**Robot class Documentation/Guide**

Refer to this documentation for information on the functions in the Robot class.

For help navigating this documentation, try using our *experimental* [AI ChatBot](https://poe.com/Bot_for_TIC_S24):

<https://poe.com/Bot_for_TIC_S24>.

Please confirm any suspicious instructions with the team.

**Provided Code**

boiler\_plate.py

The script that will contain the logic you write will be in a copy of the boilerplate.

\_\_init\_\_.py

Class definitionwhichhas the robot class. This will be initialized as an object which allows you to use the attributes, methods, and functions which interface with the robot.

solution-joystick.py

The joystick solution provides an example of a basic keyboard solution which uses a function in the robot class.

**Robot Hardware**

**NOTICE:**  
When connecting to the robot or to use the sim you need to have the following 3 lines in the solution python script:

| #Setting up for using the sim  TMMC\_Wrapper.is\_SIM = False  if not TMMC\_Wrapper.is\_SIM:  TMMC\_Wrapper.use\_hardware() |
| --- |

Add **global is\_SIM** to the **\_\_init\_\_.py:**

| 52 def use\_hardware():  53 global is\_SIM  54 if not is\_SIM: |
| --- |

**Overview of Robot Wrapper Class for ROS and Hardware Interaction**

The Robot class acts as an API to interact with ROS (Robot Operating System) and various hardware components on the TurtleBot. This class encapsulates functionalities needed for controlling a robot, subscribing to sensor data and publishing commands.

Note: You can change and modify this class to best suit your solution

Key Components

* Node Initialization and Subscriptions:
  + The class inherits from rclpy.node.Node, initializing itself with the node name 'notebook\_wrapper'.
  + It subscribes to multiple topics such as LaserScan, IMU, camera images, camera info, and battery state. These subscriptions use appropriate QoS profiles to manage data flow.
* Action Clients:
  + The class initializes action clients for docking and undocking the robot, using ROS 2's action client interfaces.
  + It waits for these action servers to be available before proceeding.
* Transform functions:
  + A transform listener is set up to handle transformations between different coordinate frames using tf2\_ros.
  + Utility methods include transforming coordinates, reducing transformations to 2D, calculating rotation from transforms.
* Velocity Commands:
  + The class includes multiple functions for different modes of velocity control for both linear and angular motion.
* Logging:
  + Methods are provided to start and stop logging of various topics to a ROS 2 bag file. This is useful for recording sensor data and other relevant information for debugging or analysis.
* Sensor Data Handling:
  + Callback methods process incoming sensor data and store the latest messages.
  + Methods such as checkScan, checkImu, checkImage, checkCamera, and checkBattery provide access to the sensor data.
* Teleoperation:
  + Functions for manual control using keyboard input are provided, including methods to move forward, backward, and turn. A keyboard listener is implemented for this purpose.
* Vision Functions:
  + Vision functions provide AprilTag detection, for color filtering and contouring.

**Robot Wrapper Functions (Highlighted are key functions)**

**Initialization and Setup**

1. init(self)
   * Params: None
   * Return: None
   * Description: This will initialize an object of the robot class. Allows you to use the attributes, methods, and functions which interface with the robot.
2. use\_hardware()
   * Description: Checks vpn connectivity and connection to robot (Not used in simulation).

**Transformation Functions**

1. get\_tf\_transform(self, parent\_frame, child\_frame, wait=True, time\_in=rclpy.time.Time())
   * Retrieves the transform between two frames.
2. reduce\_transform\_to\_2D(self, transform\_3D)
   * Reduces a 3D transform to a 2D transform.
3. rotation\_from\_transform(self, transform\_2D)
   * Extracts the rotation angle from a 2D transform.

**Logging Functions**

1. configure\_logging(self, topics)

* Configures topics for logging.

1. start\_logging(self)
   * Starts logging data for the configured topics.
2. stop\_logging(self)
   * Stops logging and returns the directory where the log data is stored.
3. get\_logging\_data(self, logging\_dir)
   * Retrieves log data from the specified directory.
4. delete\_logging\_data(self, logging\_dir)
   * Deletes log data from the specified directory.

**Listener Callbacks and Data Retrieval**

1. scan\_listener\_callback(self, msg)
   * Callback for receiving LaserScan messages.
2. checkScan(self)
   * Params: None
   * Return:last\_scan\_msg <http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html>
   * Description: Retrieves the latest LaserScan data as a message object (Lidar data).
3. imu\_listener\_callback(self, msg)
   * Callback for receiving IMU messages.
4. checkImu(self)
   * Params:None
   * Return: last\_imu\_message <https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/Imu.html>
   * Description: Retrieves the latest IMU data as an imu message object.
5. rotation\_angle(self,q)
   * Extract the rotation from the quaternion
   * Params: q (quaternion object)
   * Return: angle (float, radians)
   * Description: Gives the rotation component of a quaternion.
6. conjugate(self,q)
   * Helper to get the conjugate of the quaternion
   * Params:q (quaternion object)
   * Return: (quaternion object)
   * Description: Returns the conjugate of the passed quaternion.
7. quaternion\_multiply(self,q1, q2)
   * Multiply two quaternions
   * Params: q1, q2 (quaternion objects)
   * Return: (quaternion object)
   * Description:Returns the product of two quaternions.
8. has\_rotation\_occurred(self, orientation1, orientation2, desired\_rotation\_angle)
   * Compares two quaternions and checks if the angle between the two is the desired angle
   * Params: orientation1, orientation2 (quaternion objects) and desired\_rotation\_angle (Radians)
   * Return: is\_angle\_close Boolean
   * Description: Return if the angle between two quaternions is near the given parameter angle.
9. image\_listener\_callback(self, msg)
   * Callback for receiving Image messages.
10. checkImage(self)
    * Params: None
    * Return:last\_img\_msg <https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/Image.html>
    * Description: Returns the image message object

**Note: If running in simulation, edit your \_\_init\_\_.py file to include the following lines in the \_\_init\_\_ function in your robot class:**

**Find self.image\_future = rclpy.Future(). The lines below it should reflect an if condition as such:**

if is\_SIM:

self.image\_subscription = self.create\_subscription(Image,'/camera/image\_raw',self.image\_listener\_callback,qos\_profile\_sensor\_data)

else:

self.image\_subscription = self.create\_subscription(Image,'/oakd/rgb/preview/image\_raw',self.image\_listener\_callback,qos\_profile\_sensor\_data)

self.image\_subscription # prevent unused variable warning

**Similarly, find self.camera\_info\_future = rclpy.Future(). The lines below it should reflect an if condition as such:**

if is\_SIM:

self.camera\_info\_subscription = self.create\_subscription(CameraInfo,'/camera/camera\_info',self.camera\_info\_listener\_callback,qos\_profile\_sensor\_data)

else:

self.camera\_info\_subscription = self.create\_subscription(CameraInfo,'/oakd/rgb/preview/camera\_info',self.camera\_info\_listener\_callback,qos\_profile\_sensor\_data)

self.camera\_info\_subscription # prevent unused variable warning

1. checkImageRelease(self)
   * Params: None
   * Return: None
   * Description: Displays the image to the screen
2. camera\_info\_listener\_callback(self, msg)
   * Callback for receiving CameraInfo messages.
3. checkCamera(self)
   * Retrieves the latest CameraInfo data.
4. battery\_state\_listener\_callback(self, msg)
   * Callback for receiving BatteryState messages.
5. checkBattery(self)
   * Retrieves the latest battery percentage.

**Velocity and Movement Control**

1. cmd\_vel\_timer\_callback(self)
   * Timer callback for publishing velocity commands.
2. set\_cmd\_vel(self, velocity\_x, velocity\_phi, duration, stop=True)
   * Params:velocity\_x (float), velocity\_phi (float) , time (int)
   * Return: None
   * Description: Sets the robot's velocity for a specified duration. Linear + angular velocity.
3. send\_cmd\_vel(self, linear\_x, angular\_z)
   * Params: linear\_x (float), angular\_z (float)
   * Return: None
   * Description: Sends a command with linear and angular velocity.

**Docking Actions**

1. undock(self)
   * Initiates the undocking action.
   * Params: None
   * Return: undock (boolean)
   * Description: Accomplishes undocking from changing station (Not for sim).
2. dock(self)
   * Params: None
   * Return: dock (boolean)
   * Description: Accomplishes docking from changing station (Not for sim).

**Teleoperation and Keyboard Control**

1. rotate(self, angle, direction)
   * Starts keyboard control for the robot.
   * Params: Angle(degrees float), direction (+1 or -1 int)
   * Return: None
   * Description: Rotates the robot a specified amount of degrees.
2. start\_keyboard\_control(self)
   * Starts keyboard control for the robot.
   * Params: None
   * Return: None
   * Description: Allows interaction with robots using w,a,s,d keys.
3. stop\_keyboard\_control(self)
   * Stops keyboard control for the robot.
   * Params: None
   * Return: None
   * Description: Terminates the keyboard control.
4. on\_press(self, key)
   * Callback for handling key press events.
5. move\_forward(self)
   * Moves the robot forward using send\_cmd\_vel().
6. move\_backward(self)
   * Moves the robot backward using send\_cmd\_vel().
7. turn\_left(self)
   * Turn the robot left using send\_cmd\_vel().
8. turn\_right(self)
   * Turn the robot right using send\_cmd\_vel().

**Lidar and Obstacle Avoidance**

1. lidar\_data\_too\_close(self, scan, th1, th2, min\_dist)
   * Checks if any lidar data points are too close within the specified angles and distance.
   * Params: scan (last\_scan\_msg object), th1 and th2 (float angle radians ), min\_dist (float in meter)
   * Return: (float)
   * Description: Checks a window of angle to see if there is lidar readings within a given distance
2. detect\_obstacle(self, scan)
   * Params: scan (last\_scan\_msg.ranges) Note: the scan parameter is the array of data from the last\_scan\_msg object
   * Return: min\_dist (float in meter) and min\_dist\_angle (float in degrees) Refer to image below
   * Description: Detects lidar data less than 0.3m away in from 45 to 135 degrees view in the front of the robot.
3. test\_lidar\_orientation(self)
   * Params: None
   * Return: None
   * Description: Tests the orientation of the lidar sensor by printing distances in different segments.

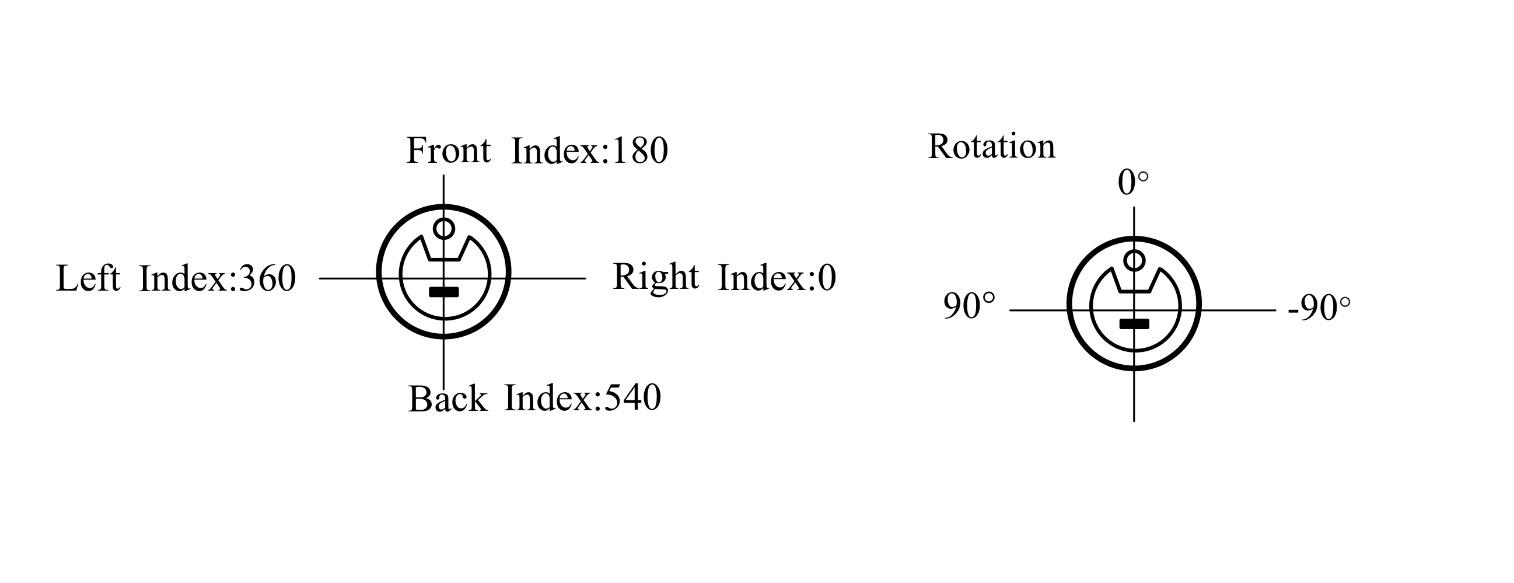
**Vision and AprilTag Detection**

1. detect\_april\_tag\_from\_img(self, img)
   * Detects AprilTags in a given image and returns their IDs and spatial information.
   * Params: image (np array)
   * Return: tags (dictionary) (Only use ID for tags)
   * Description: A function that takes in images and checks for tags and adds it to a dictionary. The distance and rotation output is incorrect.
2. red\_filter(img)
   * Filtering for only red pixels and converting to a binary image
   * Params: img (np array)
   * Return: filtered\_img (np array)
   * Description: Takes in an image and filters only for red pixels returning a binary black&white image
3. add\_contour(img)
   * Params: img (np array)
   * Return: contoured (np array), max\_area (float) and (cX,cY) (tuple of floats)
   * Description: Adding polygon contours to the stop sign
4. rosImg\_to\_cv2(self)
   * Params: None
   * Return: img\_3D (np array)
   * Description: Takes the image from checkImage() and converts it for openCV usage
5. ML\_predict\_stop\_sign(self, model, img)
   * Params: model (pt model), img (np array)
   * Return: stop\_sign\_detected (boolean), x1,x2,y1,y2 (float)
   * Description: Prediction using pretrained model, Returns the detected boolean and bounding box coordinates.

NOTE: Add self to the ML\_predict\_stop\_sign parameters (in red above)

| ML usage:  # Load a pretrained model (COCO dataset)  model = YOLO('yolov8n.pt')  stop\_sign = 0  stop\_sign\_prev = 0 |
| --- |

**Lidar Data and Orientation:**

The lidar data outputs a list with 720 distance readings covering 360 degrees of rotation.   


**Odometry:**We have added some odom functionality to the Robot class. Note these functions are untested

47. reset\_odometry(self)

* Params: None
* Return: None
* Description: Resets the origin location, so the robot thinks it is at (0,0,0)
* <http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html>

Sample Code:

| robot.reset\_odometry() *# make the robot think it is at position (0,0,0)*  *# Display raw data: Odometry based position*  robot.odom\_future = rclpy.Future()pose1 = robot.spin\_until\_future\_completed(robot.odom\_future).pose.pose  print(pose1.position)  print(pose1.orientation) |
| --- |

**Boilerplate Script:**

The scaffold script provides a baseline for you to implement your solutions. There is a while loop to have constant running logic on the robot. There are also some given function calls, rcly.spin\_once(), this function updates the Ros topics which means the sensor data will update.

**Brief Ros Notes:**  
Ros is a middleware. It is an interrupt-event based system; it handles communication for a system. The Apis handle hardware and multiple software communication. Developers develop individual packages that can be easily added into the Ros of a robot. The wrapper class is considered a package which abstracts the hardware.

An overview of the methods of communication. There are three main ways for communication to occur with Ros; Messages, Services and Actions. Messages have something called a topic which is where data is posted and pulled from by a publisher and subscriber, respectively. Services are client based, where requests are made to get information. Actions are used as commands for actuations.

More on this: <https://docs.ros.org/en/humble/Concepts/Basic.html>

**Extra Useful Documentation:**  
<https://turtlebot.github.io/turtlebot4-user-manual/>

<https://docs.ros.org/en/humble/index.html>

<https://docs.ros2.org/foxy/api/rclpy/index.html>