UNSW

Faculty of Engineering
School of Computer Science & Engineering

COMP3431
Robot Software Architectures

Project 2 Individual Report

Group: Bob Ros

Course Coordinator: Claude Sammut
Due Date: 29/11/2020

Dan Nguyen
z5206032

Abstract

Autonomous driving is widely considered to be the next step in revolutionising the automobile industry in the foreseeable future. However this technology is far from mature and requires improvements in safety and reliability before autonomous driving can be commercialised. The purpose of this research paper is to explore a simple robotic software architecture (in accordance with the learning objectives of COMP3431) to handle the autonomous driving of a TurtleBot3 Waffle Pi in real-time, sparse-road, constant-illumination conditions using a Raspberry Pi camera sensor and OpenCV2 library. The autonomous driving system was adequate in the project demonstration for COMP3431 and its performance is evaluated with some suggestions for improvements.

Contents

List of Figures

1 Introduction

Autonomous driving is the ability for a vehicle to drive itself with minimal or no human interaction, and is considered to be the next leap in innovation for the automobile industry. Research into this technology is pushed by its benefits of increased traffic efficiency, increase in quality of life, and potential reduction of traffic accidents. For autonomous driving to be feasible, a software architecture must be described. In the most simplistic form, sensors are required for the vehicle to perceive its environment, the sensor data processed then a decision must be made to actuate the vehicle [1]. However in reality, autonomous driving software architectures are complex with various types of sensors and detailed vehicle control planning. This has applied limitations on the technology as it is not mature for commercialisation where accuracy, reliability, speed, and safety are the limiting key factors.

The development of autonomous driving requires the knowledge of robotic software architectures where this report explores the use of a simple 2-layered robotic software architecture and discusses the appropriateness and fundamental layers of this stack in achieving autonomous driving on the Turtlebot3 Waffle Pi in a controlled environment.

2 Environment

The environment is shown in Figure 1 below.

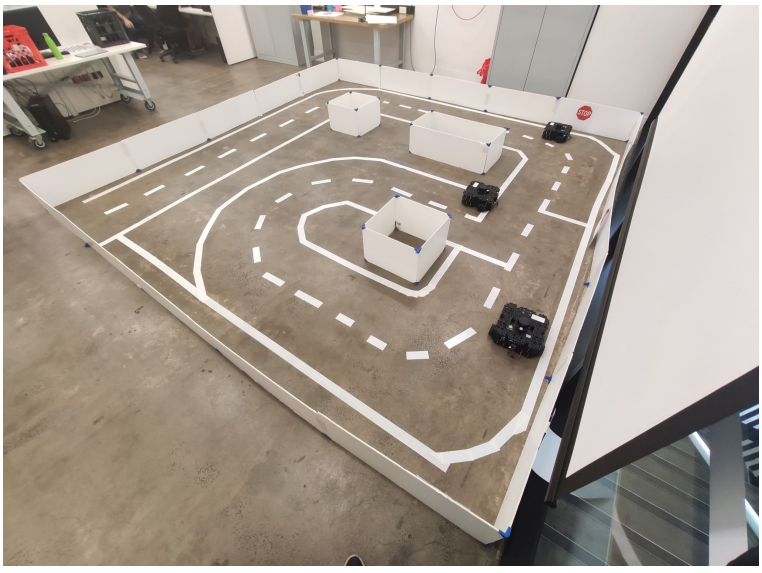

Figure 1: Turtlebot Environment

It can be assumed that there exists white walls bounding the environment, the lane markers are thick and white, stop signs are red, and turtlebots are black. It is also assumed there is constant illumination, there are no dynamic environmental changes, and road conditions are sparse and flat.

3 Robotic Software Architecture

From the summary of the environment in Section 2, the computer vision objectives can be outlined as:

- Lane detection
- Turtlebot detection
- Stop sign detection

The technologies that achieve the computer vision objectives can be abstracted as the vision processing layer of the autonomous driving stack. The output of this stack is an interpretation of the world which is inputted to a central decision-maker (robot control layer) which encapsulates a state machine representation and commands for actuation. The interactions between the world and software layers is visualised in Figure 2 below.

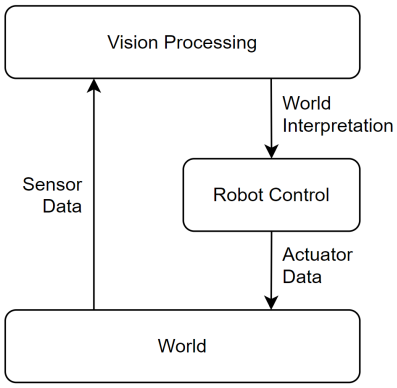

Figure 2: Autonomous Driving Software Architecture

The autonomous driving stack can be broken down into a publisher-subscriber pattern using ROS framework (Figure 3 below).

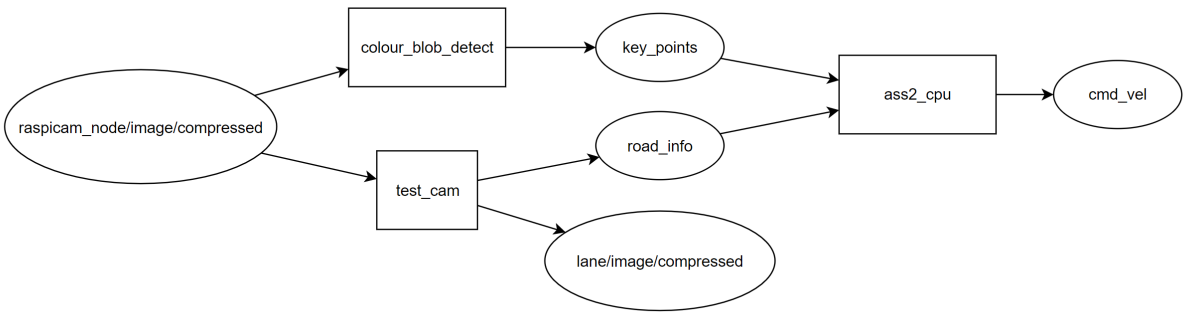

Figure 3: Autonomous Driving ROS Node Diagram

The autonomous driving stack consists of three nodes:

- **colour_blob_detect**: Handles the detection of stop signs and turtlebots.
- **test_cam**: Handles the detection and interpretation of lanes.

- **ass2_cpu**: Encapsulates the state machine representation.

And consists of four fundamental topics:

- **raspicam_node/image/compressed**: Contains images from a Raspberry Pi camera.
- **key_points**: Contains a boolean that is true for successful detection of a stop sign or turtlebot.
- **road_info**: Contains lane metrics.
- **cmd_vel**: Contains the velocity message to actuate the Turtlebot.

The lane/image/compressed topic is for visual debugging using RQT.

3.1 Lane Detection

The ROS node, "test_cam", subscribed to "raspicam_node/image/compressed" and published a custom ROS message type RoadInfo to "road_info". This node handled the detection and interpretation of lanes where the pseudocode is [2]:

```
function detectLanes(image):
    edges = get_canny_edges(image)
    edges = dilate_edges(edges)
    road_info[0] = (1 - (edges[:, 0] != 0)[::-1]).argmax()
    road_info[1] = (edges[-1] != 0).argmax()
    road_info[2] = (1 - (edges[:, 320] != 0)[::-1]).argmax()
    road_info[3] = (edges[-1] != 0)[::-1].argmax()
    road_info[4] = (1 - (edges[:, -1] != 0)[::-1]).argmax()
    return road_info
```

RoadInfo which is of type uint32[] representing distances in the following diagram in Figure 4 below. Each integer in the diagram represents an index of the array (from 0 to 4).

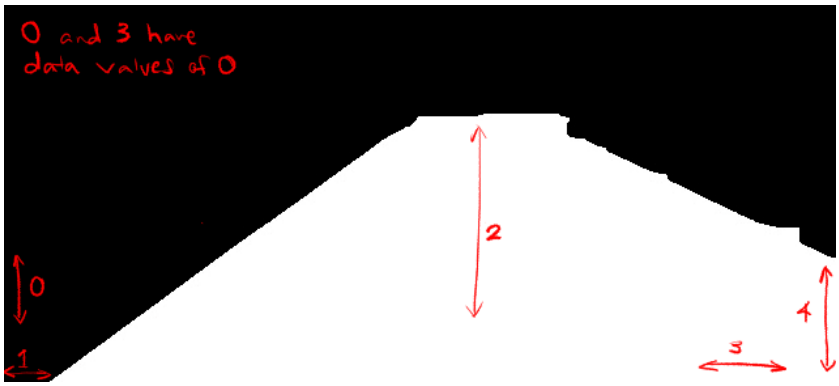

Figure 4: Road Information Diagram

Intersections are trivially detected if the lane height is large and constant in thickness, that is:

```
at_intersection = true if road_info[0] - road_info[4] -> 0 else false
```

3.2 Blob Detection

The ROS node, "colour_blob_detect", handled the detection of both stop signs and turtlebots to simplify code complexity as the computer vision technique to detect stop signs and turtlebots were the same i.e. using OpenCV2's simple blob detector. This simple blob detector was considered for its ease of use which allowed rapid development and greater confidence in code correctness. It is also guaranteed that no other objects in the environment will be red or black other than stop signs and turtlebots respectively.

The "colour_blob_detect" node subscribed to "raspicam_node/image/compressed" and published a boolean to "key_points". With each subscriber callback, the obtained image was decompressed and passed into the functions: detectTurtlebot (see Appendix A) and detectStopSign (see Appendix B). These functions followed roughly the same function structure:

1. Apply mask to detect colour of choice.
   (a) To detect the red stop sign, the image was first converted to an HSV image which was a better colour model for masking red.
2. Invert the mask.
   (a) This step depends on whether the simple blob detector detects black or white blobs. It may be omitted.
3. Fill holes in the mask.
   (a) For detectStopSign, a morphological open then morphological close was performed on the mask to fill the "STOP" text.
   (b) For detectTurtlebot, a flood fill was used to fill the spaces in the turtlebot.
4. Get keypoints (i.e. blob coordinates and size) from applying OpenCV2's Simple Blob Detector on the mask.
5. Parse through the keypoints for a valid blob representation.
6. Return the boolean on detection of valid blob.

3.3 State Machine Representation

The robot control layer which consists of the state machine representation is encapsulated in a single node, "ass2_cpu". The state machine is visualised in Figure 5 below.

If the boolean returned in "colour_blob_detect" is true then blocked is true. If an intersection is detected then at_intersection is true, and a suitable direction is picked to cross the intersection. If the intersection is not detected (i.e. the turtlebot is on a directed road) then the direction picked is trivial. The moving state publishes the linear and rotational velocity values to the "cmd_vel" topic which is processed by the ROS navigation stack.

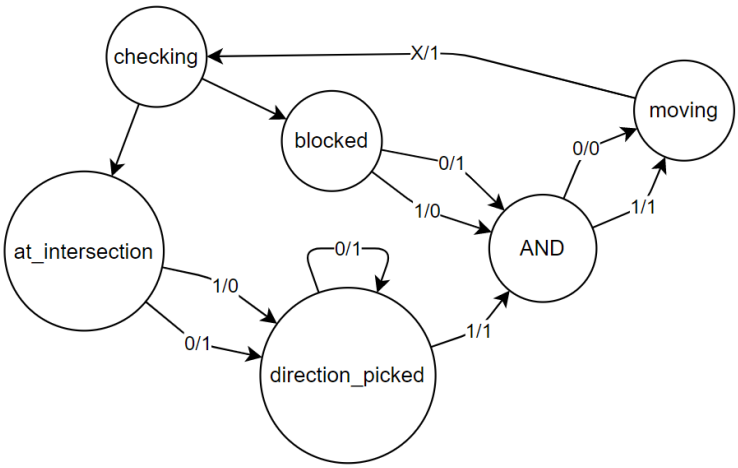

Figure 5: State Machine Representation

4 Evaluation

The results of the demonstration of the autonomous driving software stack have not been released, therefore the evaluation of the autonomous driving stack can only be discussed qualitatively. The software architecture was successful in achieving autonomous driving and performed well in directed roads. Intersection handling had some difficulties causing the turtlebot to turn too early as a result of the method of detecting intersections. This could be solved by processing the stop line of the intersection instead of applying a mask over it, therefore losing visual information. The detection of stop signs and turtlebots was excellent and integrated well with the state machine intersection handler (especially in simulation). Static test results of the turtlebot and stop sign detections can be found in Appendix C and D respectively.

5 Personal Contribution

My contributions to this project was:

- Research into a suitable detection method of stop signs and turtlebots using the OpenCV2 library.

  OpenCV2's simple blob detector was an extremely simple and effective tool to detect stop signs and turtlebots. The detection of blobs was also a trivial problem, therefore no further research was conducted.

- Design, development, and testing of "colour_blob_detect" node.
- Integration of "colour_blob_detect" with the "ass2_cpu" node.

It was difficult to incorporate my teammate's research and works involving state machine representation and lane detection as I did not work on those systems - inherent to a task allocated

work-style. Therefore, this report does not offer the best detail of the other systems.

6 Conclusion

Autonomous driving has clear benefits to how society travels by automobile but requires extensive research and development before the technology can be widely-commercialised. This report discussed the design of a 2-layered software architecture which was fundamental and simplistic, and covered vision processing in a high-level description. This software architecture was then verified through demonstration, where the architecture achieved autonomous driving and the computer vision objectives of lane, turtlebot, and stop sign detection.

References

[1] Chai, Z, Nie, T & Becker, J. 2021, Autonomous Driving Changes the Future 1st ed. 2021., Springer Singapore: Imprint: Springer, Singapore.

[2] Formoso, R. 2020. Available at: `https://github.com/COMP3431-Bob-Ros/ass2/blob/main/ass2_test/scripts/test_cam.py`

Appendix A Turtlebot Detection Pseudocode

The following pseudocode uses OpenCV2 to detect turtlebots by applying a binary mask over the black regions of the image and using simple blob detector to generate keypoints. Keypoints are parsed for a valid representation of a turtlebot and returns a boolean if a turtlebot is detected.

```
function detectTurtlebot(image):
    mask = apply_black_mask(image)
    mask = invert(mask)
    mask = flood_fill_holes(mask)
    keypoints = simple_blob_detector(mask, blob_params)
    detected = parseKeyPoints(keypoints, search_params)
    return detected
```

Appendix B Stop Sign Detection Pseudocode

The following pseudocode uses OpenCV2 to detect stop signs by applying a binary mask over the red regions of the image and using simple blob detector to generate keypoints. Keypoints are parsed for a valid representation of a stopsign and returns a boolean if a stopsign is detected.

```
function detectStopSign(image):
    image = convert_BGR_to_HSV(image)
    mask = apply_red_mask(image)
    mask = invert(mask)
    mask = morphological_open(mask)
    mask = morphological_close(mask)
    keypoints = simple_blob_detector(mask, blob_params)
    detected = parse_key_points(keypoints, search_params)
    return detected
```

Appendix C Turtlebot Detection Test Results

Test results show the original image on the left side and the applied mask with blob circling (in red) on the right.

Figure 6: Turtlebot Detection Test 1

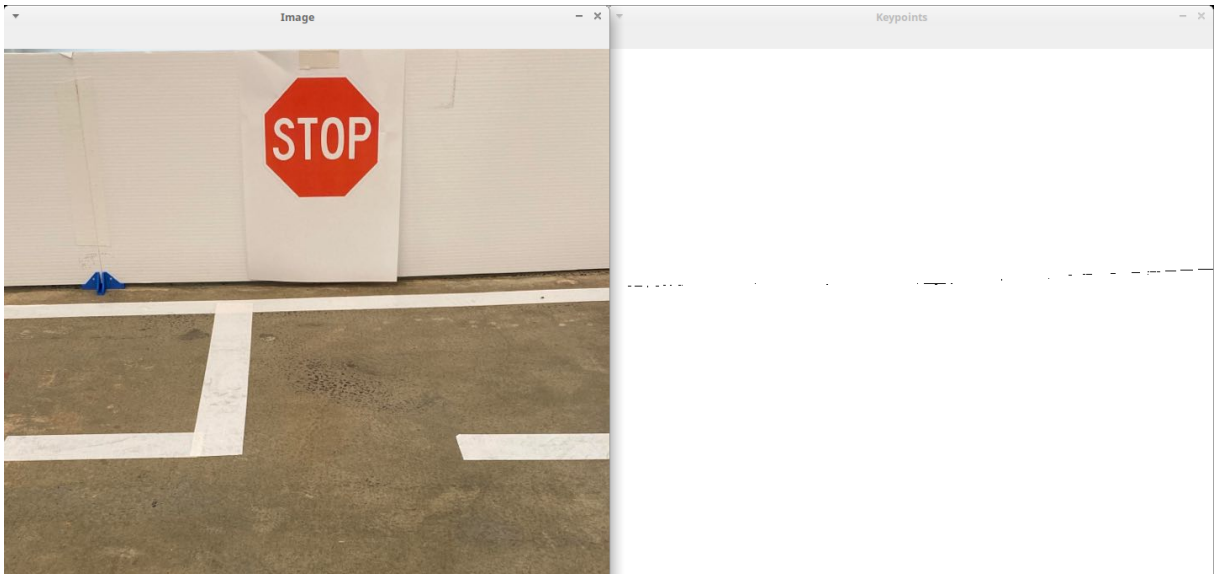

Figure 7: Turtlebot Detection Test 2

A more complex image was also tested where the mask was flood filled for stronger blob detecting.

Figure 8: Black Rose Detection Test

Appendix D Stop Sign Detection Test Results

Test results show the original image on the left side and the applied mask with blob circling (in red) on the right.

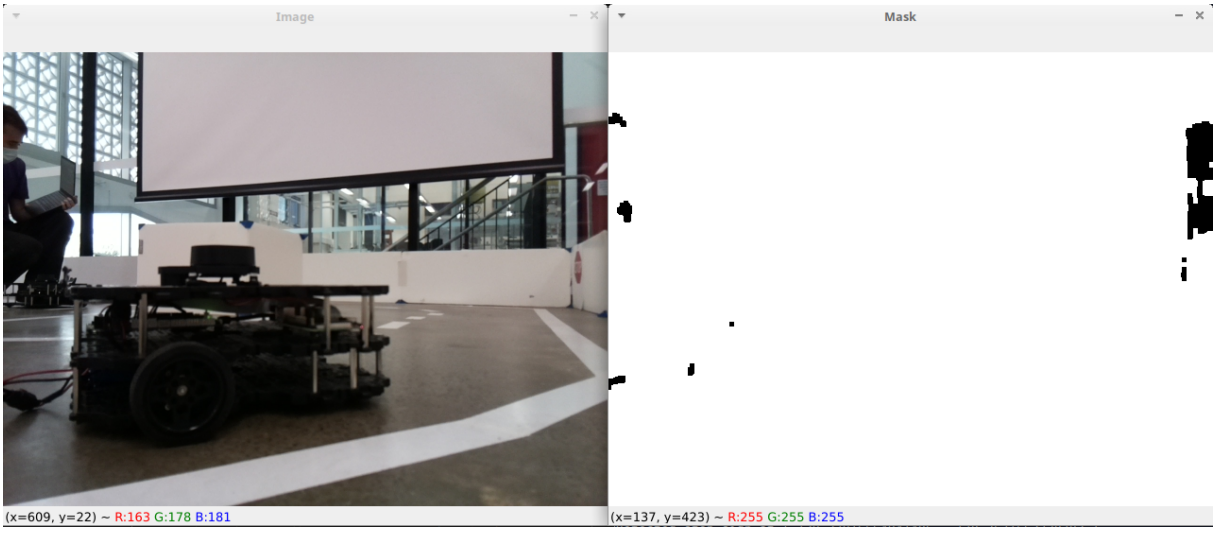

Figure 9: Stop Sign Detection Test 1

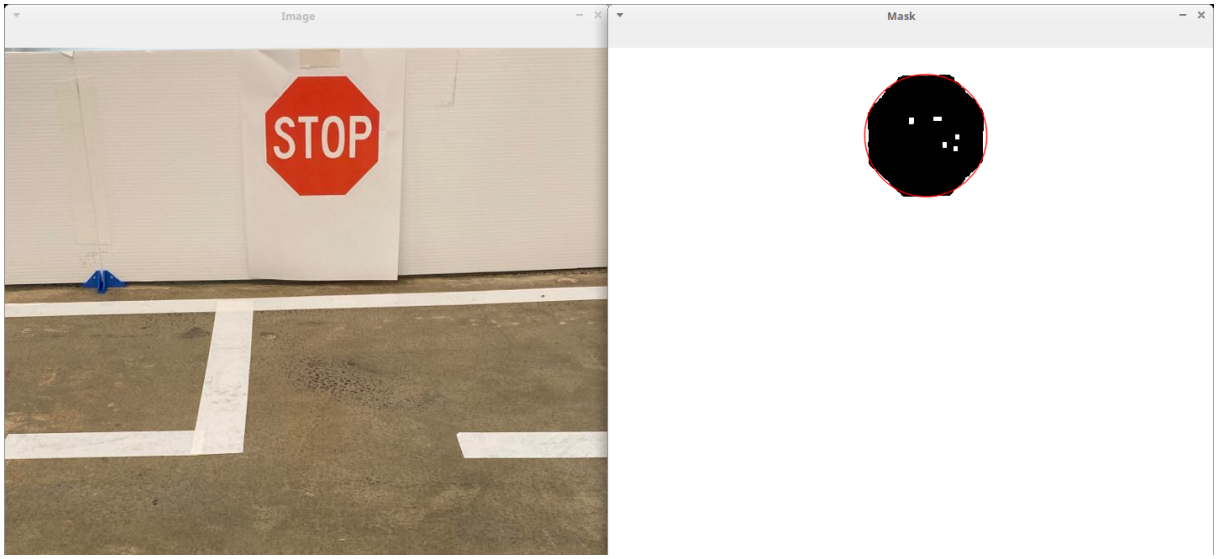

Figure 10: Stop Sign Detection Test 2

A more complex image was also tested where the mask was opened then closed to fill holes for stronger blob detecting.

Figure 11: Red Rose Detection Test