A Heuristical Approach to Autonomous Driving

Chincha León, Marcelo Andres
Facultad de Computación
UTEC
Lima, Peru
marcelo.chincha@utec.edu.pe

Carranza Bueno, Lucas Facundo
Facultad de Computación
UTEC
Lima, Peru
lucas.carranza@utec.edu.pe

***Abstract*—This document describes the implementation of a heuristics driven approach to Autonomous Driving, specifically in the context of the TurtleBot3 Burger robot.**

***Index Terms*—autonomous driving, deep learning, lidar, turtlebot, ros, fsm**

I. Introduction

This document describes the details of the Heuristical Approach used to create a self-driving TurtleBot3 robot, including the conceptual design and the implementation details. The result is a robot that was capable of completing multiple laps of a closed circuit at a decently fast pace, while using a very simplified architecture and with no previous knowledge of the circuit.

II. Architecture

In the context of class **CS5364**, the following Perception Framework for Autonomous Driving was presented:

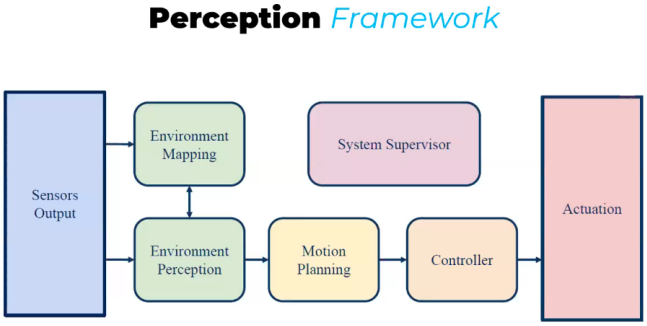
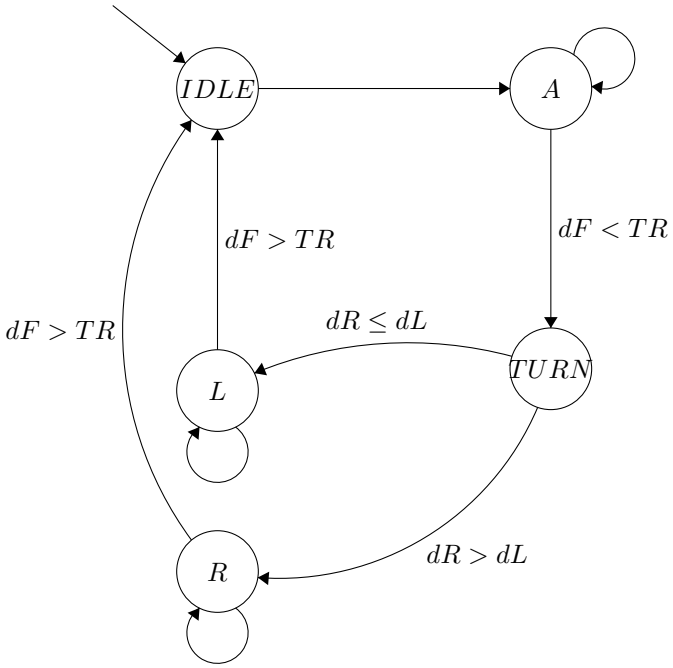


Fig. 1. Complete Perception Framework

However, the smaller scope of our TurtleBot navigation project doesn't require most of the modules. By using a heuristics-driven approach, we can ignore any mapping and planning modules. Also, since we will be only testing our robot in a controlled environment, we can ignore any supervising modules.

Therefore, we will only focus on the **perception** and **controller** modules, as well as the Heuristic itself.

This implementation in particular makes use of a Finite State Machine to ensure smooth changes between correcting and advancing maneuvers. Here is a diagram representing the FSM Logic:



- dF: Front Distance
- dR: Right Distance
- dL: Left Distance
- TR: Distance Threshold

III. Implementation

Link to github repository.

A. Setup

In order to have a working endpoint to use the robot for Autonomous Driving, a setup process is required. For this project, the manufacturer's instructions were followed to install ROS2 Humble and its dependencies.

By using requests and subscribers, similarly to how the built-in modules work, we can make code that can work both locally (run inside the bot) and remotely (from a remote pc in the same LAN as the bot).

B. Controller

In order to send movement orders to the robot, we can use the `Twist()` module, similar to what the built-in `teleop_keyboard` module uses (which is what our implementation is based on). This sends a package that contains linear and angular speed values, which allows the robot to turn and go forward.

The robot's actions (e.g., moving forward, turning) are done based on the current FSM state. Here is an explanation of each state's movement logic:

- `ADVANCE`: Move forward with optional small angular adjustments to keep centered.
- `LEFT/RIGHT`: Turn in place in the respective direction.
- `IDLE`: Don't move (yet).

```
def processState(state, pub, laser):
    twist = Twist()
    dist = laser.get_values()

    if dist["front_right"] > 0.78:
        a = -TURN_SPEED * 0.18
    elif dist["front_left"] > 0.78:
        a = TURN_SPEED * 0.18
    else:
        a = 0.0

    if state == 'ADVANCE':
        twist.linear.x = SPEED
        twist.angular.z = a
    elif state == 'LEFT':
        twist.linear.x = 0.0
        twist.angular.z = TURN_SPEED
    elif state == 'RIGHT':
        twist.linear.x = 0.0
        twist.angular.z = -TURN_SPEED
    elif state == 'IDLE':
        twist.linear.x = 0.0
        twist.angular.z = 0.0

    pub.publish(twist)
```

C. Perception - LIDAR

The LIDAR sensor is the most important perception sensor used in this project. The raw data from the LIDAR is received through a subscriber to the `/scan` endpoint:

```
create_subscription(LaserScan, "/scan",
        laser_scan_callback, qos_profile)
```

This raw data is normalized into a 360-degree range. This is what allows the definition of angle ranges like `'front'` or `'front_left'`.

```
def normalize_ranges_to_360(self, msg):
    angle_min = msg.angle_min
    angle_max = msg.angle_max
    angle_increment = msg.angle_increment
    message_range = msg.ranges
    norm_ranges = [float('inf')] * 360
    angle = math.degrees(angle_min)

    for dist in message_range:
        if not math.isnan(dist):
            index = int(round(angle)) % 360
            norm_ranges[index] = dist
        angle += math.degrees(increment)
    return norm_ranges
```

D. Perception - Camera

Our architecture was also able to include a camera sensor, where the image is captured using `opencv` and it is processed with the MiDAS Depth Perception Model from the PyTorch Library.

The camera is also an important sensor, as it allows greater detail, and in the context of the TurtleBot3, allows to detect and avoid low obstacles that are beneath the LIDAR's plane.

However, due to the heavy inference and network cost, the camera module was discarded for the competition architecture.

Future work: Get a responsive model with both LIDAR and Camera Depth Perception.

E. Heuristic - FSM Logic

The FSM determines the robot's behavior based on the sensor input and the previous state, in order to ensure coherent motion with this limited architecture.

```
def nextstate(state, laser):
    dist = laser.get_values()

    if state == 'IDLE':
        state = 'ADVANCE'
    elif state == 'ADVANCE':
        if dist['front'] < THRESHOLD:
            state = 'TURN'
    elif state == 'TURN':
        if dist['front_left']
            > dist['front_right']:
            state = 'LEFT'
        else:
            state = 'RIGHT'
    elif state == 'LEFT':
        if dist['front_right'] > THRESHOLD:
            state = 'IDLE'
    elif state == 'RIGHT':
        if dist['front_left'] > THRESHOLD:
            state = 'IDLE'
    return state
```

Summary of next state logic, given current state:

- `IDLE`: Begin moving.
- `ADVANCE`: Keep advancing while nothing directly in front.
- `LEFT/RIGHT`: Keep turning until front opposite side is clear.

IV. Testing and Results

The robot was tested in a closed circuit, alongside other robots, with different

The results were that our model was amongst the most capable models, if not the most capable. Our robot had multiple arrivals in first place, as well as two full laps completed back-to-back.

It was also very consistent at completing the circuit, showing the model consistency and reliability inside the controlled environment.

The only time where human intervention was required, was during start-line collisions with other robots, as the model had difficulty avoiding other stopped robots that were very close. However, all of the other robots also suffered from this, possibly due to the tight layout of the circuit.

V. Conclusions

After having done this project, we have come to the following conclusions:

- Simplifying the Architecture of an Autonomous Driving model can ease the implementation and provide more responsiveness, at the cost of limiting the number of scenarios where the model is effective.
- A fine-tuned Heuristics approach can make a model perform really well on controlled environments, but it will not have the generalization capabilities that Deep Learning models offer.
- It to use multiple sensors for Autonomous Driving, not only as a fail-safe, but also to have the capabilities to perform well in more scenarios.

References

[1] TurtleBot Manual, "TurtleBot Manual," Available online: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[2] Open Robotics, "ROS 2 Humble Hawksbill Documentation," Available online: https://docs.ros.org/en/humble.

[3] Reiner Birkl, Diana Wofk, Matthias Müller, "MiDaS v3.1 – A Model Zoo for Robust Monocular Relative Depth Estimation," Available online: https://arxiv.org/pdf/2307.14460.