DUCKIETOWN

Autonomous navigation and real-time mapping on iOS

MARCH 14, 2024

TUM
Tobias Coptil, Manjunath Naidugari , Yifan Zhong

1. Contents

2. Introduction

Link to our Github repository: https://github.com/CoTobias/Duckietown-Coolduck

In the field of robotics and mobile applications, the combination of autonomous navigation and real-time mapping has become an exciting frontier. Our project aims to develop a sophisticated solution that seamlessly integrates these capabilities. This will allow users to visualize the path of a robot navigating through unknown environments directly on their iOS devices.

The project aims to empower users with a comprehensive understanding of a robot's movements in real-time, even in uncharted territories. This will be achieved by harnessing lane following and various algorithms for data analysis to deliver an intuitive iOS application that offers both utility and insight into autonomous robotic navigation, as well as key information about the robot itself.

The project can be divided into three main parts, which we will go into detail separately:

1. Lane following
2. Calculation and Analysis of gathered data
3. Visualization in iOS

3. Self-Driving Duckiebot with Lane Following and AprilTag Detection

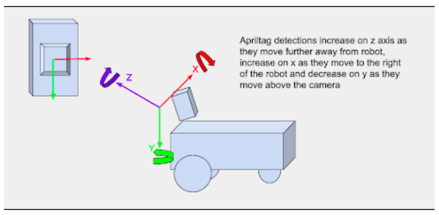
3.1 Introduction

This project focuses on implementing a self-driving system for the Duckiebot. As an autonomous vehicle navigating through lane following the bot tries to run on the lane by following the yellow dotted line while maintaining a calculated distance from it[1]. During the intersections, we use Apriltags, a type of fiducial marker consisting of a black square with a white foreground generated in a particular pattern.[1] Their unique IDs and reliable detection makes them ideal markers for precise localization, enabling the robot to navigate intersections accurately and efficiently.

3.2 April Tag Detection

The motivation behind utilizing AprilTags for navigation within Duckietown's intersections is that AprilTag payloads are so small that they only hold 4-12 bits of data, and they can be more easily detected, more robustly identified, and less difficult to detect at longer ranges.[2] April Tags offers a reliable and precise means of localization, particularly beneficial for navigating intersection. The AprilTag detection module utilizes computer vision libraries, notably OpenCV, to identify AprilTags within the camera's field of view. Upon detection, the AprilTag library returns an AprilTag object, providing valuable data such as tag ID and corner coordinates. This information is instrumental in the navigation process. By utilizing commands like **apriltag.corners**, we draw bounding boxes around the detected tags, aiding in visualization and further processing.[3] Here is an example of a tag and its ID.

STOP
apriltag.Tag36h11, id = 26


Z
X
Y
Apriltag detections increase on z axis as they move further away from robot, increase on x as they move to the right of the robot and decrease on y as they move above the camera

[1] Samiei, Mandana. "Mandanasmi/Lane-Slam." GitHub, 3 Feb. 2024, github.com/mandanasmi/lane-slam. Accessed 14 Mar. 2024.
[2] Rosebrock, Adrian. "AprilTag with Python." PyImageSearch, 2 Nov. 2020, pyimagesearch.com/2020/11/02/apriltag-with-python/.
[3] "Exercise 3." *Ekhumbata.github.io*, ekhumbata.github.io/Adventures-in-Duckietown/exercise3.html. Accessed 11 Mar. 2024.

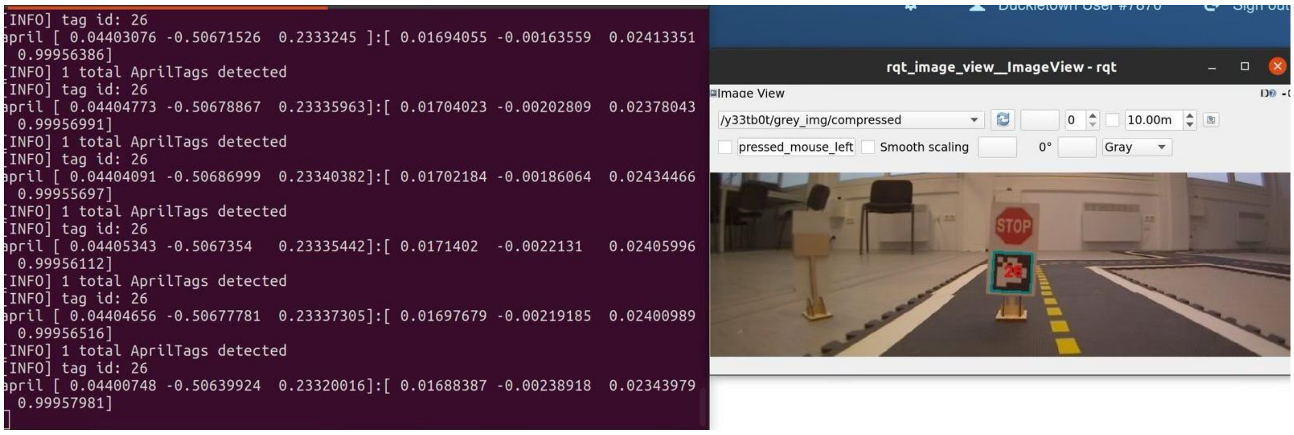
```
# subscribers
img_topic = f"""/{os.environ['VEHICLE_NAME']}/camera_node/image/compressed"""
info_topic = f"""/{os.environ['VEHICLE_NAME']}/camera_node/camera_info"""
self.img_sub = rospy.Subscriber(img_topic, CompressedImage, self.cb_img, queue_size = 1)
self.subscriberCameraInfo = rospy.Subscriber(info_topic, CameraInfo, self.camera_info_callback,  queue_size=1)

# publishers
# self.pub = rospy.Publisher('/grey_img/compressed', CompressedImage, queue_size=10)
self.pub = rospy.Publisher("/" + os.environ['VEHICLE_NAME'] + '/grey_img/compressed', CompressedImage, queue_size=1)
self.pub_led = rospy.Publisher("/" + os.environ['VEHICLE_NAME'] + "/led_emitter_node/led_pattern", LEDPattern, queue_size=1)
self.pub_april = rospy.Publisher("/april_topic", String, queue_size=1)
```

We heavily use OpenCV[4] and the AprilTag library since the task is to process the image and detect the information. Here is a breakdown of the code, which explains what we did in order to build the detector: Generally the code subscribes to the camera image topic, publishes the modified image with detected April tags to a new topic and then changes the LED color based on the detection[5].
Afterwards the image using OpenCV gets preprocessed to prepare it for as input to the AprilTag detector:[4]

```
    data_arr = np.frombuffer(msg.data, np.uint8)
    col_img = cv2.imdecode(data_arr, cv2.IMREAD_COLOR)
    grey_img = cv2.cvtColor(col_img, cv2.COLOR_BGR2GRAY)
```

The output of this method is a list of detections, each of which indicates a detected AprilTag. The position of each can be found by drawing a rectangle around it and thus we can indicate its label by iterating through these detections and using detection attributes(corner and center):



```
[INFO] tag id: 26
april [ 0.04403076 -0.50671526  0.2333245 ]:[ 0.01694055 -0.00163559  0.02413351
  0.99956386]
[INFO] 1 total AprilTags detected
[INFO] tag id: 26
april [ 0.04404773 -0.50678867  0.23335963]:[ 0.01704023 -0.00202809  0.02378043
  0.99956991]
[INFO] 1 total AprilTags detected
[INFO] tag id: 26
april [ 0.04404091 -0.50686999  0.23340382]:[ 0.01702184 -0.00186064  0.02434466
  0.99955697]
[INFO] 1 total AprilTags detected
[INFO] tag id: 26
april [ 0.04405343 -0.5067354   0.23335442]:[ 0.0171402  -0.0022131   0.02405996
  0.99956112]
[INFO] 1 total AprilTags detected
[INFO] tag id: 26
april [ 0.04404656 -0.50677781  0.23337305]:[ 0.01697679 -0.00219185  0.02400989
  0.99956516]
[INFO] 1 total AprilTags detected
[INFO] tag id: 26
april [ 0.04400748 -0.50639924  0.23320016]:[ 0.01688387 -0.00238918  0.02343979
  0.99957981]
```

3.3 Lane Following

The fundamental aspect of self-driving cars is lane following, which uses computer vision approaches to detect lane markings and modify the bot's trajectory to remain within the lane. The lane following

```
# get the image from camera and mask over the hsv range set in init
data_arr = np.fromstring(msg.data, np.uint8)
col_img = cv2.imdecode(data_arr, cv2.IMREAD_COLOR)
crop = [len(col_img) // 3, -1]
hsv = cv2.cvtColor(col_img, cv2.COLOR_BGR2HSV)
imagemask = np.asarray(cv2.inRange(hsv[crop[0] : crop[1]], self.lower_bound, self.upper_bound))
```

which I used will process images captured by Duckiebot's camera to pinpoint the yellow dotted line.

[4] "Lane Following Robot Using OpenCV." Hackster.io, www.hackster.io/Aasai/lane-following-robot-using-opencv-da3d45

[5] "Golnaz Mesbahi - Lab 3." *Github.io*, 2023, golnazmes.github.io/Lab%203.html. Accessed 11 Mar. 2024.

The PID controller maintains a consistent distance (e.g., 0.8 cm) from the dotted line to the center of the Duckiebot. This can be visualized for example in rqt_image_view.
After this, we find all the yellow dotted lines used to follow the lane.

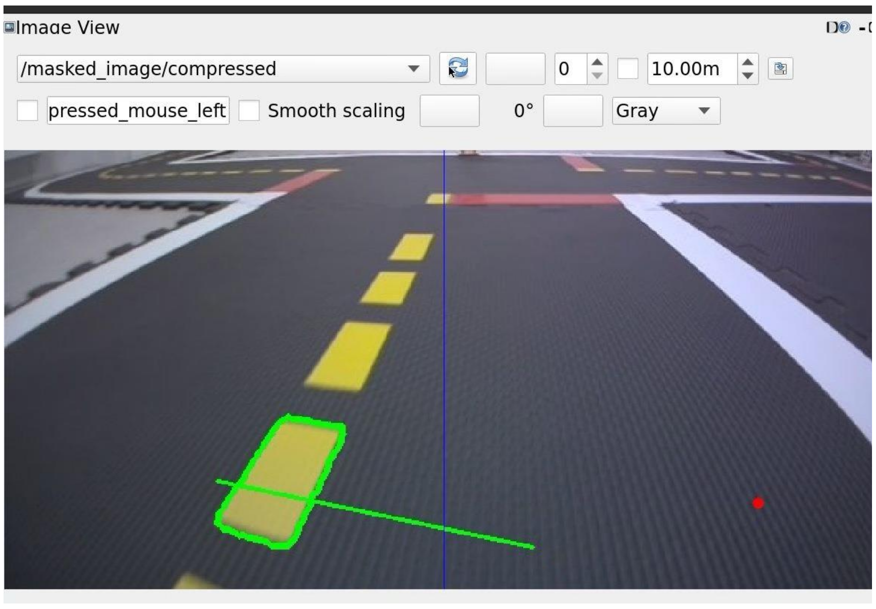
```
# find all the yellow dotted lines
contours, hierarchy = cv2.findContours(imagemask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

By changing the offset of the PID controller, we can change English driving mode to American driving mode[6]. The error in the controller is found by the following general equation:[7]

$$error = (x + (sizeratio * y)) - goal) * pixelscale$$

Where:

- x = The left most side of the closest dotted line
- y = The top of the closest dotted line divided by two
- size-ratio = Handles distance distortion. It creates a linear function describing how close the dotted line is vs how close the Duckiebot should be to it. This addresses the vanishing point problem.
- goal = Desired location of the bot. Represented by a blue vertical line in the image
- pixel-scale = Scales the error down to account for a large error in pixels relative to a small error in angle



3.4 Challenges

PID parameters: To run the Duckiebot on the lane, we need to find the right values for the parameter in the PID controller, otherwise the bot won't run the desired way.
Integration issues: The bot is unable to move in the desired way when integrating lane following and AprilTags at the intersections.

[6] "Exercise 3." *Ekhumbata.github.io*, ekhumbata.github.io/Adventures-in-Duckietown/exercise3.html. Accessed 11 Mar. 2024.

3.5 Future Improvements

- Enhance the robustness of lane-following algorithms to handle complex road scenarios and varying lighting conditions.
- Implement advanced navigation techniques like SLAM to improve localization and mapping capabilities.
- Implementing static and dynamic vehicle avoidance. Integration of additional sensors, such as LiDAR, for enhanced perception, obstacle avoidance and path planning.
- Implementing a parking feature where Duckiebot will park on its own once it’s reached the destination.

4. Trajectory Generation and Analysis

The main part of this part of the project involves the development of a trajectory generation and analysis system for the Duckiebot. The programmed system uses wheel encoder data to estimate the Duckiebot's trajectory and analyses the movement patterns on a track. The primary aim of this task is to obtain the coordinates of the robot's position, thereby mapping its location in the past and present on a coordinate system using an “Elementary Trajectory Model for the Differential Steering System[8]”.

Furthermore using the calculated coordinates, the project identifies straight and left curve segments in the path of the Duckiebot’s when controlled manually, which allows for a simulation of the track map the Duckiebot traverses.

4.1 Introduction

The system for generating and analyzing trajectories is implemented in Python using the Robot Operating System (ROS) framework. The core class “Trajectory” is a subclass of DTROS[9] (Ducktown Robot Operating System), providing the necessary functionalities and communication interfaces.

The Duckiebot system publishes information on the rotation of its left and right wheels over ROS topics. The message type of these topics contains various fields, but the code only uses “*ENCODER_TYPE_INCREMENTAL*” to count the number of ticks the robot has at any given time and thus also their change. This is possible because each motor records 135 ticks per full revolution, and that data starts at 0 from the initial position of the wheel when the robot is turned on[10]. Additionally, parameters are assigned to ensure the highest possible accuracy of the coordinates. The parameters required for this task are the wheel radius (cm), wheel distance(cm) and speed (cm/s), initialized in the main method of the code using the constructor.

4.2 Trajectory Class

The main method instantiates the 'Trajectory' class with the aforementioned parameters and invokes its 'run' method. To keep the Python script responsive to ROS events, the 'rospy.spin' [11] function is called. The 'run' method updates coordinates, analyzes tracks, publishes coordinates and executes commands at the end of each track. To provide a more detailed explanation of how this code functions, it will be divided into three sections.

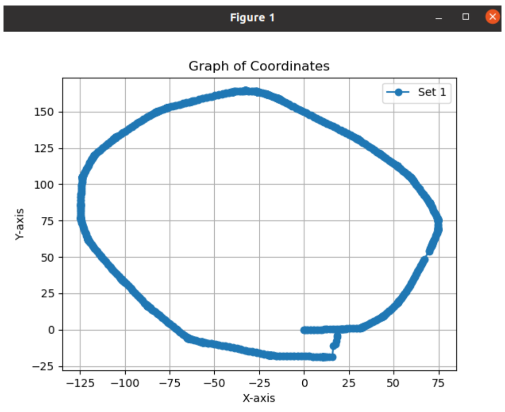
[8] “A Tutorial and Elementary Trajectory Model for the Differential Steering System.” Rossum.sourceforge.net, rossum.sourceforge.net/papers/DiffSteer/
[9] “New ROS DTProject — Book-Devmanual-Software - Daffy.” *Docs.duckietown.com*, docs.duckietown.com/daffy/devmanual-software/beginner/ros/create-project.html. Accessed 21 Feb. 2024.
[10] “Subscribe to Wheel Encoders — Book-Devmanual-Software - Daffy.” Docs.duckietown.com, docs.duckietown.com/daffy/devmanual-software/beginner/ros/wheel-encoder-reader.html. Accessed 20 Feb. 2024.
[11] “What Is Difference between Rospy.spin() and While Not Rospy.is_shutdown()?” *The Construct ROS Community*, 2 Jan. 2020, get-help.theconstruct.ai/t/what-is-difference-between-rospy-spin-and-while-not-rospy-is-shutdown/1849. Accessed 12 Mar. 2024.

4.3 Calculation of Coordinates

This section updates the robot's position with a frequency of 20Hz using wheel encoder data, the orientation theta value, and previous values. The method then adds each coordinate to the list of coordinates visited by the Duckiebot. Furthermore, functionality has been added to detect when the bot reaches the end of the track after no movement is recorded for more than 2 seconds by either wheel encoder. This ensures that the calculation of track types is possible at a later point in time. The crucial aspect of this code is the calculation of the coordinates.

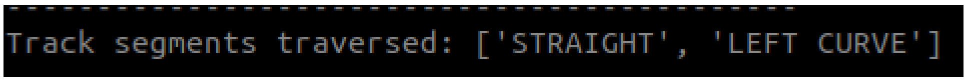
This calculation is used to calculate the “robot's position in dead-reckoning or *localization by odometry* applications”[12] as used by this code to create coordinates.

The x and y coordinates are calculated using the distances travelled by each wheel and the last known value of orientation 'theta' in the Differential Drive model. 'Theta' represents the direction of movement of the Duckiebot and updates itself with each calculation. The final result of the calculation are the values of ‘x’ and ‘y’. The initial value of 'theta' is '0', indicating that the Duckiebot is facing east.



4.4 Analyzation of Track Types

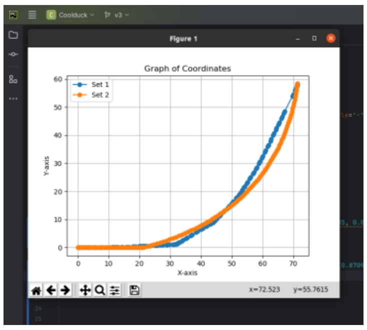
Only when reaching the end of the track the main method of this section starts processing information. Similar to the previous section it chains different methods to create the wanted output. It determines the movement pattern of a track using the absolute degree of the slope between the start and end coordinates of the track the bot has just crossed. Moreover it simulates the direction of movement using the orientation value at the end and start point. Special attention has to be paid here to the calculation of the slope as it depends on the orientation value ‘theta’, defining the direction of orientation and thus also if it a east-west movement or north-south movement[13]. The method thus returns one of the three track types considered in this code: Straights, Left Curves and None.
This works only when controlling the bot manually and waiting at the end of the track for 2 second.



```
Track segments traversed: ['STRAIGHT', 'LEFT CURVE']
```

[12] “A Tutorial and Elementary Trajectory Model for the Differential Steering System.” Rossum.sourceforge.net, rossum.sourceforge.net/papers/DiffSteer/
[13] Mazur, Izabela, and Kim Moshenko. “3.4 Understand Slope of a Line – Optional.” *Opentextbc.ca*, 8 Sept. 2021, opentextbc.ca/businesstechnicalmath/chapter/understand-slope-of-a-line-2/. Accessed 23 Fev. 2024.

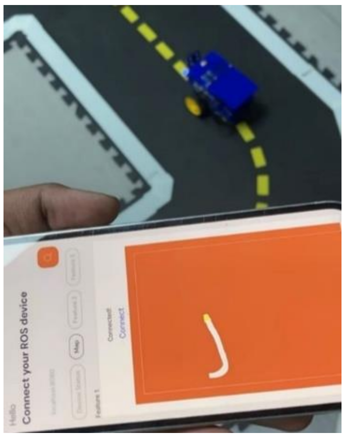
4.5 Creating the simulation of a track

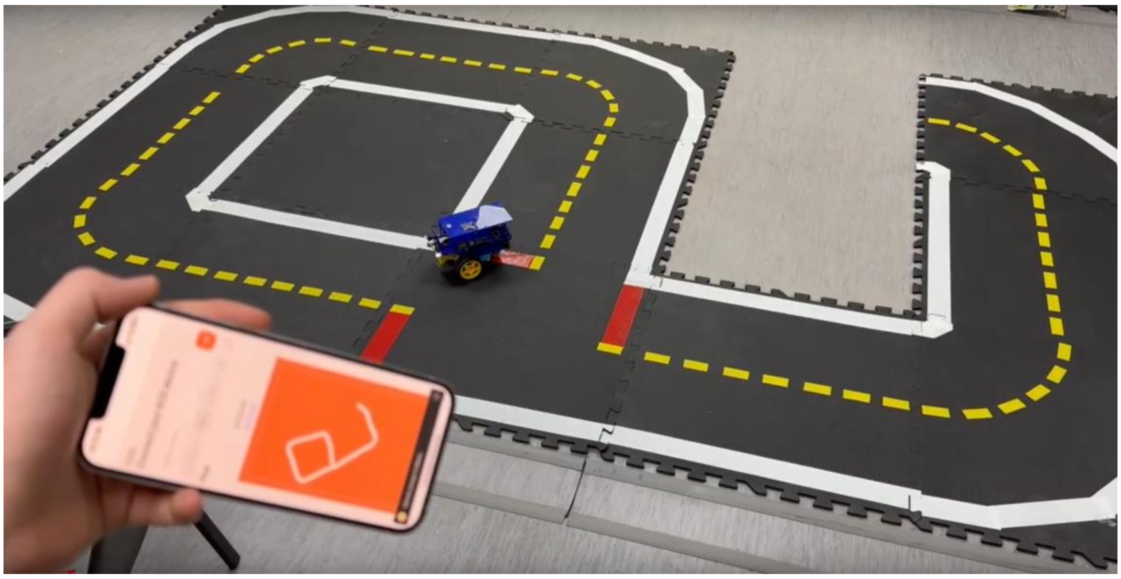


This section uses the pattern found out in the previous step and simulates the outline of the map of tracks the Duckiebot traverses.

4.6 Visualization

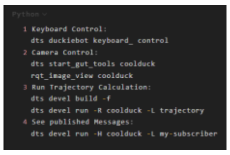
The mobile application designed for this purpose is used to visualize the bot's path and simulated track on the iOS device. The before mentioned section provides more detail on this, but the transmission of data from the bot to the mobile application works as follows:
The publisher[14] in the Trajectory class publishes[15] a message as a string containing the trajectory coordinates and the simulated trajectory coordinates every time the run method runs (at a frequency of 20Hz). The mobile application's only identification measure is the '@' symbol between the concatenated strings.





4.7 Usage

To use this model the user has to be connected in the same network as the Duckiebot. Once confirmed, execute each command shown in the provided picture in separate terminal windows.



```
Python
Keyboard Control:
  dts duckiebot keyboard_ control
Camera Control:
  dts start_gut_tools coolduck
  rqt_image_view coolduck
Run Trajectory Calculation:
  dts devel build -f
  dts devel run -R coolduck -L trajectory
See published Messages:
  dts devel run -H coolduck -L my-subscriber
```

[14] "Publish Chassis-Level Commands — Book-Devmanual-Software - Daffy." *Docs.duckietown.com*, docs.duckietown.com/daffy/devmanual-software/beginner/ros/twist-control.html. Accessed 23 Feb. 2024.
[15] "Publish to Wheels — Book-Devmanual-Software - Daffy." *Docs.duckietown.com*, docs.duckietown.com/daffy/devmanual-software/beginner/ros/wheel-control.html. Accessed 23 Feb. 2024.

4.8 Challenges

In the beginning the analysis of the duckiebot movement was a tough challenge to master. From the start the use of wheel encoders seemed to be a good but unreliable idea, which is why we first started with the use of slam[16]. Quickly we realized that without a surrounding ecosystem this method will prove even more difficult to implement. After advancing in this matter we began to make progress and still could only detect a minority of the track reliable. Because of this reason we decided on testing wheel encoder analysis, which proved in the end successful and in most cases reliable. Through trial and error, adjusting of parameters and mathematical perspective on the matter of Trajectory Analysis we then got a solution that has just minor issues:

- Network Connectivity: Transmitting at a rate of 20Hz per second is optimal for proper calculation of the track, but the network is not fast enough to transmit all data to the mobile application in this period of time which leads to a delay on the mobile phone when operating the Duckiebot for more than 4 minutes.
- Error Handling: Most cases of error handling have been nullified, we did not find a solution for the issue of extremely quick turns of the robot. This case happens very rarely in our lane following, if this issue arises though miscalculations happen and a right trajectory of the Duckiebot cannot be guaranteed anymore.

4.9 Conclusion

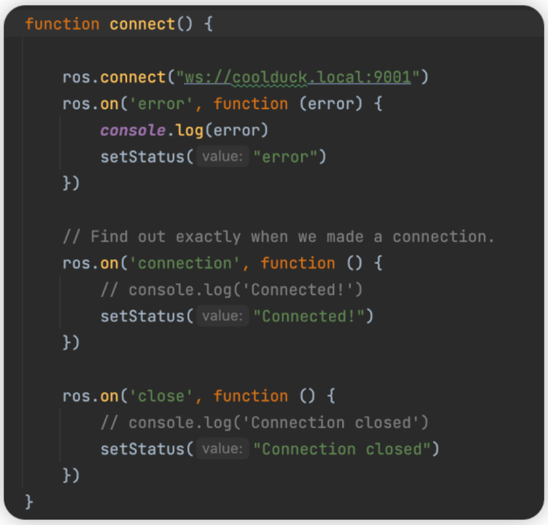
Overall, this trajectory generation and analysis system provides a comprehensive solution for mapping the Duckiebot's movement, identifying track types, and visualizing the trajectory on an iOS device. The system demonstrates the integration of ROS, wheel encoder data, and mathematical models for accurate trajectory analysis. Future improvements would contain the implementation of further Track Types such as right curves or intersections and Path planning (Integrate algorithms that allow the Duckiebot to autonomously navigate through the before analyzed track based on the data it gathered).

5. iOS application for ROS (Mobile ROS)

5.1 Introduction

To get started, the fundamental problem of building an application to communicate with robots is the lack of Linux and ROS environment, especially for iOS application. As our goal is to build mainly a iOS mobile application, we choose React Native as our development framework[17]. It allows us to build cross-platform application over iOS, Android as well as web using Javascript via same codebase. Thus upon the selection of React Native, we also used expo for the software. Expo[18] is an open source platform for building and deploying native iOS and Android apps that is based on the React Native. We also adopted the router part from expo.
The application development can be divided in these parts: communication, UI and functionality.

5.2 Communication



```
function connect() {

    ros.connect("ws://coolduck.local:9001")
    ros.on('error', function (error) {
        console.log(error)
        setStatus( value: "error")
    })

    // Find out exactly when we made a connection.
    ros.on('connection', function () {
        // console.log('Connected!')
        setStatus( value: "Connected!")
    })

    ros.on('close', function () {
        // console.log('Connection closed')
        setStatus( value: "Connection closed")
    })
}
```

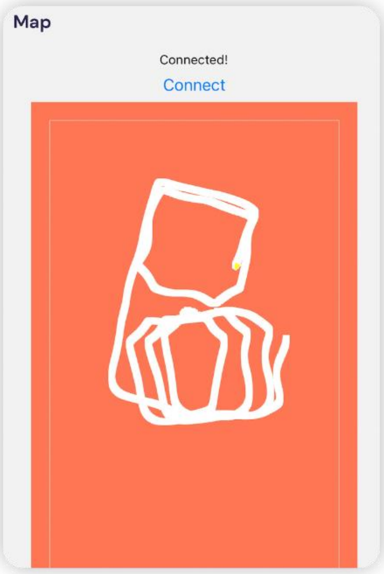
ROS bridge[19] is providing JSON API to non_ROS programmes. In this sense smartphone could treat ROS communication as common communication to backend regardless of the Linux and ROS environment and dependencies, which exactly suits our need. ROS bridge consists of two parts, server side and client side[20]. Server side, as stated in its name, is installed on server ROS robot. Client side could be on different platform and using different programming languages.The main obstacle here is integration of the client side for ROS bridge over React Native framework. As the one of the ROS bridge client is designed for Javascript, in theory it should be straightforward to call the functions accordingly. Nevertheless, as the library is not actively maintained and lack of documentation. We still encountered several issues while connecting. For example, the compatibility from server side to client side. So we have to switch in between different version for expo, react native, and roslib (the client side for ROS bridge).

5.3 UI:

Different from old fashioned UI design for php applications, react native offers a modern UI possibility for us. Along with standard stylesheet, it provides a consistent experience and smooth dynamic animations such as button animation, sliding tab for mobile, stacking design to suit mobile

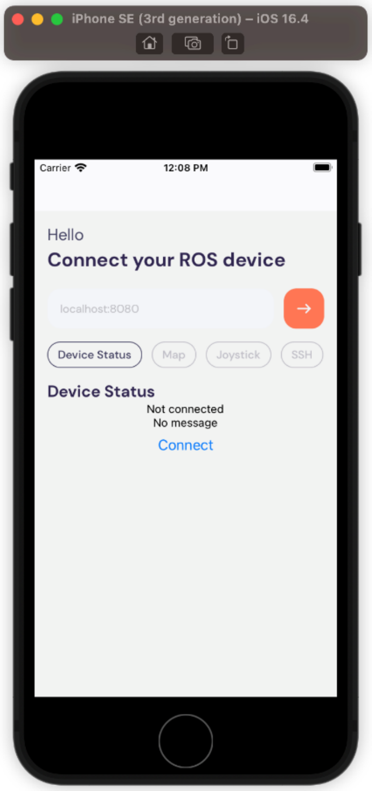
[17] React native · learn once, write anywhere React Native. Available at: https://reactnative.dev/ (Accessed: 10 March 2024).
[18] *Github Expo: An open-source platform for making universal native apps with react. expo runs on Android, IOS, and the web.*, GitHub. Available at: https://github.com/expo/expo (Accessed: 10 March 2024).
[19] *Rosbridge Protocol - JSON API to ROS functionality for non-ROS programs. ros.org*. Available at: http://wiki.ros.org/rosbridge_suite (Accessed: 10 March 2024).
[20] Rosbridge v2.0 protocol specification GitHub. Available at: https://github.com/RobotWebTools/rosbridge_suite/blob/ros1/ROSBRIDGE_PROTOCOL.md (Accessed: 10 March 2024).

application interface. Apart from learning from scratch to use both framework and styling, we spend most of our time in developing the map component.



To have a live map system my first thought is using library similar to google map. But none of these libraries suit the indoor mapping and have to work with real GPS coordinates. So we heading to 3D plotting engine such as Threejs, it worked perfectly fine on web version, but still it does not support React Native. To be specific, for the purpose of rendering the graphics, react-three and expo-three need to be used as bridge, but again the repositories are not actively maintained and this time without any luck I failed to find suitable compatible version, as our other dependencies are fixed due to roslib (ROS bridge client side).

Thus, I made a function to draw the map dynamically myself using react-native-svg. React-native-svg[21] is a library enable user to draw svg graphics in react-native. The basic function contains shapes such as circle, ellipse, path, polygon, line etc. All of applicable shapes are quite basic, on the other hand, the performance is relatively good.



5.4 Functionality:

As our goal is to immigrate current functionality of common application from web application, functionality is consistent. Nevertheless for mobile application it provides much smooth and convenient user experience. Joystick function for example, it could control the robots with force sense mechanism as well as precise control due to natural subtle input advantage compared with keyboard. Also we fetch the common device status including battery, temperature, disk utilities etc. Trajectory mapping is one of our key function, we could generate two type of trajectories. One is called wheel encoder trajectory and another one is simulation track trajectory.

5.5 Conclusion:

In this section, we success to immigrate the common features needed for ROS robots on iOS (not only for iOS it supports android and web as well). We utilized the result from our work on trajectory generation and self-driving. As the main obstacle of cross-platform issue and compatibility is solved, it also leaves spaces for further potential application development.

6. Final System

The Final System of our project is thus an end-to-end system designed for autonomous navigation and realtime transmittion of Trajectory onto iOS. It can autonomously navigate unknown maps using Lane Following and transmit it’s path taken to the users mobile application. Additionally, the mobile

[21] Software-Mansion (no date) Software-mansion/react-native-SVG: SVG Library for react native, react native web, and plain react web projects., GitHub. Available at: https://github.com/software-mansion/react-native-svg (Accessed: 10 March 2024).

application delivers important information to the user such as Battery, Temperature, disk utility and other important Information. With our system the user can thus optimally use our resources to achieve efficient and safe navigation in various environments while staying informed about the status of the system components and conditions. By seamlessly integrating autonomous navigation capabilities with real-time trajectory transmission onto iOS devices, our solution empowers users to make informed decisions, maximize resource utilization, and enhance overall operational effectiveness.

7. Bibliography

Software-Mansion (no date) Software-mansion/react-native-SVG: SVG Library for react native, react native web, and plain react web projects., GitHub. Available at: https://github.com/software-mansion/react-native-svg (Accessed: 10 March 2024).

React native · learn once, write anywhere React Native. Available at: https://reactnative.dev/ (Accessed: 10 March 2024).

Github Expo: An open-source platform for making universal native apps with react. expo runs on Android, IOS, and the web., GitHub. Available at: https://github.com/expo/expo (Accessed: 10 March 2024).

Rosbridge Protocol - JSON API to ROS functionality for non-ROS programs. ros.org. Available at: http://wiki.ros.org/rosbridge_suite (Accessed: 10 March 2024).

Rosbridge v2.0 protocol specification GitHub. Available at: https://github.com/RobotWebTools/rosbridge_suite/blob/ros1/ROSBRIDGE_PROTOCOL.md (Accessed: 10 March 2024).

"Publish Chassis-Level Commands — Book-Devmanual-Software - Daffy." *Docs.duckietown.com*, docs.duckietown.com/daffy/devmanual-software/beginner/ros/twist-control.html. Accessed 23 Feb. 2024.

"Publish to Wheels — Book-Devmanual-Software - Daffy." *Docs.duckietown.com*, docs.duckietown.com/daffy/devmanual-software/beginner/ros/wheel-control.html. Accessed 23 Feb. 2024.

"A Tutorial and Elementary Trajectory Model for the Differential Steering System." Rossum.sourceforge.net, rossum.sourceforge.net/papers/DiffSteer/
Mazur, Izabela, and Kim Moshenko. "3.4 Understand Slope of a Line – Optional."*Opentextbc.ca*,8 Sept. 2021, opentextbc.ca/businesstechnicalmath/chapter/understand-slope-of-a-line-2/. Accessed 23 Fev. 2024.
"A Tutorial and Elementary Trajectory Model for the Differential Steering System." Rossum.sourceforge.net, rossum.sourceforge.net/papers/DiffSteer/
"New ROS DTProject — Book-Devmanual-Software - Daffy." *Docs.duckietown.com*, docs.duckietown.com/daffy/devmanual-software/beginner/ros/create-project.html. Accessed 21 Feb. 2024.
[1] "Subscribe to Wheel Encoders — Book-Devmanual-Software - Daffy." Docs.duckietown.com, docs.duckietown.com/daffy/devmanual-software/beginner/ros/wheel-encoder-reader.html. Accessed 20 Feb. 2024.

"What Is Difference between Rospy.spin() and While Not Rospy.is_shutdown()?"*The Construct ROS Community*,2 Jan. 2020, get-help.theconstruct.ai/t/what-is-difference-between-rospy-spin-and-while-not-rospy-is-shutdown/1849. Accessed 12 Mar. 2024.

"Lane Following Robot Using OpenCV." Hackster.io, www.hackster.io/Aasai/lanefollowing-robot-using-opencv-da3d45

"Golnaz Mesbahi - Lab 3."*Github.io*,2023, golnazmes.github.io/Lab%203.html. Accessed 11 Mar. 2024.

Samiei, Mandana. "Mandanasmi/Lane-Slam." GitHub, 3 Feb. 2024, github.com/mandanasmi/lane-slam. Accessed 14 Mar. 2024.

Rosebrock, Adrian. "AprilTag with Python." PyImageSearch, 2 Nov. 2020, pyimagesearch.com/2020/11/02/apriltag-with-python/.

"Exercise 3."*Ekhumbata.github.io*, ekhumbata.github.io/Adventures-in Duckietown/exercise3.html. Accessed 11 Mar. 2024.