CONSTRUCTOR UNIVERSITY

Individual Report - Group Project

Author: **Anuraag Deshpande**

CO-548-A RIS Project

Instructor: Dr. Fangning Hu

Date: June 3, 2025

Group Members: Ziyu Gu (Anderson), Andres Moreno

GitHub code: https://github.com/ZiyuGu-Anderson/lili2.0

Abstract

The main objective of this lab was to implement a white line detection algorithm along with speed control, to make the duckiebot follow white lines throughout the duckietown map. For this purpose, white line detection was implemented using based on HSV Thresholding and Probablistic Hough Transform. Position and orientation errors were calculated from detected lines, and subsequently using the error as feedback for a proportional controller, we were able to achieve smooth line following. Collaborating with my teammate Ziyu Gu for line detection, I particularly contributed in the research and implementation of probabilistic hough transform, along with some fine-tunings for improved line detection. Key findings from this project include the importance and role of image processing and thresholding for line detection and a proportional controller for stability and easy small-scale modification.

1 Introduction

The main aim of our project was to investigate how line following can be implemented and potentially improved on a duckiebot. In the context of real-world applications, "Lane detection is a key component for self-driving cars to maintain the correct position on the road, especially in the absence of reliable GPS or when driving in complex urban environments" [1]. This emphasizes the importance of detecting and following lane markings, which serve as virtual guides for vehicle navigation. Since this is a crucial function of autonomous vehicles and uses a blend of different topics from computer vision and control systems, we were highly motivated to investigate this area. The final implementation of our project resulted in a line following algorithm that follows the white line in the duckietown map and uses a proportional controller to change speed accordingly. The main task was divided into two sections - line detection and speed control. Since the line detection part was more extensive, it was further divided into two parts, the first being HSV thresholding as a preprocessing step to find pixels that are likely part of white line. The second part, which I worked on, involved probabilistic hough transform to detect lines from those groups. Starting from relevant techniques that we used, our group took a fairly trial-and-error based approach for finetuning various parameters for both line detection and following, taking into consideration time constraints and other issues related to hardware. We used GitHub as our main platform for collaboration, along with other tools such as WhatsApp and Google Drive for effective communication and storing project-related files and sources.

2 My Contribution

As mentioned in the previous section, the main task was divided into line detection and speed control, and line detection was further divided between me and my teammate Ziyu Gu (Anderson) into two main parts - HSV thresholding and probabilistic hough transform. While we collaboratively worked and assisted each other, Ziyu Gu mainly worked on HSV thresholding, while I worked mainly on proabilistic hough transform. I implemented and tuned the Probabilistic Hough Transform to extract clean and consistent white line segments from the HSV-filtered binary image. This allowed us to compute the robot's heading angle relative to the white line, which was crucial for the orientation control. I also experimented with parameters such

as minimum line length and maximum gap to reduce false detections and improve robustness.

This project, including my work was done using ROS Noetic, by running it using Docker on Ubuntu 22.04 LTS. We considered Python to be the best choice for implementing these algorithms, given that it provides powerful libraries such as OpenCV for image processing, integration with ROS, and strong support for data processing and scientific computing made it ideal for analyzing image data, tuning control parameters, and debugging line detection performance efficiently.

3 Technical Explanation

3.1 Centroid-based vs. Contour based approach

As mentioned in the previous section, the main task was divided into line detection and speed control, and line detection was further divided between me and my teammate Ziyu Gu (Anderson) into two main parts - HSV thresholding and probabilistic hough transform. Although I worked mainly on the latter, I would like to also briefly touch upon the relevant part of HSV thresholding as a precursor to my individual work.

```
# 4. Centroid error
M = cv2.moments(mask)
err = 0.0
if M['m00'] > 0:
    cx = M['m10'] / M['m00']
    err = cx - (new_w / 2.0)
self.pub_error.publish(err)
```

Above is the specific part from our code for line detection that after the cropped camera image is converted into HSV color space, calculates the centroid using image moments for calculating the approximate distance of the line from the robot. As my teammate Ziyu Gu worked on this part, without going into too much detail, I would like to note that I had developed and tested an alternative approach to part 4 above, that uses a contour-based calculation to compute the centroid. The code snippet of that is shown below:

```
    # 4. Contour clustering to find largest white blob
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)

    err = 0.0
    if contours:
    # Sort by contour area (descending), pick the largest
    largest = max(contours, key=cv2.contourArea)
    M = cv2.moments(largest)
    if M['m00'] > 0:
        cx = M['m10'] / M['m00']
        err = cx - (new_w / 2.0)

    if self.debug:
        cv2.drawContours(roi, [largest], -1, (0, 0, 255), 2) # Red contour
```

Initially, after finding all continuous white regions or blobs from the binary mask, they are sorted by area and the largest one, which is likely the true line, is selected. The centroid is calculated using the same logic as before, and this information is further used for calculating position errors, etc. in later parts. The aim was to make our line detection code more robust to noise and distractions compared to using all white pixels. However when tested in the laboratory, results obtained were worse than the original centroid-based approach. After some research, we found out that there are some interesting issues that can arise with "visual tracking control of a wheeled mobile robot equipped with an on-board monocular camera", and especially external uncertainties during visual tracking process such as image noise, varying illumination and temporary occlusion are often overlooked [2]. And of course, as experimentally verified, the gray floor in the laboratory, uneven illumination, and other white objects and lines introduced uncertainties to our problem which could not be fully solved till the end. However, our centroid based approach offered more robustness against the noise and occlusion.

3.2 Probablistic Hough Transform - Background

The Hough transform is a technique for mapping points in digital pictures into a parameter space where patterns of points are identified as peaks [3]. As an example consider the problem of detecting ,lines. Points (x, y) on a line are constrained by the relation:

$$y = ax + b \tag{1}$$

In order to detect lines in a set of points, each point (x, y) "votes" for all pairs (a, b) that are related according to Eq. 1. Several techniques have been suggested for reducing the computational complexity of the standard Hough transform algorithm, the probabilistic approach being one of them. According to Kiryati *et al.* in the paper "A Probabilistic Hough Transform", "Its (referring to the Probabilistic Hough Transform) essence is the replacement of full scale voting in the incrementation stage

of the algorithm by a limited poll of a small number of edge points, selected at random." [4]. Intuitively, the subset of points that are voted reduces from M to m reducing complexity, and this method can still detect lines effectively. Hence, this method was chosen.

3.3 Probabilistic Hough Transform - Implementation

The code below is the actual implementation of the probabilistic Hough transform, which is the core of my work. Extra comments have been added to explain the code line by line.

```
    # 5. Hough line detection for orientation
    # Initialize orientation angle of the white line, to be updated based
        on detected lines
    angle = 0.0

    # Apply Probabilistic Hough Transform to find line segments in the
        white mask
    lines = cv2.HoughLinesP(mask,
                      self.hough_rho,
                      self.hough_theta,
                      self.hough_threshold,
                      minLineLength=self.hough_min_line,
                      maxLineGap=self.hough_max_gap)

    """
    cv2.HoughLinesP() is the OpenCV implementation of the Probabilistic
        Hough Transform. It takes the binary mask (where white pixels
        represent detected white line areas) and detects line segments in
        the image.
    """

    """
    Parameters:

    - mask: The edge-detected binary image (in your case, from HSV +
        morphology).
    - hough_rho: The resolution of the accumulator in pixels (e.g., 1 px).
    - hough_theta: Angular resolution (e.g., np.pi/360 means 0.5 per bin).
    - hough_threshold: Minimum number of "votes" (points on a line) to
        keep a line.
    - minLineLength: Minimum length of line to be accepted.
    - maxLineGap: Maximum allowed gap between segments to be merged into a
        single line.
    """

    if lines is not None:
        # Calculate orientation angle (in radians) of each detected
        # line segment with respect to the horizontal axis
        angles = [np.arctan2((y1 - y2), (x2 - x1)) for x1, y1, x2, y2 in
            lines[:, 0]]
        # Average the angles to get an overall heading direction
        angle = float(np.mean(angles))

    # Publish orientation angle for debugging or downstream use
    self.pub_angle.publish(angle)
```

The computed centroid and angular errors are used for a proportional controller seen in section 6, which was done by my teammate Andres, who was mainly involved with the proportional controller.

3.4 Hough Transform Parameters

There are many variables such as **minLineLength**, **hough_threshold**, etc. as seen in the above two sections. These variables are set before HSV thresholding or Probabilistic Hough Transform can be implemented, and are shown below:

```
# Hough parameters
self.hough_rho = rospy.get_param('~hough_rho', 1)  self.hough_theta =
    rospy.get_param('~hough_theta', np.pi/360)
self.hough_threshold = rospy.get_param('~hough_threshold', 80)
    self.hough_min_line = rospy.get_param('~hough_min_line', 60)
self.hough_max_gap = rospy.get_param('~hough_max_gap', 10)
```

These variables are set according to OpenCV's implementation of the Hough Transform [5]. Of course, some small changes have also been made using a limited amount of trial-and-error testing in the laboratory. After HSV thresholding parameters handle color-based detection for white lines in this case, these Hough Parameters are useful for detecting line structures the binary image generated.

3.5 Debugging

```
# 7. Debug visuals
if self.debug:
    # Convert the binary mask (white line segmentation result) to a ROS
        image message
    mask_img = self.bridge.cv2_to_imgmsg(mask, encoding='mono8')
    mask_img.header = msg.header # Preserve the original image header
        (timestamp, frame_id)
    self.pub_mask.publish(mask_img) # Publish the mask for
        visualization in RQT or RViz

    # Create a color version of the mask to draw colored lines on top
    vis = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)

    # If Hough lines were detected, draw them in green on the
        visualization image
    if lines is not None:
        for x1, y1, x2, y2 in lines[:, 0]:
            cv2.line(vis, (int(x1), int(y1)), (int(x2), int(y2)), (0,
                255, 0), 2)

    # Convert the Hough visualization to a ROS image message and
        publish it
    hough_msg = self.bridge.cv2_to_imgmsg(vis, encoding='bgr8')
    hough_msg.header = msg.header
    self.pub_hough.publish(hough_msg)
```

This section was collaboratively completed by Ziyu Gu and I, since it involved elements from both of our individual works. It was useful as a visual and in case of any required debugging.

Looking at the above code firstly, it converts the binary mask obtained from HSV thresholding and converts it into a BGR image for drawing colored lines and draws all the lines detected using the probabilistic hough transform using green lines of thickness 2 pixels. Then, it converts the annotated image (vis) into a ROS message (bgr8 encoding = 8-bit color image), adds the original header for time/frame alignment and Publishes the result so the detected lines can be easily seen overlaid on the mask. This code only runs if the debug mode is enabled on ROS, which is checked in the first line. In short, it enables visualization of detected lines and thus helps to debug the code in case of unexpected problems/outcomes.

3.6 ROS Nodes

HSV thresholding combined with my work in probabilistic hough transform runs on the node **WhiteLineDetectorFollowerNode**, as shown below:

```
if __name__ == '__main__':
    node =
        WhiteLineDetectorFollowerNode(node_name='white_line_detector_follower')
    node.run()
```

As seen above, if run in the main program, our code creates an instance of our custom node class, registering the ROS master under the name **white_line_detector_follower**. This, combined with our ROS node for speed control called **TwistToWheelsNode** are the two main nodes that define our project.

However, there is a third important component. The prefix **car_cmd_switch_node** is used as a namespace to organize related topics, and **cmd** is the specific topic used for sending motion commands (**Twist2DStamped** messages). This is shown below, and defined towards the beginning of our file for line detection.

```
# Publishers
self.pub_error = rospy.Publisher('~line_error', Float32, queue_size=1,
    dt_topic_type=TopicType.DEBUG)
self.pub_angle = rospy.Publisher('~line_angle', Float32, queue_size=1,
    dt_topic_type=TopicType.DEBUG)
self.pub_cmd = rospy.Publisher('car_cmd_switch_node/cmd',
    Twist2DStamped, queue_size=1, dt_topic_type=TopicType.CONTROL)
```

This structure helps keep the system modular and scalable, especially when multiple components or robots are involved. The line-following node publishes to this topic so that **TwistToWheelsNode** can subscribe and act on the command.

4 Conclusion and Future work

My work with the probabilistic hough transform worked sufficiently well, although it faced some challenges with precise line detection due to numerous white lines on the duckietown map. As mentioned earlier, it was a crucial aspect of our node dealing with line detection. Firstly, the HSV thresholding method creates a binary mask by separating the white parts which leads to my work with proabilistic hough transform to detect lines. These detected lines are further used to adjust position and orientation for speed control using a proportional controller.

I learned a lot technically, including the use of OpenCV, the implementation of the probabilistic Hough transform in Python, and how to fine-tune parameters and algorithms based on project context, while also understanding how different ROS nodes and topics integrate into a larger system. This project required me to use a lot of soft-skills as well including critical thinking, collaboration and time management, to maximize our results even with minimal time and resources. As a group, we managed to create a white line following program that worked sufficiently well as demonstrated in the laboratory, despite not having sufficient time for testing because of repeated hardware issues throughout the semester.

Implementation-wise, issues such as uneven illumination, a gray floor, white objects, and white lines near the map's edge made perfect line-following difficult. Therefore future improvements include improving the HSV and Hough parameters to make them stricter, by running repeated tests a.k.a. through trial-and-error, and expanding functionality by using the white lines on either side to drive on the road, as well as detecting e.g. red lines to stop at a crossroad. This would be more practical by mimicking real-world scenarios [1].

References

J. Janai, F. Güney, A. Behl, A. Geiger *et al.*, "Computer vision for autonomous vehicles: Problems, datasets and state of the art," *Foundations and trends® in computer graphics and vision*, vol. 12, no. 1–3, pp. 1–308, 2020.

C.-Y. Tsai and K.-T. Song, "Dynamic visual tracking control of a mobile robot with image noise and occlusion robustness," *Image and Vision Computing*, vol. 27, no. 8, pp. 1007–1022, 2009.

J. R. Bergen and H. Shvaytser, "A probabilistic algorithm for computing hough transforms," *Journal of algorithms*, vol. 12, no. 4, pp. 639–656, 1991.

N. Kiryati, Y. Eldar, and A. M. Bruckstein, "A probabilistic hough transform," *Pattern recognition*, vol. 24, no. 4, pp. 303–316, 1991.

OpenCV, "Opencv," https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html, 2025, accessed: 2025-06-03.