

4) Take a look at the section that says ***SET CUSTOM LED COLORS***

The values can be changed, added, or removed. The LED's are designed to be cycled while the robot is moving.

5) Take a look at the function called ***"def dance_routine"*** towards the bottom of the program.

This function combines a number or pre-set routines that go into the dance along with a pasue command.

- The pause is optional (I included it to better show when one move ends and the other starts)
- Each function (step in the routine) has two (optional) parameters:
  - ***Duration_ms*** – how long each move should last
  - ***Speed*** – how fast the move should be performed
    - 1.0 is full speed 0.1 is the slowest possible
  - The ***wiggle*** function is special because it takes a third parameter **cycles** which tells the move how much times to repeat an action.
    - This is only valid when used in a **loop**

6) Take a look at one of the functions in the dace routine, for example **curve_right_gentle**. Find that function **def** in the code above.

```
def curve_right_gentle(duration_ms=600, speed=1.0):
  s = get_speed(speed)
  set_motors(s, s / 2)
  run_move(duration_ms)
```

Every function NEEDS only ONE of these

This sets the motor speeds

This runs the motors at the setting above for the length passed into the function

The ***set_motor*** and ***run_move()*** commands can be repeated if a move has multiple 'steps'

7) Take a look at the ***def custom_move*** function. This is where you can create your own custom move similar to the pre-defined moves.

- Replace ***pass*** with your own ***set_speed*** command. Make sure ***run_move*** is called after every set speed.

- You can also change the name of the function from ***custom_move*** to anything else, just make sure to use that name in the ***dance_routine*** function if you want that move/function to run.

8) For an added challenge you can:

- Try to make a custom move like **wiggle** that uses a **cycle**

- Coordinate a dance routine with another robot

Notes

🔧 Lab Activity 1: Tuning a PID Controller for Straight-Line Motion

🧭 Objective

In this lab, you will tune the **Proportional-Integral-Derivative (PID)** parameters to help the robot travel in a **straight line** for a specific distance.

Ideal behavior:
If the PID parameters are perfectly tuned, the robot will drive straight toward the target distance **without veering left or right**, and stop **precisely at the desired location**.

🔧 Setup Instructions

1. Upload the code *5_encoder_PID_Tune.py* to the robot
2. Place the robot on a flat, open surface with enough room to drive at least 0.5 meters forward.
3. Make sure the variable target_value = _____ is set in the code before uploading.

🔢 Tuning Parameters

Locate these values in the code and adjust them during the lab:

Kp = ___ # Start with 30

Ki = ___ # Start with 1

Kd = ___ # Start with 0

You will experiment with each term **one at a time**, observing how the robot's behavior changes.

🔬 Part 1: Proportional Control Only (P)

1. Set Ki = 0 and Kd = 0. Vary Kp.

Kp Value	Behavior Observed	Does it go straight? Is it wobbly? Does it correct too much?
10		
20		
30		
40		

💬 Which value gave the straightest motion with minimal correction?

Part 2: Add Integral Control (PI)

2. Set Kp = your best value, try small Ki values.

Ki Value	Behavior Observed	Does it fix long-term drift? Any overshooting or overcorrection?
0.5		
1.0		
2.0		

💬 Does integral control help reduce drifting over time or increase it?

Part 3: Add Derivative Control (PID)

3. Use your best Kp and Ki, and try adding Kd.

Kd Value	Behavior Observed	Does it dampen the motion? Does it reduce wiggling?
5		
10		
15		

💬 Did adding Kd make the robot smoother or more sluggish?

✅ Final Testing

Using your best tuned values:

Kp = ___

Ki = ___

Kd = ___

Place the robot at the start and run the code.
Record the final behavior:

🏁 Bonus Challenge (Optional)

Try increasing the target_value to 12,000 or 20,000.

Can your robot **maintain straightness over a longer run** with your tuned values?

🔧 Lab Activity 2: Warehouse Navigation Challenge

In this challenge the goal is to use a pre-tuned PID system in order to navigate precisely between a few waypoints.

1) Start by uploading code ***_6_PID_controller_*** to the robot.

While we CAN tune ***base_speed***, ***kp***, and ***kd*** in this code, we will attempt to leave it as it is (at least at first)

2) Upload Code ***6_PID_main***

Inside this code there is a function that says ***def move_sequence*** this is where you will program your route. There are three commands to be aware of:

1) ***controller.drive_straight(<ticks>)*** This controls how far the robot should move forward

   This can only be a positive number

2) ***controller.turn_degrees(<degrees>)*** This controls how many degrees to turn.

   Positive value is **TURN LEFT**, negative value is **TURN RIGHT**

3) ***process_movement()*** This has to follow any straight or turn movement (It would have made the code more complicated by not having it in there)

3) Find the correct sequence of moves that can precisely hit the waypoints of the 'warehouse'

Tip #1: Make sure to align the robot as straight as possible when starting the drive routine

Tip #2: If you work out how long a distance of say 1000 ticks is in centimeters or inches then it could reduce some of the guesswork when making the routine.

Notes

Student Activity 2: "Tuning for the Perfect Escape"

How does the code work?

1) The code reads the presence of black tape under multiple sensors using the ***border_threshold*** parameter.

2) To prevent false positives, the code uses the ***strong_black_threshold*** parameter to confirm that at least one sensor has a good view of the black tape.

3) The code uses the ***confirm_required*** parameter to get a second reading of the tape after a short amount of time to ensure a shadow or other factors don't effect an actual reading.

Variable	Used for...	Benefit
border_threshold	Early black detection	Reacts faster to approaching lines
strong_border_threshold	Confident detection	Prevents reacting to noise or shadows
confirm_required	Requires sustained detection	More reliable decisions

Objective

Students will adjust line detection and avoidance parameters to improve how reliably the robot stays inside a black tape boundary without overcorrecting or missing lines.

Setup

- Place the robot inside a **square or circular black-tape boundary**.
- Load the boundary-avoidance script (7_boundary_contain) with these parameters exposed at the top of the file.

Instructions

1. **Calibrate** the sensors by pressing Button A.
2. Observe robot behavior with default settings.
3. Tune **one parameter at a time**, rerun the robot, and **record the results** in a table.
4. Repeat for all five parameters.

Parameters to Adjust

Param	Try Values	Observe
border_threshold	800, 880, 940, 980	When does the robot detect the line? Too early? Too late?
strong_black_threshold	880, 940, 970, 990	How often does it false-trigger vs. miss a real line?
confirm_required	0, 1, 3	Does it wobble near the line or act decisively?
drive_speed	1200, 2000, 3000	Does speed affect accuracy or responsiveness?
turn_duration_ms	300, 550, 700	Does the robot oversteer or not turn enough?

Custom Boundary Activity

Use the black tape to create a custom boundary and test if the robot is able to stay inside of it as effectively as in the rectangle.

Notes

🔧 Lab Activity 3: Lane-Assist Robot Lab

Simulating Real-World Vehicle Safety Systems

Overview

In this lab, you'll explore how a robot can use its sensors to **detect and avoid lane markings**—simulating the behavior of modern **lane-assist systems** in vehicles. By adjusting key parameters like speed and sensor sensitivity, you'll fine-tune its ability to stay "in the lane" while avoiding crossing black tape boundaries.

🎯 Objective

- Understand how IR sensors can be used to simulate lane detection
- Tune parameters to improve the robot's boundary avoidance behavior
- Explore **how speed affects** the robot's ability to stay within a lane
- Design your own test track to challenge the robot's lane-assist behavior

---

🔬 Part 1: Setup and Calibration

1. **Upload the following codes:**
    1. _8_lane_keep_controller
    2. 8_lane_keep_main
2. **Place** the robot on a clean surface with black electrical tape forming a square, lane, or path
3. **Press A** to begin calibration. The robot will spin briefly to capture values for white floor and black tape
4. After calibration, **press A again** to begin forward movement. The robot will attempt to stay within boundaries by **steering away** from dark lines

---

🔬 Part 2: Parameter Tuning Experiment

There are three parameters to tune and all three are in the ***_8_lane_keep_controller*** (You will still run it from the other program though)

1) ***border_threshold:*** *Minimum sensor value considered "black"*

2) ***speed:*** Forward driving speed. Recommended range: 1500–3000.

3) ***turn_strength****: How sharply the robot steers to avoid a black line.*

Try different parameter combinations suggested in the chart and observe the result. **Your goal is to find the fastest speed that still avoids tape reliably.**

Trial #	Speed	Border Threshold	Turn Strength	Possible Prediction of Results (may not though)	Your Results/observation May be different than previous column
1	2500	500	0.2	Too sensitive, many false positives	
2	2000	600	0.4	May wobble or turn too early	
3	1500	800	0.6	More stable but slower	
4	2000	850	0.4	Likely to avoid and recover cleanly	
5	2000	900	0.4	May miss tight turns, less responsive	
6	⭐ **Your Ideal**			Most successful settings	

Part 3: Create your own lane

Design a lane-marked track using black tape. Test how efficiently your robot can move with different types of curvature. For example, will your robot be able to turn if the lane steeply banks left?

The lane can include:

- **Straight paths**
- **Gentle Curves**
- **Tight curves**
- Optional: add broken lines to simulate a passing lane or an intersection that splits the lane in two paths

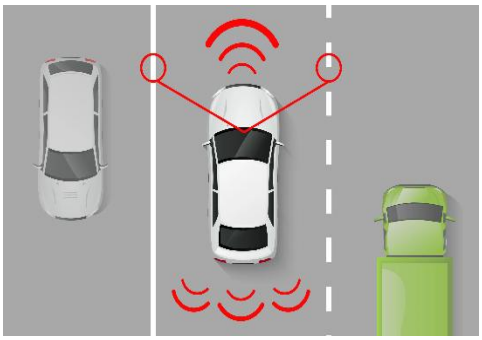
Goal: Find the **maximum speed** the robot can handle while still successfully staying within the lane!

Test #	Curve Type	Turn Radius or Angle	Speed Tested	Did Robot Avoid Line?	Notes (e.g., overcorrect, failed)
1	Gentle curve	~30 cm radius		✅ / ❌	
2	Moderate curve	~20 cm radius		✅ / ❌	
3	Tight curve	~10 cm radius		✅ / ❌	
4	Sharp corner	~90° turn		✅ / ❌	

Real-World Connection

Modern cars use **lane departure warning systems** and **lane-keeping assist** to help drivers stay in their lanes. These systems use cameras and sensors to detect road lines and steer the car back if it drifts. This lab demonstrates a similar concept using IR sensors and basic motor control.

Instead of line sensors and black tape, real cars use:



- **Forward-facing cameras** (usually mounted near the rearview mirror) that look at the road ahead.
- **Computer vision algorithms** to detect white or yellow lane markings and determine if the vehicle is drifting.
- **Electric steering controls** to gently nudge the car back into the lane — this is called **Lane Keeping Assist (LKA)**.
- Some systems also include **Lane Departure Warnings (LDW)** that beep or vibrate the steering wheel if the driver starts to leave the lane unintentionally.

Advanced vehicles combine this with **radar or lidar** to track nearby vehicles and keep the car centered in the lane, even in bad weather.

➞ Just like your robot had to tune its sensor thresholds and adjust its steering to stay between the lines, real cars need careful calibration and advanced software to help drivers stay safe on the road.

Notes

Notes: Getting Started with MentorPi

Documentation (new): https://docs.hiwonder.com/projects/MentorPi/en/latest/index.html

Documentation (old): https://drive.google.com/drive/folders/1Ox5xN5zpxXqDK-9ruDwwgcQgePXvMvHr

Assembly

Note 1: Please see the assembles example for any 'confusing' steps

Note 2: Don't use excessive force and BE GENTLE, some of the wires and components are quite delicate

1) Follow getting ready to assemble
    a. 1.4 Step 1
    b. Skip to Ackerman Chassis and follow those steps
    c. Follow the top plate build instructions (as shown in the mecanum version)
    d. Refer to page 03 of the printed manual for the interface instructions to connect the components to the Rpi5

2) Turn on the robot and let it FULLY BOOT UP.
    a. If the power is switched off during first boot it will corrupt the EEPROM and will have to be reset using a special process
    b. You should hear a single beep when all the systems are ready

Important MAYBE!

It is possible that the robot will be defaulted into mecanum drive mode. If that is the case, this needs to be switched to Ackerman drive by connecting to VNC, following 2.2 in the new docs and switching machine to Acker

First Run:

3) Turn on the wireless remote. As long as the USB dongle is connected to the Pi, the remote should be recognized almost immediately.
    a. See 1.5 in old documentation

Connecting to AP and Optional App Control :

4) The WonderPi app on iOS and Android allow experimenting with various features of the robot.

    a. The device has to first be connected to the robots AP wifi which will be **'GCCC_Robot_#'** The # is listed on the box
    b. The password is **hiwonder**

Development and ROS2 access:

Connecting

5) Connect to realVNC using the robots ip **192.168.149.1**

    a. Note the passwords for VNC connecton:
    Username: pi
    Password: raspberrypi

6) The actual system of the robot runs in a Docker container that can be opened with the Terminator application (icon shown here)

IMPORTANT!

Before running any commands App control needs to be disabled using ~/.stop_ros.sh or thing will not work correct or at all.

Some of the custom scripts include commands that disable it when run.

To re-enable app control the robot can be restarted.

Notes

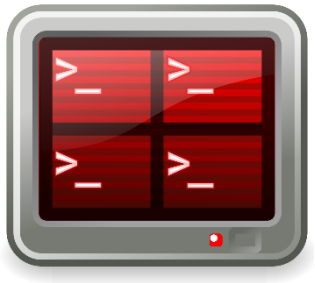
Notes: Exploring ROS2 nodes and topics using the MentorPi robot

Important notes:

- The tab key gives the ability to autocomplete commands to prevent entering the entire line.
- After every restart ensure that app control is disabled when using the terminal as detailed in step1

Terminator Commands

Ctrl + C	Stop execution in a terminak
Ctrl + L	Clear the terminal
Ctrl+Shift+O	Split terminals Horizontally.
Ctrl+Shift+E	Split terminals Vertically.
Ctrl+Shift+T	Open new tab.
Ctrl+Shift+X	Toggle terminal full screen



1) Stop app control with the following command.

```
~/.stop_ros.sh
```

2) Enter the command below.

```
ros2 node list
```

Notice that nothing appears since there are no nodes that are on the ROS2 network.

3) Enter: `ros2 topic list`

You should see two topics shown. These are internal to ROS2 itself and are used to manage the specific nodes and topics.

4) Enter the following:

```
ros2 launch ros_robot_controller ros_robot_controller.launch.py
```

This command starts a launch file that can be configured to do a variety of things related to the robots operation. In particular, this launch file starts the **`ros_robot_controller`** node as well as handles a few other configuration tasks.

Confirm that the node mentioned above is running by entering: `ros2 node list`

Check which topics the node establishes with the command: `ros2 topic list`

6) Let's look at HOW messages are sent on specific topics. We will chose the /ros_robot_controller/set_rgb topic and look at what **interface** it uses.

Enter: `ros2 topic info /ros_robot_controller/set_rgb`

You should see that this topic has one subscriber and no publishers. That means that one other node is listening to messages that this topic broadcasts.

You should see under 'type' that there is an interface for this topic that is called 'RGBStates'

Enter: `ros2 interface show ros_robot_controller_msgs/msg/RGBStates`

You will see that there are four data types that this interface uses:

int32 index
uint8 red
uint8 green
uint8 blue

This represents which led to trigger and the value it should take

You are now ready to send commands (publish) commands on this topic using the interface queried.

NOTE: Some topics such as set_motor require other nodes to be launched. SO while we can do the same for this (and any other) topic, we might not be able to do anything useful yet.

7) Let's publish to the topic to remontely trigger an LED on the robot enter the following (Don't press enter until everything is typed it as shown):

```
ros2 topic pub /ros_robot_controller/set_rgb
ros_robot_controller_msgs/msg/RGBStates "{states: [
  {index: 1, red: 255, green: 0, blue: 0},
  {index: 2, red: 255, green: 0, blue: 0}
]}"
```

- Notice the text on your screen that says publishing #<n> and that the two LEDs on your robot have turned red.
- This confirms that your message is being received by a **node that subscribes to the /ros_robot_controller/set_rgb topic**.
- That node runs code which processes the message and **physically sets the LED colors** on the robot.

8) Try it – use what you learned to learn about the /ros_robot_controller/battery topic.

What is the msg type (interface) used by the battery topic: _____

Notice that /ros_robot_controller/battery is a **publisher** topic — this means a node is **sending data** on this topic, and any other node can subscribe to receive the battery readings. This topic **does not receive** data; it only **publishes** battery status updates.

9) We can view the data that is being published by using: `ros2 topic echo /ros_robot_controller/battery`

We can observe something like this: data: 7651

This represents battery voltage in mV, so 7651mV = 7.65V

High-Level Movement Data

Now let's start digging into the system that allows the robot to move efficiently by calculating its position. There are two types of data that need to be considered, IMU and odometry.

IMU calculates the acceleration and angular velocity of the robot in each of its three axes.

Odometry uses encoders on the wheels to calculate the motion of the wheels.

10) We can start the main controller as in step 4:

```
ros2 launch ros_robot_controller
ros_robot_controller.launch.py
```

11) We then need to start the IMU controller:

```
ros2 launch peripherals imu_filter.launch.py
```

12) If we examine the new topics that the IMU introduced: `ros2 topic list`

…. We can see that there is a new /imu and /imu_corrected topic.

By typing `ros2 topic info /imu`, we can see that one node is publishing to this topic — which means that IMU data is being sent and is available for other nodes to receive.

13) Just like with the battery topic, we can view the data being published on the /imu topic using:

```
ros2 topic echo /imu
```

There is a lot of data that moves fast. It should look something like the image here:

You might notice that if you move the robot some of the numbers will begin to change. This shows a change in the **angular velocity** and **linear acceleration**

14) Now let's view the odometry data. Start your controller (or ensure it is running):

```
ros2 launch controller controller.launch.py
```

15) Look for the **/odom** topic in your topic list

```
ros2 topic list
```

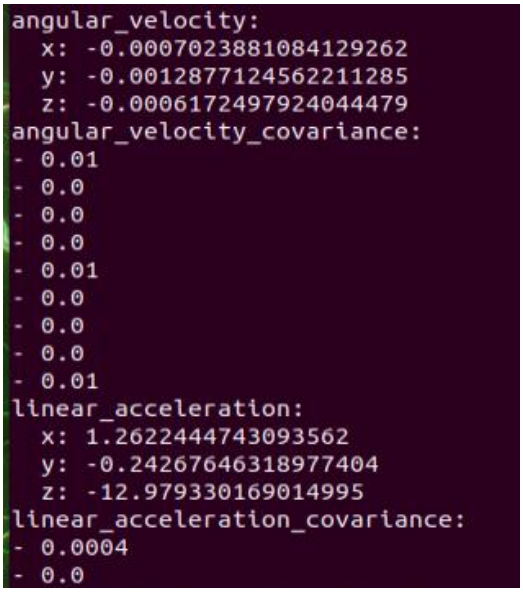
We can also see HOW MANY topic are running (we should see 33):

```
ros2 topic list  | wc -l
```

16) Let's view data that shows the position and orientation of the robot in the three axes.

```
ros2 topic echo /odom
```

While the topic data is displayed move the robot around. While moving the robot press **ctrl+c**. Then scroll up to observe the data. This makes it a little easier to see changes then to keep up with the data being echoed.



Another useful source of data that shows both linear acceleration and angular velocity is:
/imu_corrected

High-Level Movement Control

Let's now drive the robot by publishing data on linear and angular velocity.

The **linear velocity** (relative to the robot) is the ***x-axis*** which represents forward and backward movement. This can range from about **-0.6 to 0.6 (m/sec)**

The **angular velocity** (relative to the robot) is ***z-axis*** which represents the left and right turn. This can range from **-2 (full right) to 2 (full left)**

17) The /cmd_vel command is what enables high level movement control (common in many ROS applications). Let's see what type of message the cmd_vel topic uses:

```
ros2 topic info /cmd_vel
```

We see - **Type: geometry_msgs/msg/Twist**

18) Let's inspect how these messages are structured:

```
ros2 interface show geometry_msgs/msg/Twist
```

This shows that the message contains:

- A linear field with 3 float values (x, y, z)
- An angular field with 3 float values (x, y, z)
  These are used to represent linear and angular velocity in 3D space.

19) Let's send move commands by publishing to the velocity control topic using the message type we observed:

```
ros2 topic pub /controller/cmd_vel geometry_msgs/Twist "linear:
 x: 0.2
 y: 0.0
 z: 0.0
angular:
 x: 0.0
 y: 0.0
 z: 0.0"
```

This command tells the robot to **drive forward at 0.2 m/s with no rotation**.

⚠️ Make sure the motion controller node is running (ros2 launch controller controller.launch.py), or the robot will not respond.

⚠️ Make sure the robot is on a flat surface and has room to move

⚠️ It's a good idea to have a second window open and ready to send a follow up stop command whenever a move command of this sort is published.

```
ros2 topic pub /controller/cmd_vel geometry_msgs/Twist "linear:
 x: 0.0
 y: 0.0
 z: 0.0
angular:
 x: 0.0
 y: 0.0
 z: 0.0"
```

LiDAR (Light Detection and Ranging)

A LiDAR on a robot or autonomous vehicle collects thousands of distance readings per second. As it rotates, it emits laser pulses and measures the time it takes for each pulse to bounce back. By combining this timing with the angle of each pulse, the LiDAR builds a detailed 2D (or sometimes 3D) map of its surroundings. This data provides accurate information about the shape and distance of nearby objects, helping the robot detect obstacles, navigate, and understand its environment. As it continuously scans, the LiDAR updates this map in real time, enabling effective navigation and environmental awareness.

In more advanced LiDARs, the system can generate full 3D maps by using multiple laser beams or by tilting the sensor, and some can even measure the velocity of moving objects using Doppler shift, enabling detailed perception for applications like autonomous driving and high-precision mapping.

20) Double check app control is disabled: `~/.stop_ros.sh`

21) Launch the LiDAR control stack:

```
ros2 launch app lidar_node.launch.py debug:=true
```

By viewing the active topics `ros2 topic list`

...we can see that:

1) the topics from the launch controller all appear as we used previously
2) A new topic **/scan_raw has** appeared

23) Tell the robot to begin processing lidar data by entering:

```
ros2 service call /lidar_app/enter std_srvs/srv/Trigger {}
```

You can confirm that data is being sent from the LiDAR by entering: `ros2 topic echo /scan_raw`

24) The LiDAR has a few different working modes (which is often the case in many other robotic systems built on ROS).

Enter the following:

```
ros2 service call /lidar_app/set_running interfaces/srv/SetInt64
"{data: 2}"
```

This put the robot into 'Following' mode. If you (or any object) is in the field of view of the robot and moves, the robot will attempt to follow it while maintain its distance.

We can stop this behavior and put the robot back into idle with the command:

```
ros2 service call /lidar_app/set_running interfaces/srv/SetInt64
"{data: 0}"
```

25) Now, let's put the robot into **Obstacle Avoidance** mode. In this mode, the robot will navigate its surroundings and actively avoid obstacles — as long as those obstacles are at or above the height of the LiDAR sensor.

Enter:

```
   ros2 service call /lidar_app/set_running interfaces/srv/SetInt64
"{data: 1}"
```

⚠ Safety Note

Avoid testing near edges or stairs — the robot does **not** currently have any mechanism running to detect or avoid drop-offs.

Keep in mind, putting in '0' for data will again put the robot and LiDAR in standby.

The chart below summarizes the modes:

data value	Mode	Description
0	**Standby**	**Idle, not doing anything**
1	**Obstacle Avoidance**	**Moves forward while avoiding obstacles**
2	**Following**	**Follows a moving object in front of the robot**
3	**Guarding**	**Turns to keep front pointed at object in view (not available on Ackermann chassis)**

Depth Camera

The MentorPi robot is equipped with a **binocular 3D depth camera** that allows it to **see and understand its surroundings in three dimensions**. Unlike a regular camera that only captures color, the depth camera also measures **how far away each part of the scene is**, creating a "depth map" of the environment.

26) Run the following two commands to stop the app control and activate the depth camera topics:

```
~/.stop_ros.sh
```

```
ros2 launch peripherals depth_camera.launch.py
```

27) ROS2 has a built in tool that allows developers to visualize and debug data from a system. We will use the tool to view the output from the camera. In the command line enter the following:

```
rqt
```

This should bring up the rqt tool

In the tool go to: **Plugins > Visualization > Image View**

In the drop down box at the top make sure **/ascamera/camera_publisher/rgb0/image** is selected

28) We also have the ability to view our video feed from a browser. To do this we first have to re-enable app service in the terminal  (NOT THE DOCKER CONTAINER!)

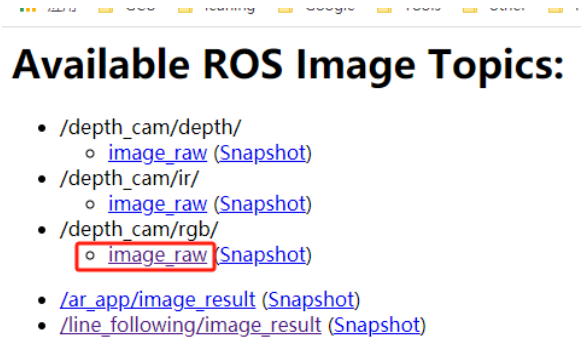
 Enter: `sudo systemctl restart start_node.service`

29) Now we can bring up a broswer and enter:

`http://192.168.149.1:8080/`

Alternatively, as long as we are running in AP mode we can enter:

`http://localhost:8080/`



Either way are broweser should opn something like the image here:

Clicking on 'image' below /ascamera/camera_publisher/rgb0/ will bring up the video feed

AI and Machine Learning

The MentorPi comes with built in AI models that allow it to do a number of things such as pose detection, posture control and autonomous driving.

Fingertip Trajectory Recognition

30) Start the depth camera:

```
ros2 launch peripherals depth_camera.launch.py
```

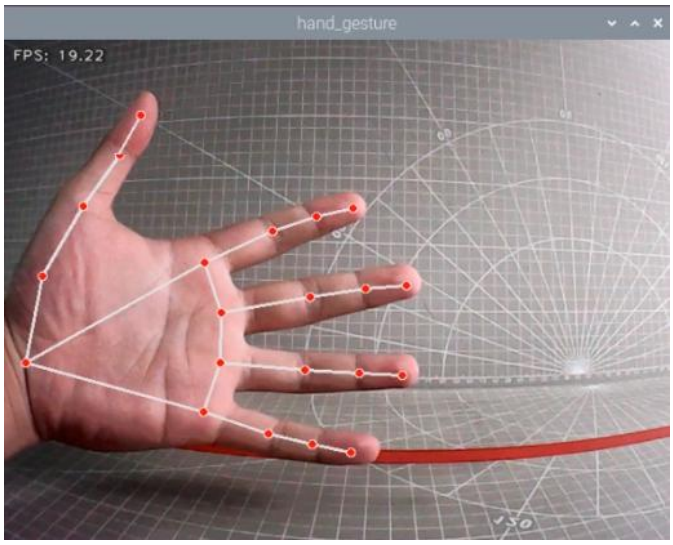
31) Change directories to the hand recognition program:

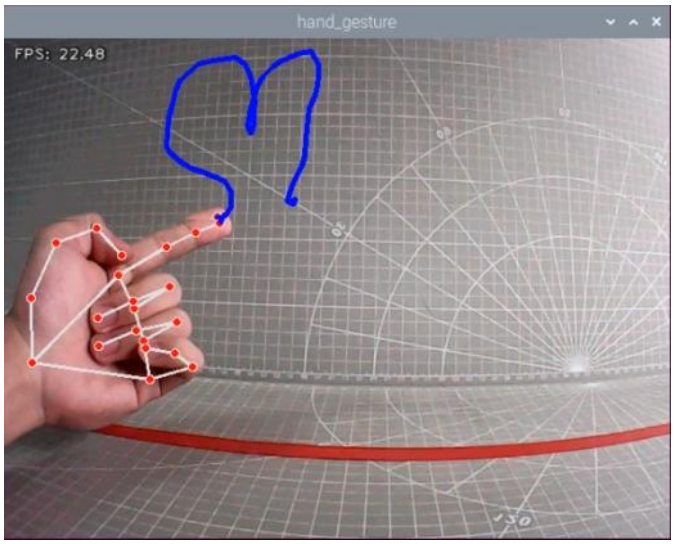
```
cd ros2_ws/src/example/example/mediapipe_example
```

32) Start the hand recognition program:

```
python3 hand_gesture.py
```

If the robot detects the "**1**" gesture, the trajectory of your fingertip motion will begin to be recorded on the live camera feed. If it detects the "**5**" gesture, the recorded fingertip trajectory will be cleared.



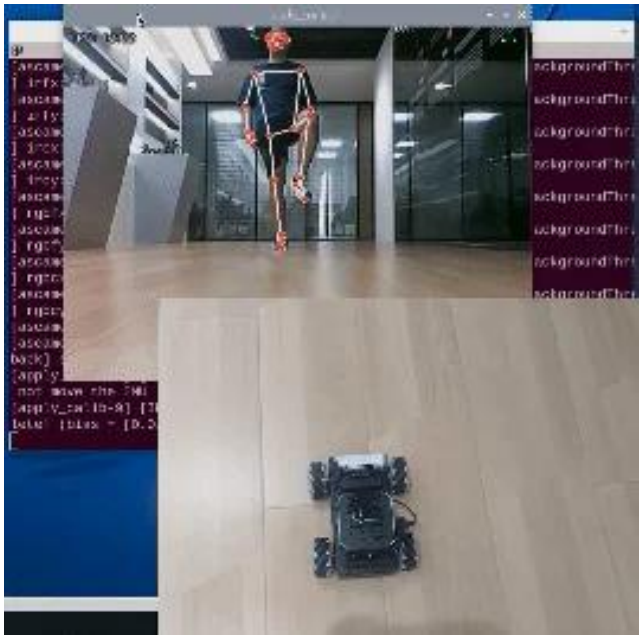


Posture Control

Stop all previous nodes `~/.stop_ros.sh`

33) Start the posture control program:

```
ros2 launch example body_control.launch.py
```



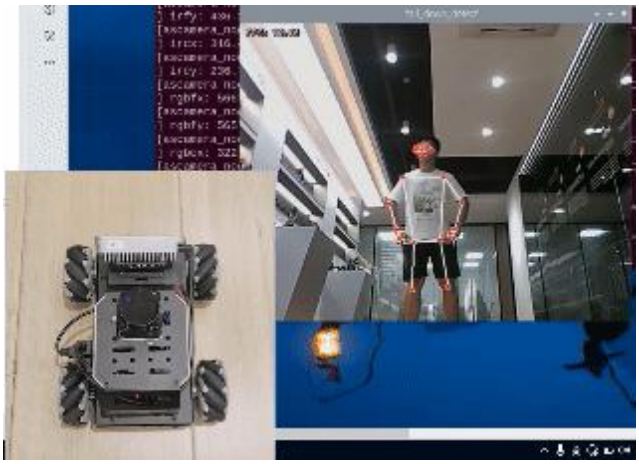
When the program starts make sure the body is fully in the robots view.

From the perspective of the robot, when you raises you left arm, the robot will turn left; when your right arm is raised, the robot will turn right; when your left leg is raised, the robot will move forward; and when your right leg is raised, the robot will move backward.

Pose Detection

34) Start the pose detection program:

```
ros2 launch example
fall_down_detect.launch.py
```



Ensure the body is within the view of the camera. Quickly get into a pose like your are trying to sit down on a chair.

The robot will register this pose as **‘falling’.** It will beep and move back and fourth as though it is ‘panicking’ or calling for help.

Notes