RIDE & ROSRider Beginners Guide

RIDE:

Ride consists of three parts, Views Menu (vertical menu with large icons on the left), Active View (just right of Views Menu) and Editor View (large blank area on right).

In the Views Menu, you can see Explorer, Source Control, Extensions, Debugging, Search and Riders menus. We will explain their usage along the way.

Simulation

Click "Start" in "Simulation" section. This will start the Robotic Operating System and Simulator behind the curtain, and you should see a small mobile robot moving in circles in the Simulator screen.

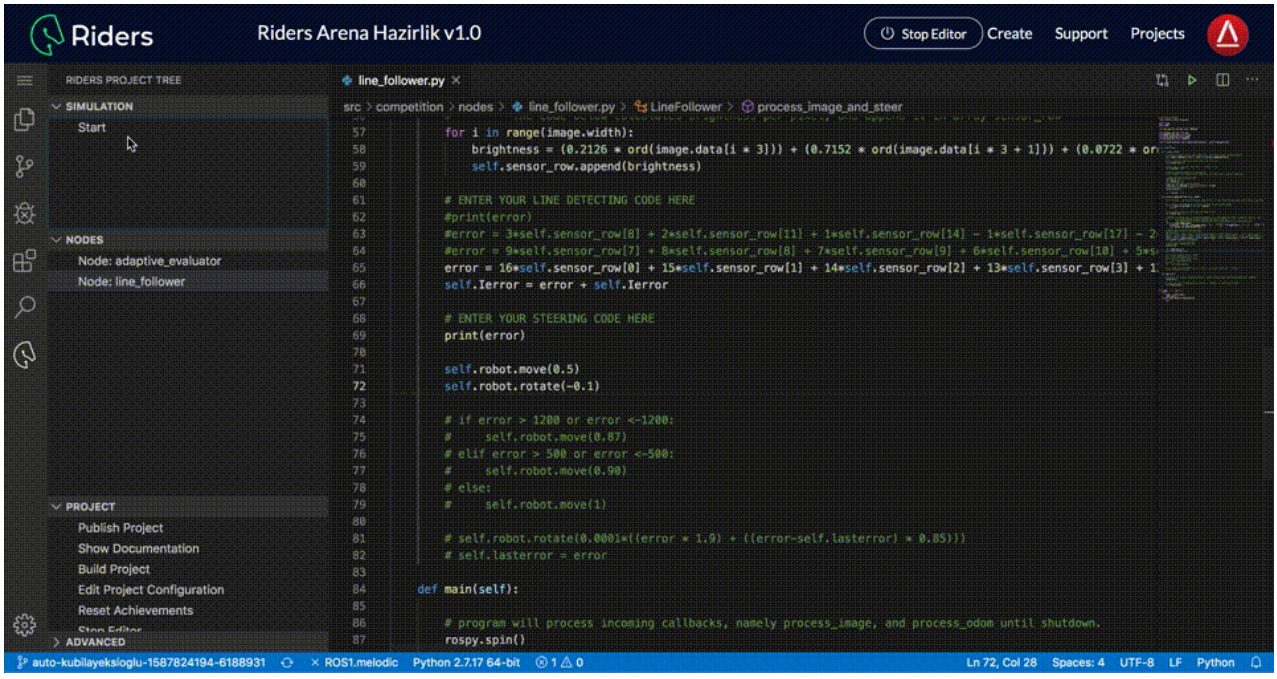


Image 1: Simulation Screen

Once simulation starts, you will see 1 new section called "Sensors" and 4 new menu items in Simulation section.

- **View Simulator**: Opens the simulator view (the one that automatically opened when you clicked Start)
- **Visualize Sensors**: Opens sensor visualizer (you won't be using this in this tutorial)
- **Reset**: Resets simulation to its initial state and restarts your code
- **Stop**: Stops the current simulation, it is useful if simulation fails to resets for some reason (such as an infinite loop)

Before diving into the sensors, let us describe our robot further.

ROSRider

ROSRider is a mobile robot with 2 front wheels, 1 ball caster wheel and 1 RGB line camera at front.

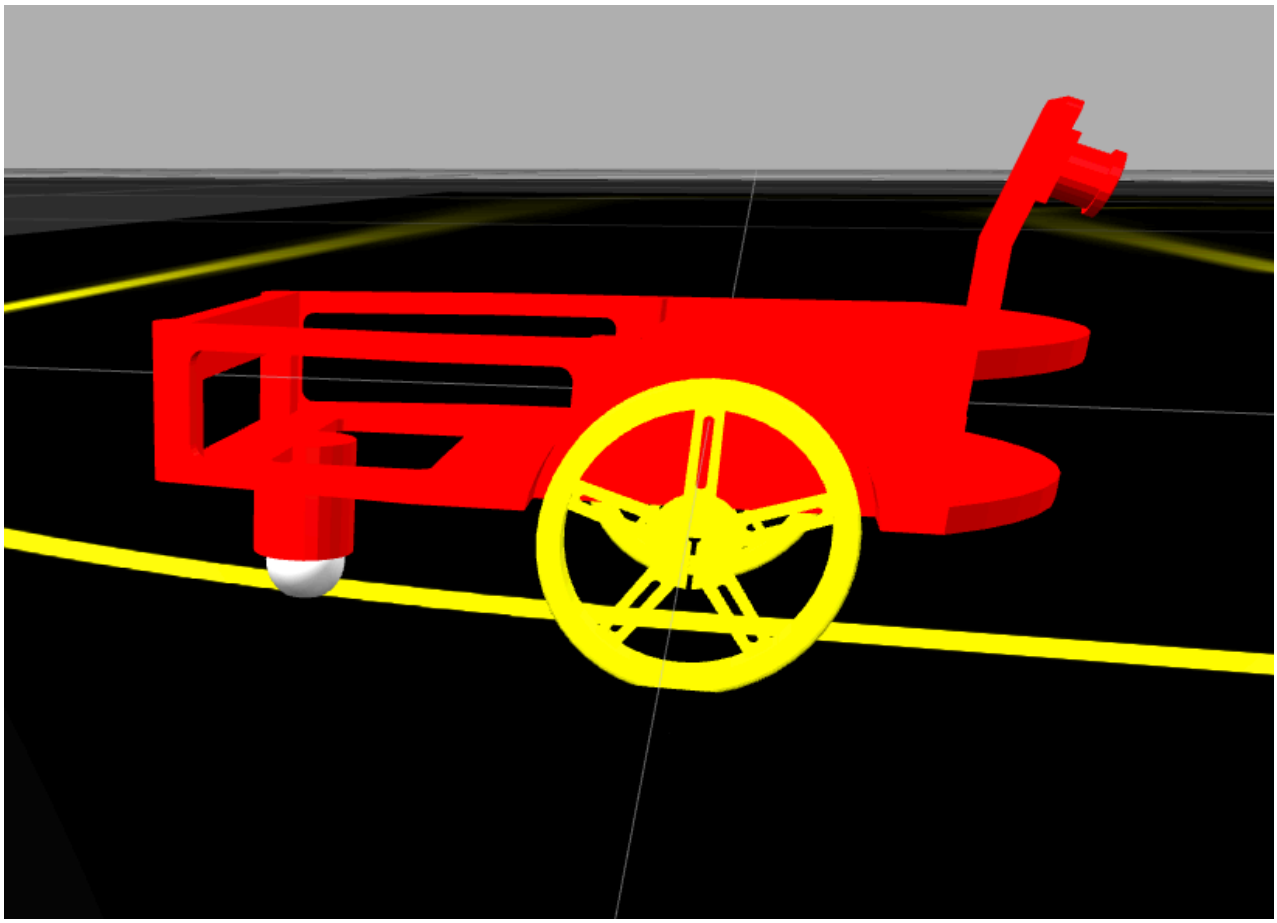


Image 2: Robot ROSRider

The camera sensor can detect objects up to 0.38 meters away from the center point of the robot. As the camera is placed towards the floor, you can track the line with this camera easily. The ability of the camera to detect far distances will make it easier for you to follow the line. The camera sensor has a horizontal viewing distance of 0.36 meters. The thickness of the line covers an area of about 6 pixels as can be seen in the picture below. You can also calculate the line thickness from this.

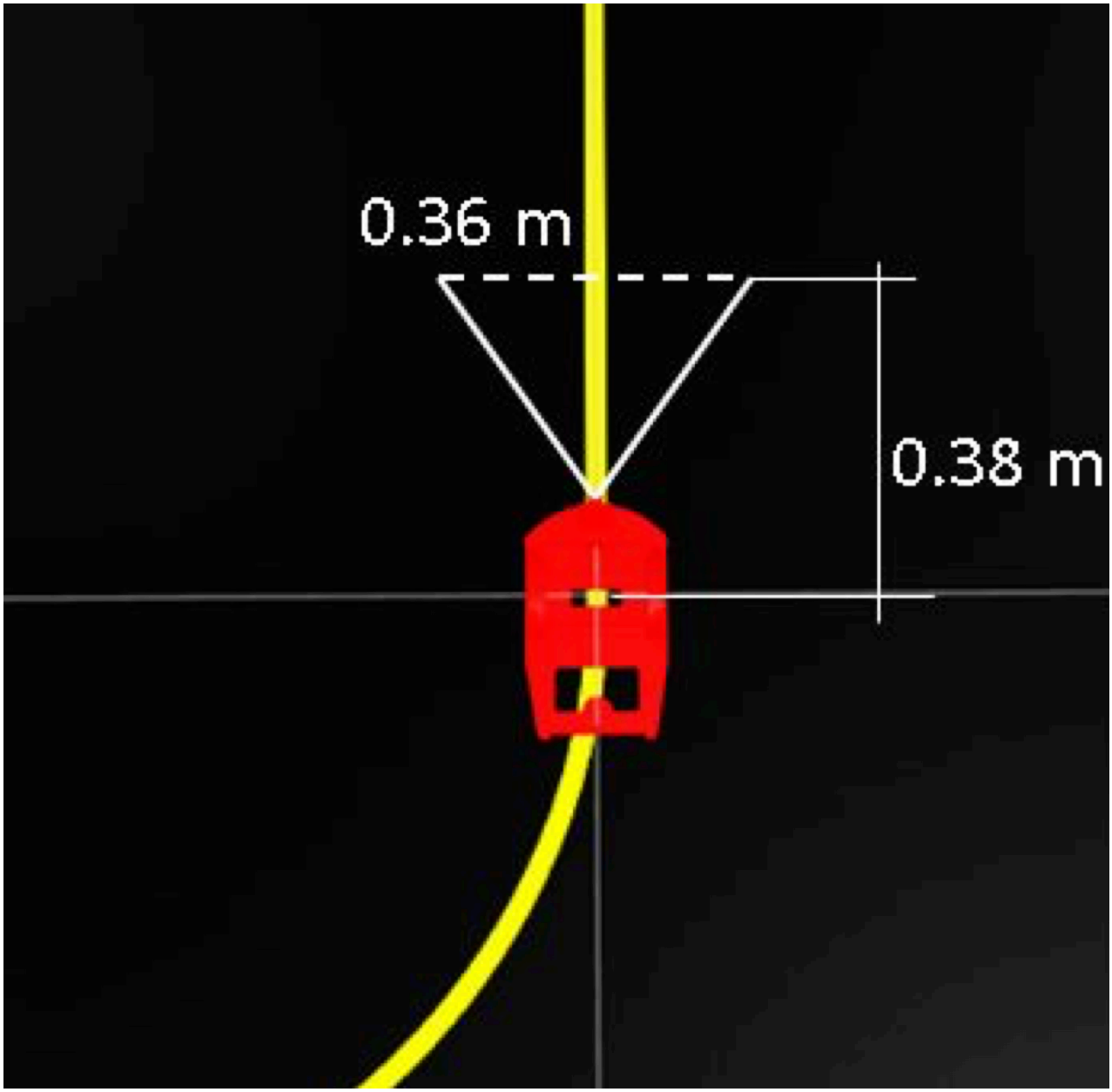
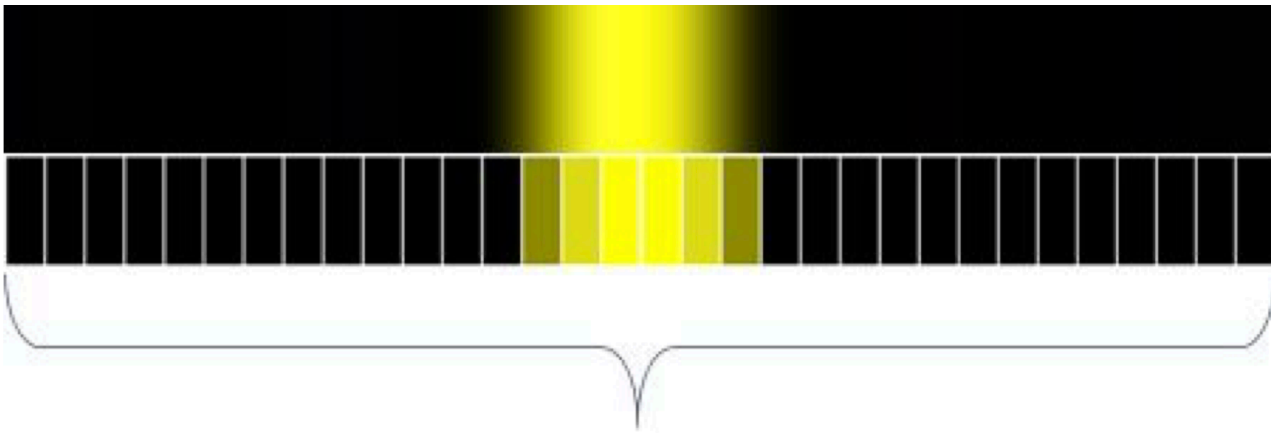


Image 3: Top View Ros Rider on the Track

RGB Camera has a resolution of 1x32, meaning that it can only scan a line and it represents that line with 32 RGB pixels. Here's a sample line, from the beginning of the simulation:



1x32 RGB Camera Output

Image 4: Track Line Representation from the ROS Rider Camera

You can get a snapshot or stream by clicking the icons next to camera of the robot in Sensors section:

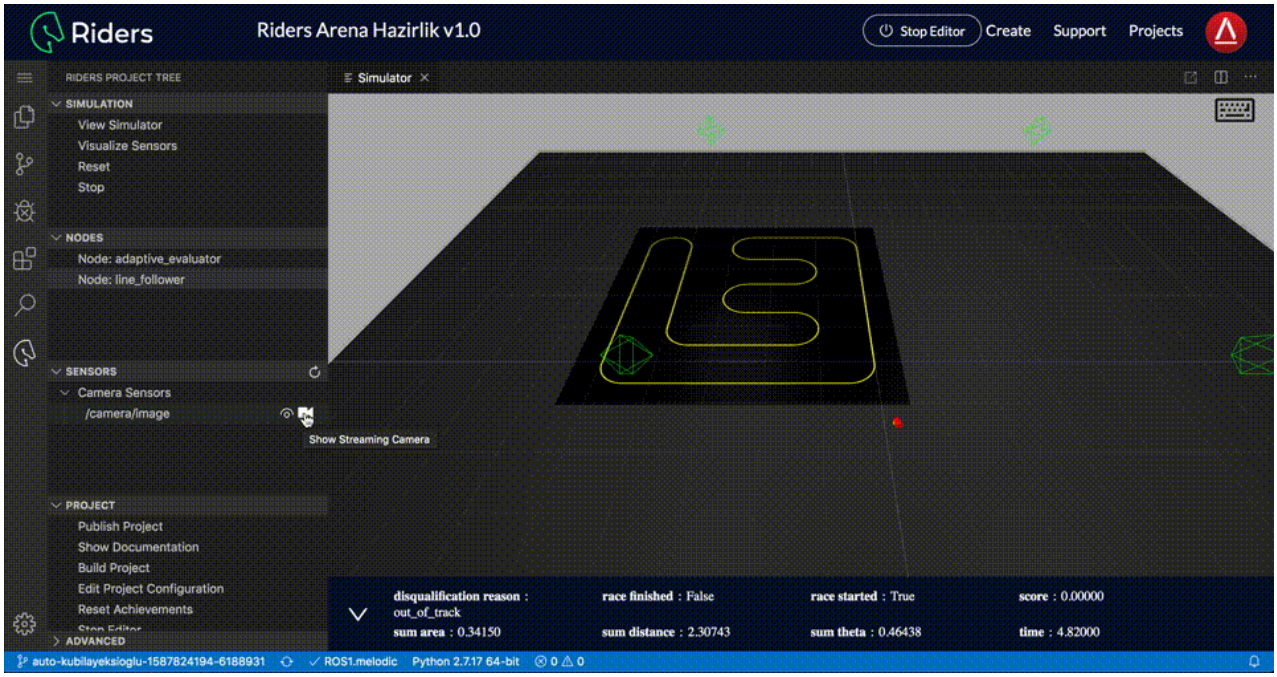


Image 5: The Brightness Data Taken from the Camera

We have a function that saves the brightness data in the pixels to a variable so that the image taken from the camera can be easily processed. In this way, we share the data from the camera in an array with you. This 32-element array (`self.sensor_row`) holds the brightness values of each pixel. You can read the brightness value in each pixel through this array and decide where the line is.

Now you know more about the ROSRider, let's talk about how you can command it.

Programming in RIDE & ROSRider API

In Robotics, each independent program running in the simulation and robot is called a "Node".

Robot's behaviors prepared on ROS (Robotic Operating System) are sent to the simulation program via program parts called "Nodes".

In this project you should see two "Nodes":

1. `adaptive_evaluator`: The program that evaluates performance of your code.
2. `line_follower`: The program you will write to command ROSRider.

Click on "Node: line_follower" in Nodes section. This should open the Python code for the program. Both in training and in Challenge, you will be expected to program this node, so that your robot can follow yellow line better.

Now that you can see the code, let's examine this file a little closer:

```
#!/usr/bin/env python
```

```
from __future__ import division

import rospy
import math

from rosrider_lib.rosrider import ROSRider

from std_msgs.msg import String
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image
from nav_msgs.msg import Odometry

from tf.transformations import quaternion_from_euler, euler_from_quaternion
```

At top, you should see the libraries, as a Python programmer, you already know this. The libraries added here are required for the ROSRider to work. Apart from these, it is possible to use other libraries. As an example, while developing your algorithm you can use any standard Python library or `numpy`. If you'd like to use an external library, contact us.

This code defines a single class called `LineFollower` and in the main function, simply calls its `.main()` function. **On each time your robot receives an image from its RGB sensor, `process_image_and_steer` function will be called by the system. Hence, you are expected to fill the remaining parts in this function.** To make things slightly easier for you, we have earmarked these places with:

```
# ENTER YOUR LINE DETECTING CODE HERE
```

and

```
# ENTER YOUR STEERING CODE HERE
```

comments. Please do not remove anything from `__init__()` function, while you may add anything that you feel like you'll need. For example, if you think you may use sensor's viewing distance in control functions, you can easily add a line to LineFollower initialization function:

```
class LineFollower:
    def __init__(self):
        # initialize robot...
        ...
        self.started = True

        # only add following line to __init__ function, rest should stay
unchanged
        self.camera_distance = 0.38
```

You are being asked to develop your algorithm and to find the line under the parts marked with the yellow rectangles above your motion algorithm that will allow the robot to quickly follow the line. For example, the code below will make the robot draw a circle.

```
# ENTER YOUR STEERING CODE HERE

self.robot.move(0.2)
self.robot.rotate(0.2)
```

Apart from these functions, there are different functions as well as you can create your own functions. Let's take a look at the functions and variables of the robot in our project:

- **self.sensor_row[]**
  Holds the brightness values of pixels retrieved from camera sensor.
  sensor_row: Array with 32 int values
- **self.robot.move(linear_vel)**
  Moves robot forward in the linear direction with a speed of the linear_vel value.
  linear_vel: float
- **self.robot.rotate(angular_vel)**
  Enables the robot to make a rotational movement in angular direction with an angular_vel value.
  angular_vel: float
- **self.robot.x**
  Holds the robot's x position information in space.
  x: float
- **self.robot.y**
  Holds the robot's y position information in space.
  y: float
- **self.robot.yaw**
  Holds the angular position information of the robot relative to the ground.
  yaw: float
- **self.robot.vel_x**
  Holds the robot's speed information in the linear direction.
  vel_x: float
- **self.robot.vel_z**
  Holds the robot's angular velocity information.
  vel_z: float

You can develop your code base using the functions and variables above. Now let's see how to monitor and debug your code while developing the algorithm and also variable monitoring and debugging. We recommend the following method to monitor and debug variables in your improved algorithm:

Variable Tracking & Debugging

While development you will like to understand the current state of your program. In Riders, just like you would do in a normal Python program, you can do that with `print` statements.

For example, click on "line_follower" node and update code as following:

```
# ENTER YOUR STEERING CODE HERE
self.robot.move(0.2)
self.robot.rotate(0.2)

print (self.robot.x, self.robot.y)
```

Then, click on "Reset" so that your changes would be applied. To view the output, right click on line_follower node and choose "Watch Node Logs". This will open a terminal, displaying output of your program in real time.

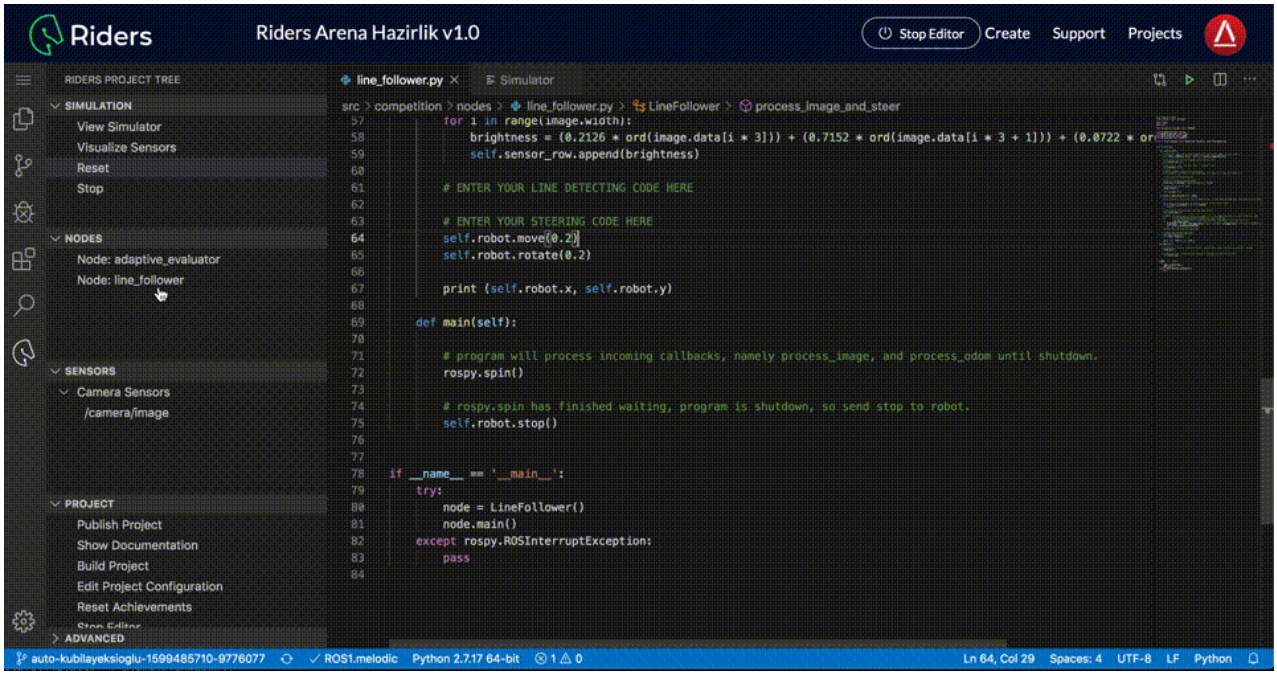


Image 6: Watch Node Logs Section

Handling Errors

First we should distinguish two different types of errors:

- Syntax Errors
- Functional Errors

Let's start with the syntax errors. Syntax Errors occur when the syntax rules of the programming language used while coding are not followed. If there is a syntax error in your code, you will face an error warning when you start or reset.

Let's place a syntax error. In line_follower replace node replace,

```
self.robot.rotate(0.2)
```

with:

```
self.robot.
```

As you might know, this is an invalid Python command. Now, click on "Reset" (or Start if you have stopped simulation earlier), you should receive an error similar to the following:

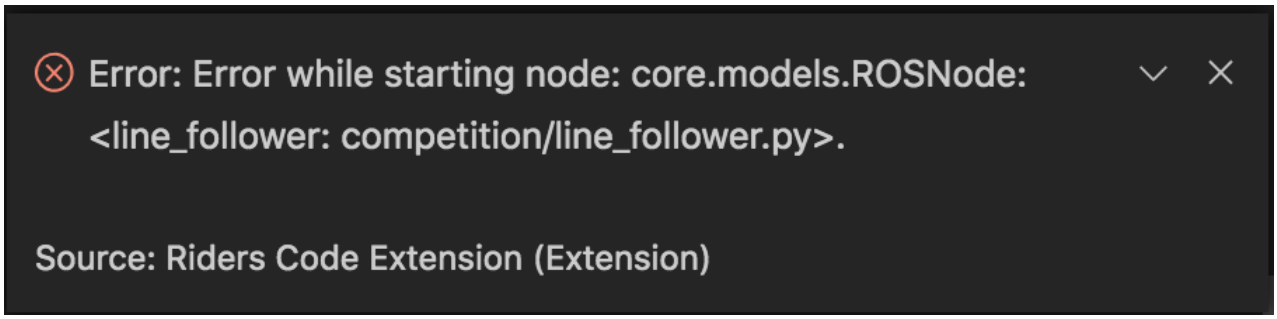


Image 7: Error Notification

To find out where exactly this error is, right click on the line_follower node and click on "Watch Node Logs". The opening window will tell us what error it is and on which line it is located in the code.

In case of a functional error, RIDE cannot show an error since it cannot know whether it was on purpose or not. But to determine the error, you should use "Watch Node Logs" functionality and proper logging in your program.

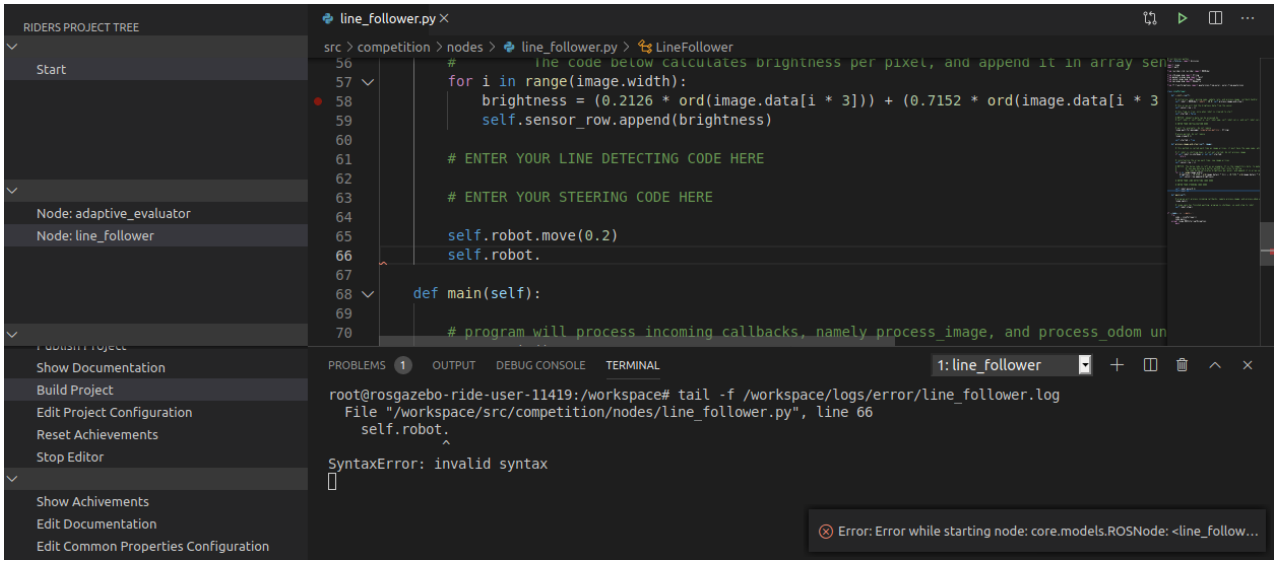
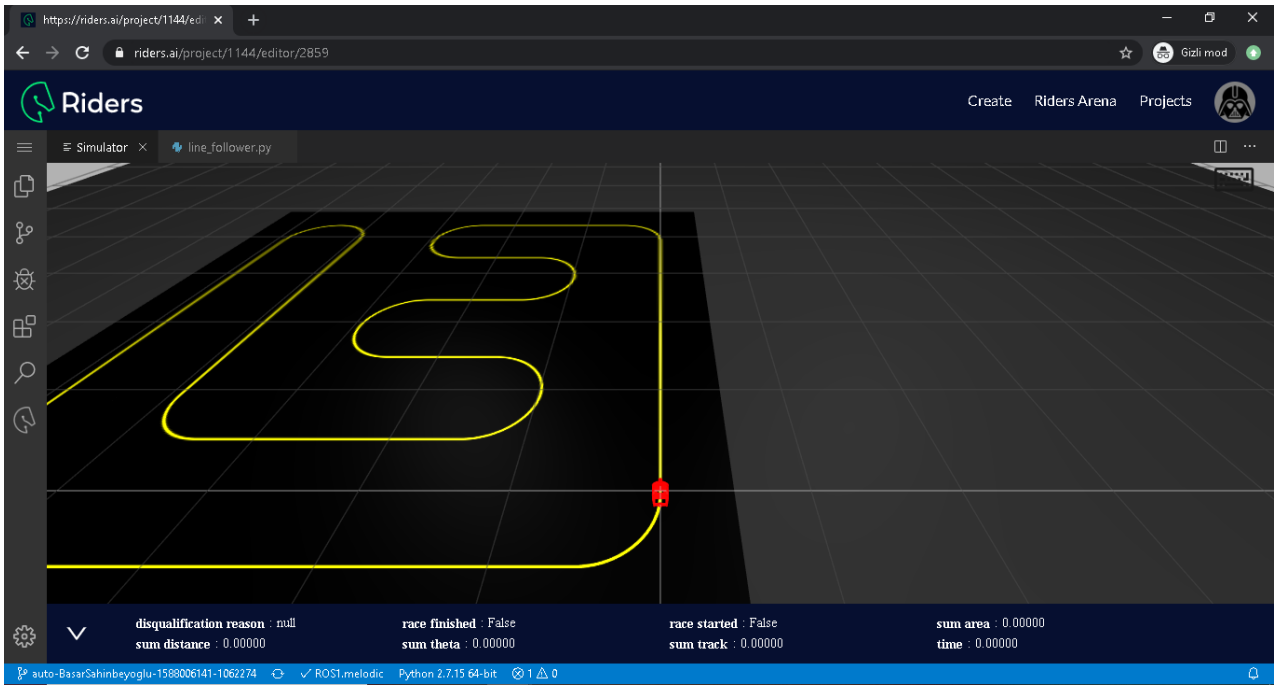


Image 8: Viewing Syntax Error

In the screenshot above, the output of "Watch Node Logs" is seen in the "Problems" tab. You can see the problems tab when you click on node: line_follower and then see the "syntax" error.

Competition Rules

There will be a live competition taking place soon and you're probably wondering how the scoring will work in this competition. After a long and detailed study, we as a company, created the scoring algorithm internally to judge submissions on automatically. Below you can find the explanation. First of all, it is useful to briefly explain the metrics in the RIDE. There is a dark blue section at the bottom of the RIDE displaying indicators of your simulation. First, let's examine these indicators:



Competition Indicators

Disqualification Reasons:

- **Null:** Not Disqualified
- **Incorrect Start:** The robot started at a different point from where it should have started
- **Out of Track:** The robot was out of 1m x 1m tiles.
- **Out of Sequence:** At least 1 track tile was skipped without passing through.
- **Out of Time:** The track was not completed within the maximum time allowed. This maximum time is **200** seconds in the practice environment and **100** seconds in the live competition environment.

If any of the above situations occur in the simulation, your code will receive a 0 score for that simulation & it means you should update your code accordingly.

race started: (True/False)

Race started will remain “False” when the simulator starts or is reset and the robot does not take any action. “Race started” is “True” as soon as the robot makes its first move.

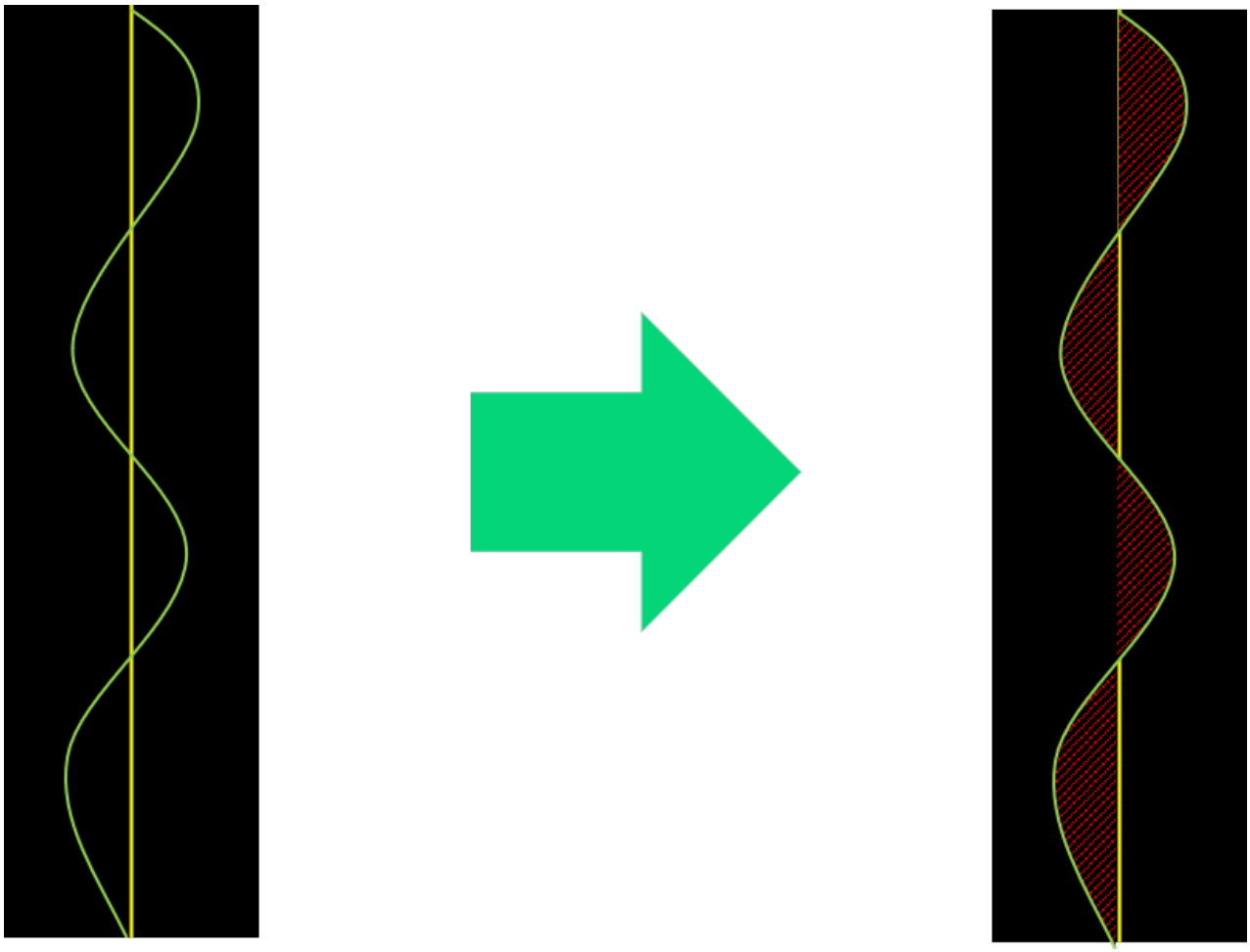
race finished: (True/False)

If the competitor completes the course without realizing the above eliminations, "race finished" becomes "True". After that, any action that robot takes will not be counted to your score.

Competition Metrics

Sum Area (m^2):

As we explained in the competition rules, the sum area is a metric that shows how far the robot deviates from the line. The area along the line that deviates from the line is continuously calculated by the scoring algorithm. The lower the sum area, the higher you will rank.



Time (s):

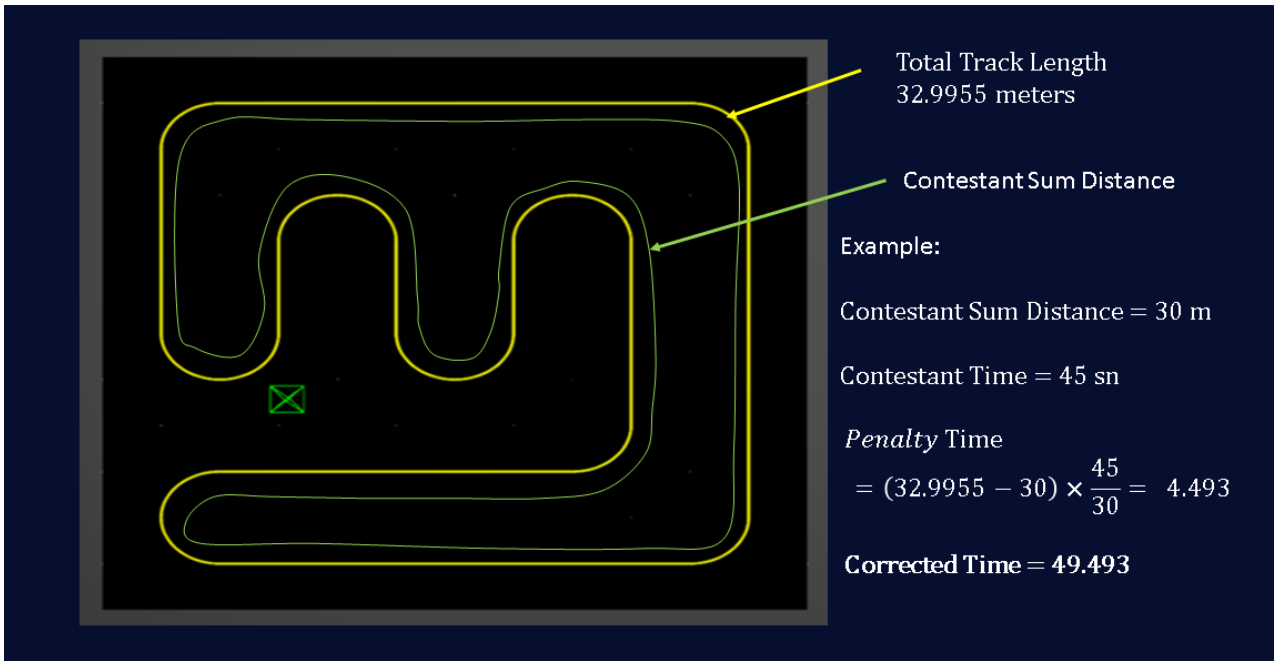
It is the metric showing how long you finished the race. So if we talk about the indicators, it is the time between race started: True and race finished: True. Consequently, the shorter this period, the higher your score will be. Sum Distance (m): It is a metric showing the distance you traveled during the race. Below we share the total length of the track with you:

Total Track Length (for practice environment) = 32.9955 metres

If the Sum Distance value you receive as a result of the race is shorter than the Total Track Length, the scoring algorithm adds the remaining distance / average speed as a penalty to your finishing time and calculates a new metric called "Corrected Time".

This metric is used in the score calculation formula.

Let's explain with an example below:



As seen in the example, although the competitor finished the track in 45 seconds, a penalty was added to the time to finish the race because she took a shorter route and the finishing time was 49,493 seconds.

Scoring formula

Competition Score calculation formula is shared below:

$$\underline{Sum\ Area \leq 0.5\ m^2}$$

$$Score = \frac{Total\ Track\ Length}{Corrected\ Time + Sum\ Area\ x\ \mathbf{10}}$$

$$\underline{Sum\ Area > 0.5\ m^2}$$

$$Score = \frac{Total\ Track\ Length}{Corrected\ Time + Sum\ Area\ x\ \mathbf{30}}$$

$$\text{Total Track Length} = 32.9955\ \text{m}$$

We basically use two different coefficients depending on whether the sum area metric is above or below 0.5 m^2. Thus, having a sum area metric below 0.5 m^2 is an important advantage. You can and you should adapt your algorithm now that you know this rule. If you have further questions, please contact us at hello@riders.ai.

Looking forward to see your works in the live competition!

Riders Team