Line Tracking with Robomaster

Foteini Rorri
rorrif@usi.ch

Bibi Arezo Massum
arezo.massum@example.com

***Abstract*—Our goal is to create a line following robot that is able to follow a path or a line segment. Once it has reached the end of the line, it should stop moving as it has completed its goal, while for the path it will simply start again.**

I. Introduction

The technologies we are using to simulate the robot and process the information gathered by the robot are: We are using ROS2 to build the robot system, with correct communication between the hardware and the sensors. We are using OpenCV for processing camera outputs of the robot, manipulating them and using them for out python implementation. We are using numpy for manipulation of the camera outputs combines with OpenCV. For the simulations we are using Coppelia Sim and the robot we are using is the RoboMasterS1 that has been modified to include three camera sensors, two perspective and one orthogonal, for the path tracking.

II. Simple: Line Following

A. Setup

The first task was to create some simple paths that would then be used to test the robot. We started by just creating a simple short open path on coppelia. We chose the path color to be red so that the difference in intensity between the path and the plane would be noticeable. This would be crucial later for the our path following algorithm to be able to distinguish where the path begins and ends. Then we modified RoboMasterS1 with the following camera configuration.

The three cameras are mounted mounted on the chassis of the robot. The middle camera is orthogonal and with a short range of view to ensure that it only sees the path just below the robot 1. The left and right camera sensors are perspective and they cover the sides of the path. This assures that the robot is always aligned with the path but it could also be used later on to detect upcoming curves in a closed path(circuit).

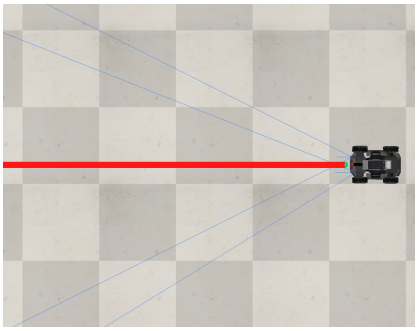


Fig. 1: Initial testing scene with modified RobomasterS1

The vision of the robot with the three cameras in front of a straight line path is the following 2

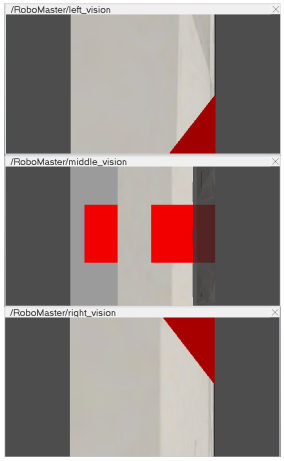


Fig. 2: Robot vision with 3 cameras in front of straight path

After successfully modifying the model, the lua script was changed to test the functionality of the sensors. In fact, the cameras were working and there was a clear distinction in the intensities of the path and the plane as viewed by the camera.

Having the sensors working and connected to our python script, we moved on to the line following algorithm. For a straight line, which is our base case, we set a threshold intensity of 0.6. If the robot detected with the middle sensor an intensity less than 0.6, then it would be on the line, if greater than 0.6 then it would be on the plane. This would allow the robot to stop once the path has ended.

Implementing this after fixing all the errors was easy and our robot was able to follow the red path until it ended and then the robot would stop. It was tested with straight lines of different lengths and orientations, and it always worked. Then we proceeded to the task of circuits in which the robot would have to steer based on the turns of the circuit. To complete that, since a circuit is a closed path, the robot would have no need to stop, and so we removed the previous logic of the robot stopping once the path ends.

Additionally, it proved a bit tricky to have two perspective vision sensors (right and left) as they captured too much of the plane, which in turn resulted in intensities that were too close to the plane intensities. Thus, we replaced them with orthogonal ones instead. 3

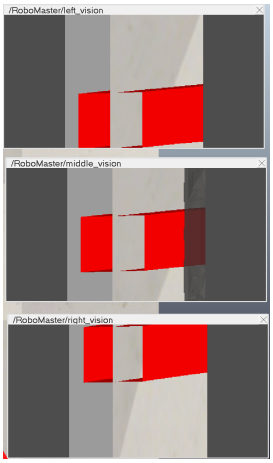


Fig. 3: Revised robot vision

III. Advanced: Line Following

A. Logic and Implementation

For the circuit follower steering logic, we defined a sf steering factor and the difference between the intensities of the right and left sensors. Then, we defined the angular velocity of the robot as their product.

$$\omega = sf * (right\_intensity - left\_intensity)$$

We also added a limit on the change of the angular velocity to avoid unnecessary or overly sharp steering (max_steer). This parameter is also controller dynamically based on the intensities that the robot sees, to assure that it has always enough steering to make the turns. Thus, the angular velocity is always in the interval of [-max_steer, max_steer] factor.

While this worked very well for simple circuits with soft curves, the robot would go off track on the sharp turns. To fix this, we added a velocity control with a base and max velocity measure and defined the actual velocity of the robot to be the normalized combination of its base velocity minus the difference between base and min. So, in the case that the robot misses the line in one of the three sensors because it is going to fast for the turn, it would then use the modified velocity as defined above to slow down and take the turn. With this modification the robot successfully completed the circuit. The only issue was that it did it too slowly. The difficulty now became to balance the tradeoff between speed and accuracy. Trying to make it go faster in the straights, it then started to miss the turns if they were very sharp, or if the straights had a slight curvature then it would go very slow. Since our logic was correct and working, the only thing left was to finetune the parameters for the angular and linear velocity and add some more dynamic controllers to help adapt the speed of the robot based on the track shape.

The testing played a crucial role in how we adapted the algorithm and the parameters. For simple circuits with straight lines and open turns the model almost always performed well and made the whole lap. Regardless, we tried to test it with more complex circuits that have shaper turns and not many straights. This allowed us to create a model that worked even in the ”worst” cases. Of course, in order for the robot to be able to make the sharp turns we had to sacrifice to some extend speed. This model is not that agile so even with dynamic speed assignments it is no formula 1. Additionally, there is a radial limit to which the robot can turn due to its intrinsic mechanism, so our turns never exceed that limit but they are the closest possible to it.

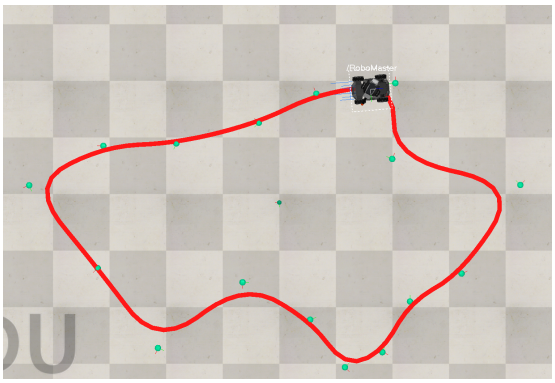


Fig. 4: Circuit 1

This turn 5 was extremely difficult for the robot to follow with the modular speed algorithm. The turn itself is very sharp and with varying radius. While the robot was able to follow it for the most part by reducing speed considerably and turning more, it was still losing the track on the exit of the curve. To make it work we mainly tweaked sf and max_steer, to make sure that the change in the angular velocity when the robot is on a steep turn is sufficient.

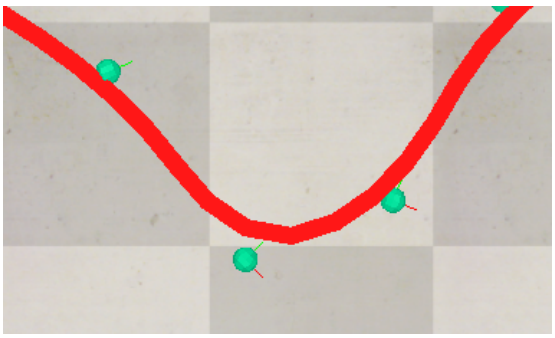
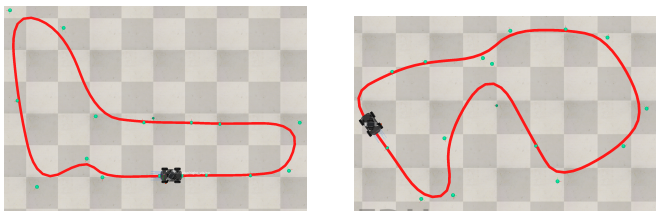


Fig. 5: Tricky Turn

With these modifications the robot successfully completed all the test circuits and the difference in speed between the straights and turns was very evident. Our algorithm was working well for all instances.



(a) Monza Circuit (b) Silverstone Circuit

Fig. 6: Testing circuits

We tested the model (robo2.1) in these three circuits 4 6a 6b with varying speeds. Our algorithm was able to support speeds for the robot from $v = 0.10$ to $v = 0.35$ as long as sf and max_steer were modified accordingly to ensure the robot would be able to turn. After $v = 0.35$, the increase in speed would demand too much of an increase in the turning factors making the robot turn very sharply. It became too sensitive to small changes in the curvature of the circuit and even for small turns it would oversteer. The behavior was too erratic so we concluded with $v = 0.35$ as the maximum speed for our model to perform well.

B. Problems and Errors

After modifying the Lua script to send the input of the camera sensors, we got the error "ERROR conn.py:107 $scan_robot\_ip : exceptiontimedout$". After further investigation, we found the root of the problem. The readVisionSensor() function that was being called for all three sensors was too costly, causing the simulation to slow down and eventually time out. Coppelia was too busy trying to process the images and so it would stop the communication with ROS. To fix this, we added a count to the script so that readVisionSensor() is only called every x steps.

IV. Advanced: Dual Robot Path Following

The next phase of our project involved setting up and controlling two robots in the same simulation environment. The aim was for both robots to follow parallel lines on the floor, each tracking a different path but working within the same scene.

A. Setup of Dual Robots

To make this possible, several changes were required:

- **Unique robot identification:** Each robot was assigned a distinct name in the simulation to ensure there were no conflicts between their configurations.
- **Sensor configuration:** The vision sensors of each robot (left, middle, and right) were renamed to avoid overlapping data streams when running both robots simultaneously.
- **Serial numbers and network:** Unique serial numbers were assigned to each robot, and different IP addresses were configured to allow simultaneous connection to the system without interference.

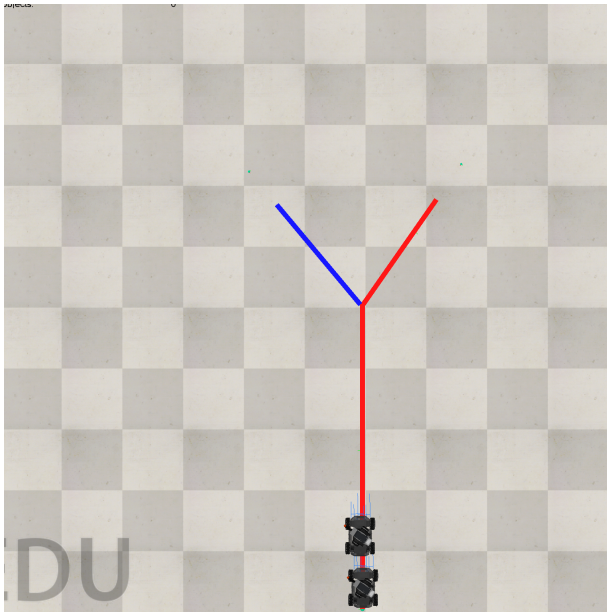


Fig. 8: Fork path

These adjustments ensured both robots could operate together without causing communication errors or sensor conflicts.

B. Execution of the Task

Once both robots were configured, we ran them with separate commands to execute the path following task. Each robot was launched with a different ROS2 run command, specifying unique topic remappings for control and sensor inputs. This allowed both robots to operate in parallel without conflict.(7).

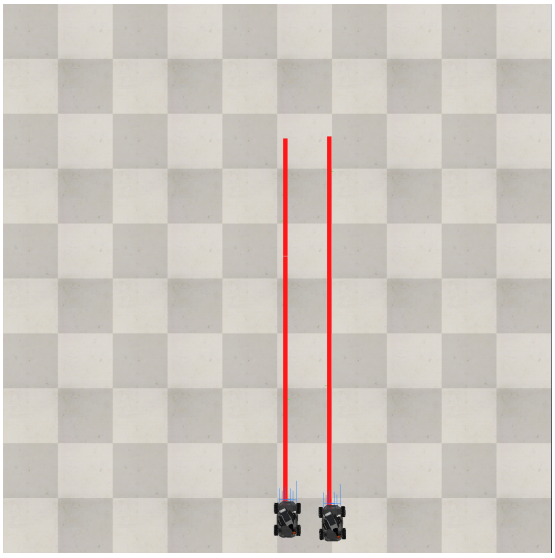


Fig. 7: Dual robots positioned at the starting point.

V. Dual Robot Fork Detection with Speed-Based Decision

After successfully configuring dual robots for path following, we focused on improving their behavior at fork points in the path. The objective was to have both robots initially follow the red line, with Robot 1 always continuing on the red path after the fork, while Robot 2 would make a dynamic decision based on its speed relative to Robot 1.

A. Logic and Implementation

The core logic was designed as follows:

- **Robot 1**: Continuously follows the red line. Upon approaching the fork point 8 where the red line diverges, it steers right to reacquire the red path and realign itself.
- **Robot 2**: Also begins by following the red line. However, if its speed is detected to be higher than that of Robot 1, it steers left at the fork to transition to the blue path. If its speed is not greater than Robot 1's, it continues on the red path by steering right at the fork.

The implementation utilized real-time line detection from each robot's middle vision sensor. Image moments were calculated to determine the centroid of the detected line (red or blue), and the line's horizontal offset from the camera's center was used to guide steering decisions. Additionally, odometry data was used to compare the linear speeds of Robot 1 and Robot 2 for the dynamic decision-making process.

B. Results and Challenges

While the implemented logic successfully introduced speed-based decision-making for fork navigation, several issues were observed during testing:

- **Robot 2 turned too early**: Sometimes it started turning left before it was even near the fork, which made it leave the red path too soon. This led to unnecessary corrections and instability.
- **Delayed reaction by Robot 1**: Conversely, Robot 1 sometimes began steering right too late, it turned right only after it had already passed the fork, which made it lose track of the red path for a bit.
- **Tuning sensitivity**: Fine-tuning the thresholds for line center offset and speed comparison was challenging. Small fluctuations in vision sensor data and odometry readings affected decision accuracy, which made it hard to get smooth behavior.

This version of the system showed that we can use both camera data and speed data to handle forks in the path for two robots. Even though the steering didn't always happen at the right time because of tuning issues and sensor glitches, it was a good starting point for building a smarter path-switching system.

VI. Conclusion

Throughout this project, we developed a multi-step system for line tracking and path following using RoboMaster robots. We started by building a basic line following node, which used vision sensors to detect a red line and adjust the robot's steering and speed. This initial implementation provided a solid base for understanding how the robots could follow paths with simple logic.

Moving forward, we enhanced the system to handle more complex circuits with curves and forks. The advanced task introduced dynamic steering adjustments and speed control, allowing the robot to slow down during sharp turns and maintain a steady pace in straight sections. Although this made the system more flexible and robust, it also introduced challenges such as balancing speed and accuracy, as well as tuning the control parameters.

Overall, this project showed how we can implement a simple line-following robot to a more advanced setup with dynamic steering and two robots working together. It demonstrated the strengths and challenges of using camera data and speed measurements to guide robots in a simulated environment. The system worked, but tuning it for perfect steering and timing was difficult because of small sensor fluctuations and delays. Future improvements could focus on making sensor readings more reliable, adding ways to predict the path ahead, and improving decision-making for smoother and more accurate robot movement.

VII. Technicalities

For the videos we provide a google drive link as they are too heavy to upload. https://drive.google.com/drive/folders/1rSRsId-UVfHFy_lgT_pjvpas9ABkRH4q?usp=share_link

We have three videos for the three circuits for $v = 0.25$ and one for $v = 0.35$. All three are able to execute the path for $v = 0.35$ but the videos were redundant.

There is also a video of the straight path being followed. Since this is the simplest version of our implementation, the code for it is commented out in the file line_follower_node.py under the comment "for straight line paths". To test it simply uncomment the code below it, but it is unnecessary as it already manages to complete circuits.