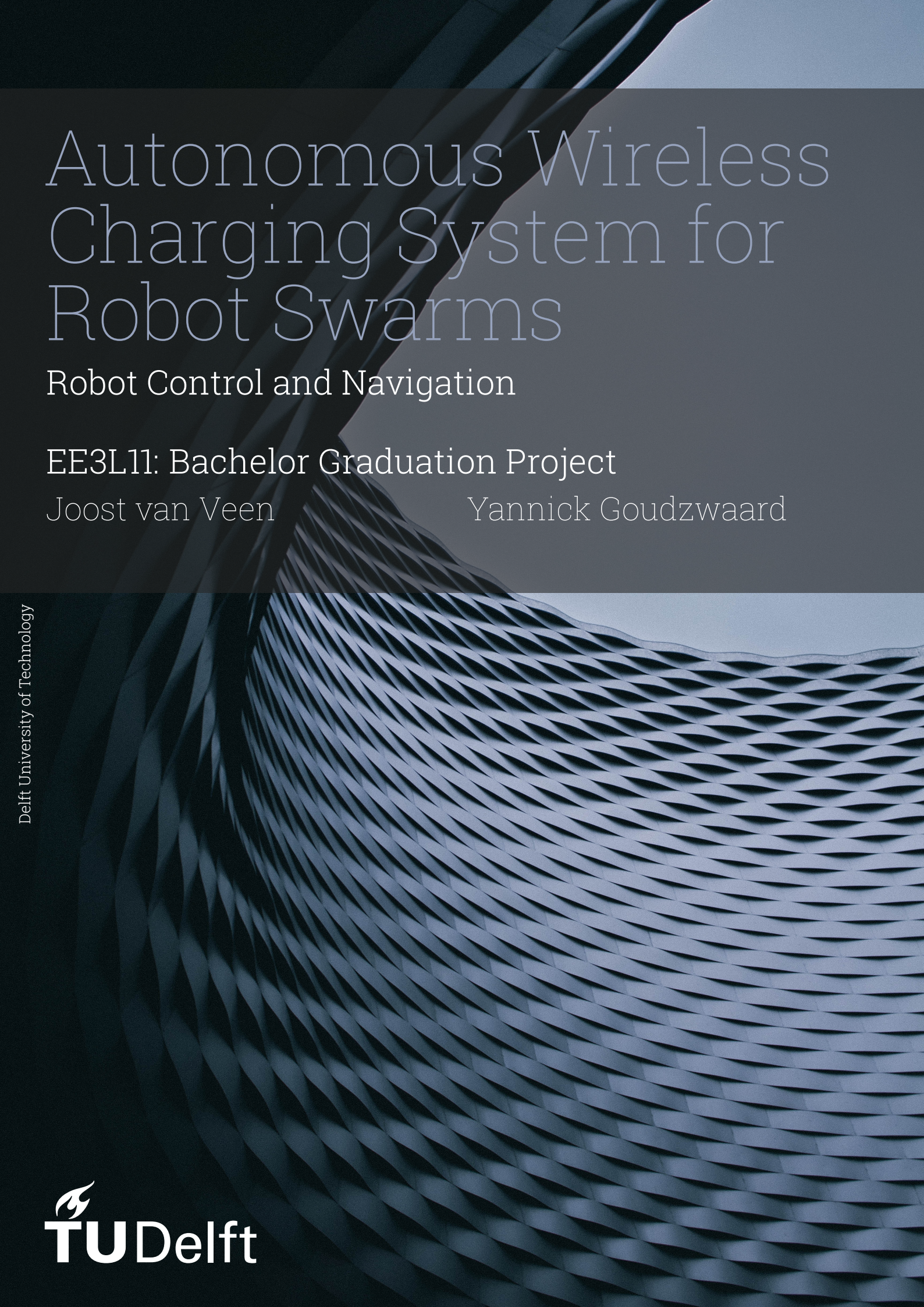


Autonomous Wireless Charging System for Robot Swarms

Robot Control and Navigation

EE3L11: Bachelor Graduation Project

Joost van Veen

Yannick Goudzwaard

Delft University of Technology

TU Delft

Autonomous Wireless Charging System for Robot Swarms

Robot Control and Navigation

by

Joost van Veen
Yannick Goudzwaard

Student Name	Student Number
Joost van Veen	5097428
Yannick Goudzwaard	5014751

Supervisor:	Ir. Dr. C.J.M. Verhoeven
Project Proposer:	Ir. Dr. C.J.M. Verhoeven
Project Duration:	April, 2023 - June, 2023
Faculty:	Electrical Engineering, Mathematics and Computer Science

TU Delft

Summary

The goal of this thesis is to develop a ROS package that facilitates the control and navigation of a Duckiebot robot. With the rise of robot swarms the need for autonomous charging system for robots is increasing. An implementation for decentralised autonomous behaviour for a Duckiebot for a wireless charging system in a Duckietown environment is discussed. The devised system is divided among three different modules and is implemented in ROS:

- An image recognition module;
- A navigation module;
- A motion control module;

The image recognition module uses linear image processing techniques and YOLO object detection in order to detect objects in images from the robots front facing camera. It detects traffic lights and road markings in order to tell the robot where to go.

The navigation module uses odometry to keep track of the robots current position. The odometry is reset in order to maintain accuracy. When the battery of the robot reaches a certain point the robot will decide to charge. It will then initiate path finding using Lee's algorithm in order to find a path to a charging park.

Finally the motion control processes all the information in order to drive the wheels of the robot. The system is thought to be able to navigate to a charging station, charge and then leave the charging station using the designed ROS package.

Preface

In the past 10 weeks we delved into the world of ROS, image recognition and path finding. With our group we together designed a wireless charging system for the Duckietown platform. We believed this experience to be valuable as it allowed for the freedom to make our own design choices and work as a group to achieve the goal of the project. The project served as a good learning experience and we hope that our implementation and methods can be used in further projects for the Duckietown or Lunar Zebro.

We would like to thank Ir. Dr. C.J.M. Verhoeven for proposing and supervising the project, and providing the facilities to be able to work on this project.

Joost van Veen
Yannick Goudzwaard
Delft, July 2023

Contents

1

Introduction

Robotic swarms are the future of large scale tasks or projects and almost all of these robots will run on batteries. When these batteries are low the robot will need to be charged. A system is proposed where the robots drive to a charging station where they are wirelessly charged by driving onto a charging pad. In order to realise such a system it is implemented using the Duckietown project.

The Duckietown project provides learning experiences in robotics and AI through their developers platform consisting of self-driving vehicles called Duckiebots and urban environment models called Duckietowns [1]. The platform is widely used by different educational institutions.

1.1. Goal of the Project

The goal of this project is to deliver a wireless charging system to serve as a basis for further development and improvements for other autonomous robot swarms such as the Lunar Zebro project. The most import goal is to provide a rule set for how the robots should behave around the charging station and to provide a behaviour on when a robot is allowed to charge. This basis is formed by implementing the autonomous wireless charging system on the Duckiebots in an adjusted Duckietown environment.

1.2. Problem Definition

In order to achieve the goal several things need to be implemented. The system has to be decentralised, therefore all decisions are to be taken by the robot. The system has to be implemented such that the swarm is up and operational 24/7 and no robots are left stranded without charge.

The project is divided among the following groups:

- Wireless charging hardware group — designs and implements the wireless charging hardware [2].
- Charging park design group — designs the charging park structure and defines the behaviour of the Duckiebots within the charging park [3].
- Control and navigation group — designs and implements a way for the Duckiebots to adhere to the predefined behaviour.

This thesis is about the control and navigation of the Duckiebots.

1.3. Introducing the Duckiebot

The Duckiebot DB-21M, as shown in Figure 1.1, is a differential-drive robot. Its two front wheels are driven separately by DC motors which allow the robot to rotate by varying the rotational speed of both wheels. It contains an omni-wheel at the back that provides stability. Each DC motor is equipped with a with an Hall effect based wheel encoder. An 8 Megapixels camera is mounted at the front, which has a field of view of 160 degrees. There are 2 LEDs mounted on both the front and the back of the robot [4].

Figure 1.1: The Duckiebot robot.

Additionally there are 2 sensors mounted on the Duckiebot, a ToF senor is mounted on the front and a Inertial Measurement Unit (IMU) is mounted on the bottom in between the wheels [4]. The ToF sensor has different modes and in its default mode can measure between 0.05 and 2 meters [5]. Lastly the Duckiebot has a NVIDIA Jetson Nano for running artificial intelligence algorithms and neural networks.

These sensors together with the information in camera images and the power of the Jetson Nano provide enough information and computational power for the robot to be able to navigate through the Duckietown environment.

1.4. The Duckietown Environment

The Duckietown environment models an urban environment. It is constructed out of three different layers:

- the floor layer defines the road network. The network is constructed as a tile map, in which each tile represents a specific road element. The available tiles are shown in Figure 1.2.
- the signal layer provides traffic signs, which enables the Duckiebots to exhibit different behaviour.
- the infrastructure layer consists of traffic lights and watch towers.

In this project only the provided floor layer will be used, as the charging park group developed their own traffic lights. Our group combines these traffic lights with self-designed traffic signs.

Figure 1.2: The 4 different road tiles used in Duckietown environments

1.5. Structure of the Thesis

This thesis is divided among a theory and implementation part. The theoretic part starts with Chapter 3, which introduces the idea of homogeneous coordinates and transformations. Chapter 4 utilises these coordinates to describe the pinhole camera model. Chapter 5 describes the image process techniques that are used within the project, after which Chapter 6 describes the process of object detection through the use of a neural network. In Chapter 7 the basis of the maze routing algorithm is explained, which the Duckiebot uses to find its way towards the entry point of the charging park. The theoretic part ends with Chapter 8, which describes pose estimation through the application of odometry.

The implementation part starts with Chapter 9, which gives an overview of the developed ROS package. In Chapter 10, the image processing and object detection techniques are combined to extract necessary information from the camera images, Chapter 11 the process of navigating the Duckiebot towards the entry point of the charging park, where it combines the aspects of maze routing, odometry and computer vision. The implementation part ends with Chapter 12, which describes the process of generating updated robot velocities from the inputs given by the image recognition and navigation modules.

At last Chapter 13 discusses the performance of the system and potential issues which could plague in the system, after which Chapter 14 concludes the project and gives future recommendations.

2

Program of requirements

The navigation and control is responsible for implementing the predefined behaviour. The requirements for the implementation are separated into functional and non-functional requirements, assumptions and key performance metrics. Assumptions are made about the environment of the robot. The functional requirements are things the behaviour has to do and the non-functional requirements are qualities which the behaviour must have. The key performance metrics define a quantifiable metric for performance requirements the system should adhere to.

Assumptions:

1. The road is assumed to only have the Duckietown environment markings and the coil line for coil alignment.
2. The surface of the Duckietown environment is assumed to be planar.

Functional Requirements:

[2.1.1] The Duckiebot must be able to autonomously drive through the Duckietown environment.

[2.1.2] The Duckiebot must be able to recognise when it has arrived at a charging park.

[2.1.3] The Duckiebot must be able to align itself with the charging pad.

[2.1.4] The Duckiebot must be able to find the entrance of the charging park.

[2.1.5] The Duckiebot must decide when it leaves the charging pad.

[2.1.6] The Duckiebot must prevent collision with other Duckiebots.

[2.1.7] The Duckiebot must stay within the lane that is driving in.

Non-Functional Requirements:

[2.2.1] The processing time of all inputs to the system should not exceed the frametime of the camera.

[2.2.2] The navigation and control module must be developed using the robot operating system (ROS).

[2.2.3] The object detection algorithm should be able to detect traffic lights with a confidence score of at least 0.5.

Key Performance Metrics:

- Input Processing Time
- Object Detection Confidence Score

Part I

Theory

3

Homogeneous Coordinates and Transformations

3.1. Introduction

In this project we make use of different transformations. In Euclidean space, each type of transformation is applied using a different mathematical operation:

- Translation is performed through the addition of some offset $\mathbf{t}$, such that $\mathbf{x}_1' = \mathbf{x}_1 + \mathbf{t}$.
- Rotation is performed through the multiplication of a rotation matrix R, such that $\mathbf{x}_1' = R\mathbf{x}_1$.
- Projection onto another vector $\mathbf{x}_2$ is performed through non-linear scaling, such that $\mathbf{x}_1' = \left(\frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\mathbf{x}_2 \cdot \mathbf{x}_2}\right)\mathbf{x}_2$.

Chaining different transformations together results in non-linear maths. By using homogeneous coordinates, each transformation can be performed using only matrix multiplication, which simplifies the concatenation of transformations [6]. This chapter gives an introduction to homogeneous coordinates. It gives the translation and rotation transformation matrices and discusses the application of combining translation and rotation which is used within the project. The transformation matrix for projection is discussed in Chapter 4.

3.2. Homogeneous Coordinates

Points in space can be described using homogeneous coordinates [7]. They can be thought of as Euclidean coordinates that have an extra dimension. If we describe Euclidean coordinates using parenthesis and homogeneous coordinates using square brackets, then the 2D space conversion between both representations is defined by

$$\begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \end{pmatrix} \longleftrightarrow \begin{bmatrix} x \\ y \\ w \end{bmatrix}. \tag{3.1}$$

Setting $w = 1$ yields the simplest form of the conversion.

By using homogeneous coordinates, linear transformations can be applied using a transformation matrix T, such that

$$\mathbf{x}_1' = T\mathbf{x}_1. \tag{3.2}$$

Due to the associative property of matrix multiplication, the application of multiple transformations can be chained into a single transformation matrix:

$$T = T_n T_{n-1} T_{n-2} \dots T_2 T_1. \tag{3.3}$$

3.3. Transformations

In this section we introduce the homogeneous forms of the translational and rotational transformations were we restrict ourselves to 2D space.

3.3.1. Translation

Translating the point (x_1, y_1) by an offset (x_t, y_t) results in a new point

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} x_1 + x_t \\ y_1 + y_t \end{bmatrix}. \tag{3.4}$$

In homogeneous coordinates translation is applied using the transformation matrix

$$T_{\text{translation}} = \begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.5}$$

such that

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 + x_t \\ y_1 + y_t \\ 1 \end{bmatrix}. \tag{3.6}$$

3.3.2. Rotation

Rotation about the origin is achieved by multiplication of a rotation matrix. For clock-wise (CW) and counter clock-wise (CCW) directions, the rotation matrices are defined by

$$R_{\text{CW-rotation}} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad \text{and} \quad R_{\text{CCW-rotation}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}, \tag{3.7}$$

which in homogeneous coordinates yield the transformation matrices

$$T_{\text{CW-rotation}} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad T_{\text{CCW-rotation}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.8}$$

where θ is the rotation angle.

3.3.3. Combining Translation and Rotation

The pose $\mathbf{q} = [x, y, \theta]^{\mathrm{T}}$ of the robot describes its rotation and translation with respect to a reference frame. By applying a rotation about the origin of the reference frame in counter clock-wise direction and then applying a translation, the pose can be represented by the transformation matrix

$$T = T_{\text{translation}} T_{\text{CCW-rotation}}, \tag{3.9}$$

$$= \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.10}$$

$$= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.11}$$

Poses are member of the Special Euclidean Group ($\mathbb{SE}$) [8]. The group $\mathbb{SE}(2)$ describes rigid motion in 2D [9] and is defined by

$$\mathbb{SE}(2) = \left\{ T = \begin{bmatrix} R & \mathbf{x} \\ \mathbf{0}_2^{\mathrm{T}} & 1 \end{bmatrix} \in \mathbb{R}^{3\times 3} \;\middle|\; R \in \mathbb{SO}(2), \mathbf{x} \in \mathbb{R}^2 \right\}, \tag{3.12}$$

where T is a transformation matrix that represents the pose, R denotes the counter-clock wise rotation matrix – which is part of the Special Orthogonal Group ($\mathbb{SO}$) – and $\mathbf{x}$ denotes the translation.

The transformation matrix in $\mathbb{SE}$ has the property [10] that

$$T^{-1} = \begin{bmatrix} R^{\mathrm{T}} & -R^{\mathrm{T}}\mathbf{x} \\ \mathbf{0}_2^{\mathrm{T}} & 1 \end{bmatrix}. \tag{3.13}$$

By approaching poses as transformation matrices, poses can easily be transformed between reference frames. Consider the situation given in Figure 3.1.

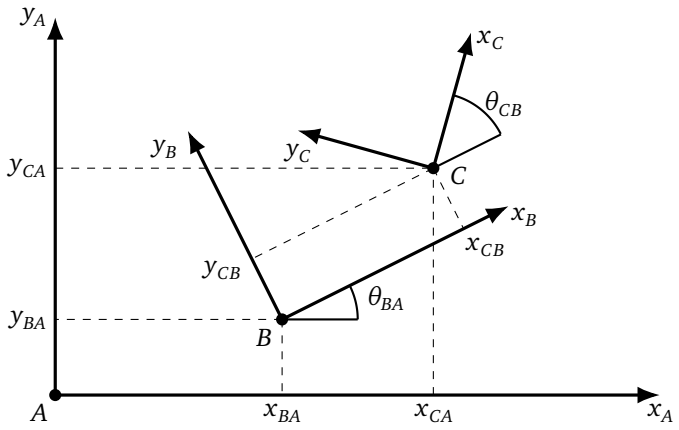


Figure 3.1: Visualisation of different reference frames.

If we denote the pose of reference frame B with respect to reference frame A by $T_{B\to A}$, then the pose of reference frame C with respect to reference frame A is described by

$$T_{C\to A} = T_{B\to A}T_{C\to B}, \tag{3.14}$$

$$= \begin{bmatrix} \cos(\theta_{BA}) & -\sin(\theta_{BA}) & x_{BA} \\ \sin(\theta_{BA}) & \cos(\theta_{BA}) & y_{BA} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_{CB}) & -\sin(\theta_{CB}) & x_{CB} \\ \sin(\theta_{CB}) & \cos(\theta_{CB}) & y_{CB} \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.15}$$

$$= \begin{bmatrix} \cos(\theta_{BA}+\theta_{CB}) & -\sin(\theta_{BA}+\theta_{CB}) & x_{BA}+x_{CB}\cos(\theta_{BA})-y_{CB}\sin(\theta_{BA}) \\ \sin(\theta_{BA}+\theta_{CB}) & \cos(\theta_{BA}+\theta_{CB}) & y_{BA}+x_{CB}\sin(\theta_{BA})+y_{CB}\cos(\theta_{BA}) \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.16}$$

where we used the angle sum identities

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta)+\cos(\alpha)\sin(\beta), \tag{3.17}$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta)-\sin(\alpha)\sin(\beta). \tag{3.18}$$

Therefore, the resulting pose becomes

$$\mathbf{q}_{C\to A} = \begin{bmatrix} x_{BA}+x_{CB}\cos(\theta_{BA})-y_{CB}\sin(\theta_{BA}) \\ y_{BA}+x_{CB}\sin(\theta_{BA})+y_{CB}\cos(\theta_{BA}) \\ \theta_{BA}+\theta_{CB} \end{bmatrix}. \tag{3.19}$$

4

Pinhole Camera Model

4.1. Introduction

The Duckiebot uses a forward facing camera in order to translate its 3D surroundings to a 2D image. Camera models are used in order to describe how the pixels in an image are determined from reflected light from real world objects. Cameras work by recording these reflected light waves. The camera that is mounted on the Duckiebot achieves this by the means of an array of light sensors. These sensor converts photons into electric signals which make up an image.

This chapter describes the pinhole camera model, which is the most common and simple method of modelling a camera [7], and discusses the principle of homography, which maps points from one plane to another plane.

4.2. Pinhole Camera Model

The pinhole camera model views the camera as if the photoelectric sensors are surrounded by a box which has one tiny hole in it. The only light that will therefore affect the image is light which passes through this hole. The hole can therefore be seen as the centre of projection. Figure 4.1 shows how a reflected light wave enters the camera through the pinhole.

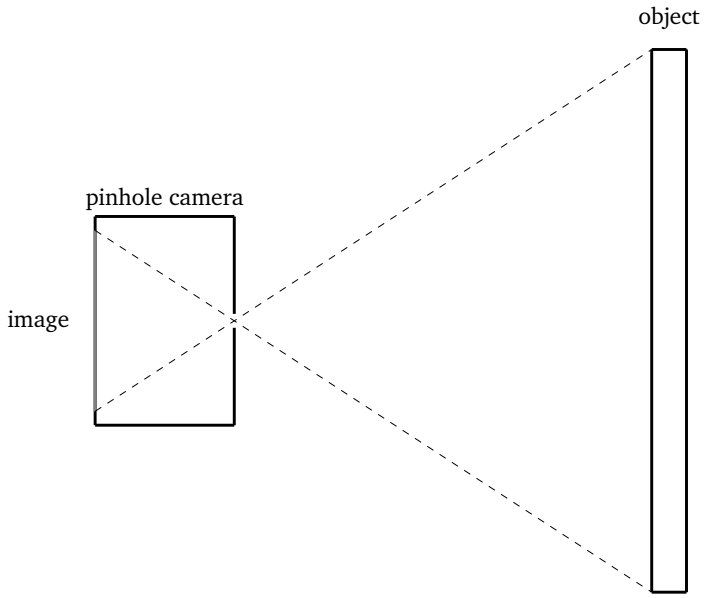


Figure 4.1: Pinhole camera: object to image relation.

Due to this structure, the resulting image will be inverted. Since most light is blocked by the surrounding box, the pixels will have a low intensity value. More light can be directed by using lenses, resulting in more exposure while maintaining the sharpness of the image. However, by using a lens the image becomes distorted [11].

The geometry of the pinhole camera model is visualized in Figure 4.2.

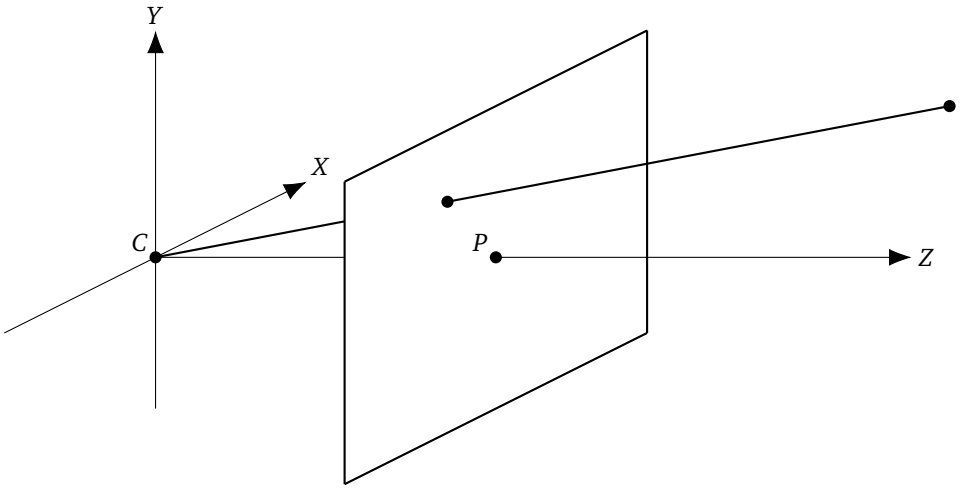


Figure 4.2: Pinhole camera model geometry.

The image plane has been moved in front of the centre of projection, such that we do not have to deal with inverted images while the corresponding mathematical operations remain the same.

The distance between the centre of projection and the image plane is known as the focal length of the camera. The origin of the reference frame of the image plane is known as the principal point. Projection is the combination of scaling a point by the focal length of the camera and translating it by the principal point. Therefore, the projection of a 3D point in the world onto a 2D image plane is defined by

$$\mathbf{x} = P\mathbf{X}, \tag{4.1}$$

$$= \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{4.2}$$

Here P is the camera projection matrix, which can be further decomposed into

$$\begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = K[I|\mathbf{0}], \tag{4.3}$$

where K is the intrinsic camera matrix.

The camera model should also account for camera rotation and translation with respect to the world frame. Therefore the point in the camera reference frame $\mathbf{X}_{\text{cam}}$ is described as $\mathbf{X}_{\text{cam}} = R\tilde{\mathbf{X}} + \mathbf{t}$, where R is the rotation matrix, $\mathbf{t}$ is the translation vector and $\tilde{\mathbf{X}}$ is the point $\mathbf{X}$ with respect to the world frame. This means that the pinhole camera model can be described as

$$\mathbf{x} = P\mathbf{X}_{\text{cam}} = K[I|\mathbf{0}] \begin{bmatrix} R\tilde{\mathbf{X}} + \mathbf{t} \\ 1 \end{bmatrix} = K[R|\mathbf{t}]\mathbf{X} \tag{4.4}$$

where the matrix $[R|\mathbf{t}]$ is referred to as the extrinsic matrix.

4.3. Homography

If the Duckietown grid is assumed to be a flat plane, the principle of homography can used to map a point in the image plane onto a point in the Duckietown ground plane. This is done by multiplying a 3×3 homography matrix H with the homogeneous representation of one point, which gives the homogeneous representation of a point in the other plane, such that

$$\begin{bmatrix} X \\ Y \\ W \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}. \tag{4.5}$$

The resulting transformation is shown in Figure 4.3

The homography matrix H is uniform for mapping any point from the image plane to the ground plane.

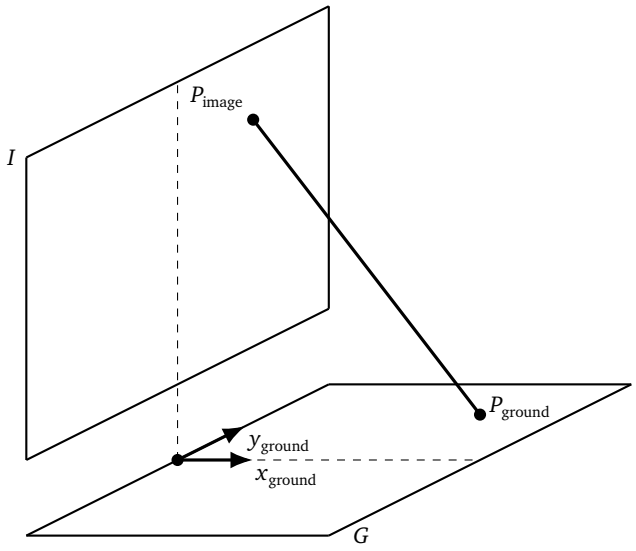


Figure 4.3: Transforming images points to the world plane.

4.4. Camera Calibration

In order to determine the intrinsic camera matrix, distortion coefficients and homography matrix corresponding to the camera model, the model needs to be calibrated. The camera is calibrated by means of the robot looking at a checker-board pattern where the size of the squares are known. By placing the pattern on the ground and looking at it from different angles and distances, the intrinsic camera and homography can be determined as well as the distortion coefficients. The intrinsic camera matrix and distortion coefficient are used to rectify the image [11].

5

Image Processing Techniques

5.1. Introduction

Image processing is the process of applying operations on an image in order to extract information from it. By processing the images captured by the camera of the robot, the robot is able to detect lane and intersection lines. This chapter describes the image processing techniques that have been used in the project.

5.2. Image Convolution

Most image processing techniques use convolution with the images in order to extract features of the image. Images are convolved with matrices called kernels. Kernels can be of different size and can highlight different features. The resulting output pixel is determined by the convolution of a pixel and its neighbourhood with a kernel. The size of the neighbourhood is determined by the size of the kernel as showed in Figure 5.1

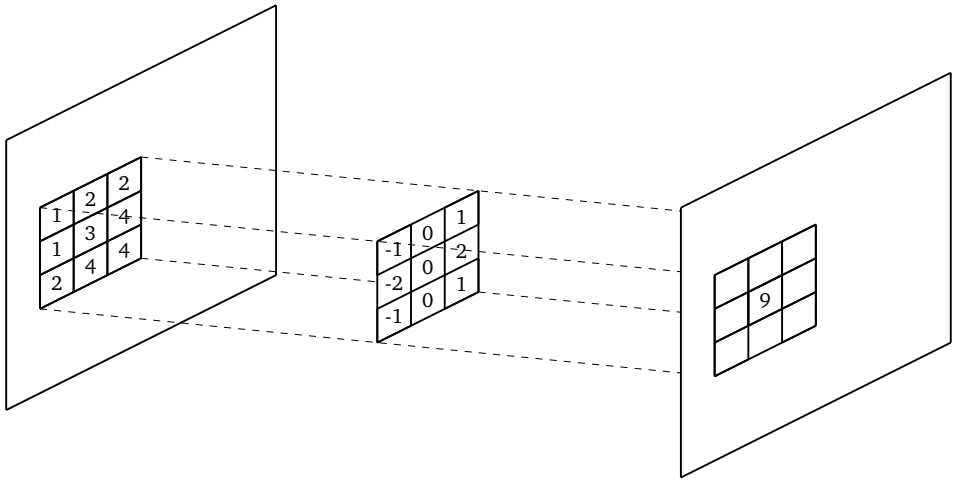


Figure 5.1: Iteration step of the convolution of an image with a 3 × 3 kernel.

5.3. Color Filtering

Since the different road lines all have different colours colour filtering is used to distinguish between these different lines. As shown in Figure 1.2, the yellow markers indicate the leftmost side of a lane, while the white line indicates the rightmost side of a lane.

The Duckiebot can also encounter horizontal lines of different colour:

- a red line indicates that the Duckiebot has arrived at an intersection.
- a blue line is used such that the Duckiebot can align itself with a charging coil.

The colours in the image can be filtered based on their values in different colour spaces. For selecting different colours in the image it is first converted into a HSV colour space since this will make filtering for a colour easier. This is due to how the HSV and RGB colour space work. Instead of being divided into a red, green and blue values a colour is divided into hue, saturation and value, this mostly has the effect that the colour is more defined by the hue and saturation while the value defines the intensity of the colour. This means that when looking at the road under different lighting condition for the HSV colour space only the value property would see significant change while for RGB all three values would change. Since it can not be expected that the environment will always have the same lighting condition the HSV colour space is far more reliable for filtering for different colours.

For the filtering itself a upper and lower value is set for each colour, then all pixel values which lie inside this bound are selected. This will return a mask in which all the pixels where the colour is the colour for which filtering was done are set to 1 and all other pixels are set to 0.

5.4. Gradient Edge Detection

Edge detection is the process of identifying edges inside of an image in which an edge is described as a sharp change in image brightness. The gradient describes the direction and intensity of this change. There are numerous different gradient edge detection filter like Sobel, Laplacian of Gaussian, Prewitt and Canny just to list a few of the most common. Of the different edge detection filters Canny is thought of by most to be the most optimal [12][13], however it is far more complex then regular Sobel edge detection. Canny edge detection works by first applying a Gaussian blur to the image. This is done to remove noise from the image before detecting the edges. Then Sobel edge detection with two 3×3 sized kernel is done to find the horizontal and vertical gradients, the used kernels can be seen below:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{5.1}$$

Using the horizontal and vertical gradients the gradient magnitude can be calculated using the formula $|G| = |G_x| + |G_y|$. It then places a threshold on the gradient magnitude to further reduce noise and find the main strong edges of the image. It then detects potential edges and removes weaker edges which are not connected to the stronger main edges by looking at neighbourhood of that edge.

For using edge detection for finding lane markings just Sobel edge detection should be enough due to the roads having very little noise. Most noise in the image will be above the horizon or have small gradient magnitudes, both of which can be filtered out. The Gaussian blur can still be applied in order to reduce the little noise that might in the image.

6

Convolutional Neural Networks and YOLO Object Detection

6.1. Introduction

For detecting more complex objects in the image, which may be surrounded by significantly more noise then the road markings, object detection is commonly implemented. It using neural networks and deep learning algorithms to locate objects within the image.

6.2. Convolutional Neural Networks

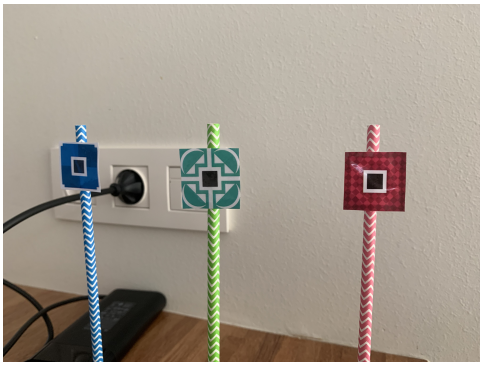
Neural networks are part of machine learning and are the centre of deep learning algorithms. They are comprised of different layers, each layer identifies properties of the data and assigns weights to these properties. Convolutional Neural Networks (CNN) use different layers to identify properties of an image and assign these properties to a class. CNN use 3 different type of layers; convolution, pooling and connected layers. Convolution layers apply multiple convolution filters to the image, each filter is meant to extract some features from an object in an images. These layers are commonly combined with Rectified Linear Units (ReLU) which maps all negative values to zero and maintaining positive values. This is then followed by a pooling layer which performs non-linear down sampling in order to simplify the output. Multiple of these layers in row allow the network to detect all the features in the image. After the features are extracted the data is put through a connected layer, this predicts the class of the data. So in short a CNN works by extracting the features of an images and then passing these through a classification layer which leads to class prediction. The strength of a CNN is the ability to process large amounts of data like images.

6.3. YOLO Algorithm

You Only Look Once (YOLO) is an object detection algorithm using CNN designed by Joseph Redmon, Santosh Divvala, and Ross Girshick [14]. YOLO mostly differs from other detection methods like Regional Convolutional Neural Networks (R-CNN) and Deformable Parts Models (DPM) because it gives the bounding box coordinates and class probabilities directly from the images pixels. Other approaches, like R-CNN, first look for possible bounding boxes and then try and classify the bounding boxes. YOLO therefore has the advantage that only a single neural network is needed for object detection. This makes YOLO faster than faster R-CNN and other object detection algorithms [14][15]. However YOLO does have a drawback that it is less accurate at locating the bounding boxes for objects [14].

6.3.1. Model Training

In order to create a model a dataset is required to train the model for the objects it needs to detect. A dataset for YOLO has a `.txt` file for each image. This file contains the bounding boxes for each object in the image for which the model needs to be trained. This file has one row per object in the image with each row consisting of the class of the image, the x and y coordinates of the centre of the bounding box and the width and height of the bounding box. For example an image and `.txt` file for a dataset will look something like in Figure 6.1.



```
0 0.160055 0.455105 0.091328 0.142681
1 0.433118 0.487085 0.128229 0.172202
2 0.779059 0.494465 0.131919 0.164822
```

Figure 6.1: Visualisation of image and `.txt` file for dataset definition.

The data set is then split into a training set and validation set. The training set is used to train the model and then the validation set is used to measure the performance of the model. In order to train the model the batch size and epochs need to be set. The batch size determines how many images the model goes through before changing the model parameters. The number of epochs is how many times the model will go through the whole data set. The values for these parameters can not be perfectly determined before hand, but only by validating the model.

6.4. Validating Model Performance

In order to validate the performance of a model the validation and training loss of the model is important. This will show if the model is underfitted, overfitted or a good fit. Underfitting means that the model is unable to model the training data. Overfitting means that works well with the training data but poorly on new data. A good fit means that the model works well on both the training data and new data. The loss is the errors the model makes for a set, so the training loss is the errors the model makes with the training data while the validation loss is for the validation data. For the YOLO model three loses are calculated: the localisation loss, the objectness loss and the classification loss. The localisation loss is the error in detecting the bounding box location and size. The objectness loss is the error in detecting objects in the image. Lastly the classification loss is the error of classifying an object in the image. So the localisation loss is important for finding where the object is in the image. The objectness loss is important for finding objects within the image and the classification loss is important for the correct classification of these objects.

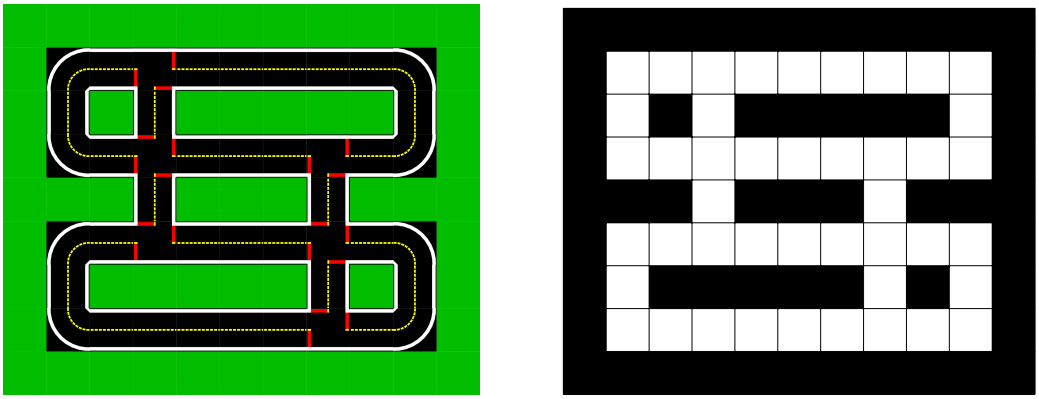
Another way of validating a model is by looking at the precision and recall. The precision is how many of the predictions that were made are correct and recall is how many of objects which were present where correctly predicted. This can also be done by looking at the mean average precision (mAP). The mAP is the mean of the average precision (AP) of all the classes [16]. The AP defines the precision-recall curve into a single value. The mAP is a good indicator of the performance of a model since it shows how good it is at detecting objects and correctly classifying them. Still the best way to test if a model works well is by testing it on new data.

7

Maze Routing Algorithm

7.1. Introduction

Due to the fact that the structure of a Duckietown is based upon a tile map, finding a route from one tile to another tile can be thought of as a maze routing problem. This analogy is visualized in Figure 7.1. Examples of path finding algorithms are the A-star (A*), Dijkstra and Breadth-First Search (BFS) algorithms. In terms of computer processing and execution time, Permana et al. [17] determined that the A* algorithm can be considered to be the best maze routing algorithm in general. However, they also demonstrated that in some cases the A* algorithm has a longer execution time than both the Dijkstra and BFS algorithms.



(a) (b)

Figure 7.1: Visualisation of analogy between grid (a) and maze (b) representations of a Duckietown.

In this project we will utilize the algorithm developed by Lee [18][19], which essentially uses the BFS algorithm to find an optimal path. In this chapter we describe the workings of the Lee algorithm.

7.2. Definitions

Maze Array

A maze can be represented as a two-dimensional array, where each cell represents a specific tile. Cells that represent the walls of the maze have a value of -1, and the remainder of cells are assigned the value 0. A cell is located at row i and column j, such that its position can be denoted by (i, j). Rows and columns are zero indexed.

Start- and Target Cell

The start and target cell represent the beginning and end of the resulting path respectively.

Cell Neighbours
Each cell in the maze array has up to 4 neighbours:

- a northern neighbour at $(i-1, j)$.
- an eastern neighbour at $(i, j+1)$.
- a southern neighbour at $(i+1, j)$.
- a western neighbour at $(i, j-1)$.

A neighbour is considered valid when it respects the boundaries of the grid and has a positive associated value.

Visitor Queue
The visitor queue is a queue in which nodes are added that need to be processed by the Lee algorithm. The benefit of using a queue is such that we do not need to iterate over all cells within the maze array, which reduces the number of operations needed to perform the algorithm and thus reduces its execution time.

Initially, the visitor queue is empty.

Path List
The path list contains the cells that together form the path resulting from the Lee algorithm.

Breadth-First Search
Breadth-first search is a searching technique that searches for a node within a tree by starting at the root of the tree and visiting every node within the current depth level before moving on to nodes at the next depth level. The technique always finds a solution node given that it exists.

7.3. Maze Routing Procedure

This section discusses the procedure of applying the Lee algorithm to an arbitrary maze, where we take the maze in Figure 7.2 as the maze to be solved. The start and target cells are assigned the colors cyan and orange respectively.

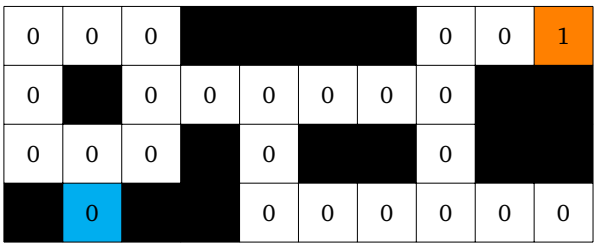


Figure 7.2: State of the maze array before BFS.

As shown in Figure 7.2, the Lee algorithm starts with assigning the value 1 to the target cell. The target cell is then added to the visitor queue, after which the BFS algorithm is applied.

7.3.1. Application of the BFS Algorithm

The breadth-first search application performs the following steps:

1. The first element in the visitor queue is de-queued.
2. Each valid neighbour of the de-queued cell is assigned the value of the de-queued cell incremented by one.
3. Each valid neighbour is added to the visitor queue.

This process, of which a single iteration is visualized in Figure 7.3, repeats itself until one of the following conditions is satisfied:

(i) The start cell holds a value greater than 0. This indicates that there exists a path from the start cell to the target cell.

(ii) The visitor queue is empty after performing a procedure iteration. This indicates that there does not exist a path from the start cell to the target cell.

In both cases, the visitor queue is emptied after completion of the BFS algorithm.

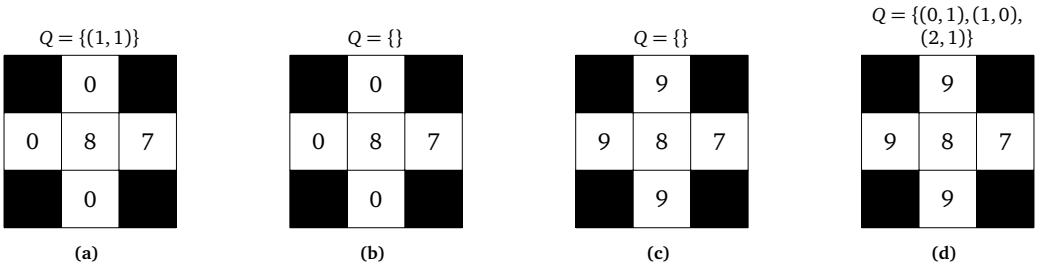


Figure 7.3: Iteration step within the BFS part, where (a) is the start situation, and in which (b) the first element is dequeued, in (c) the valid neighbors are incremented, and in (d) the valid neighbors are added to the visitor queue.

In case of Figure 7.2, the result after BFS is shown in Figure 7.4.

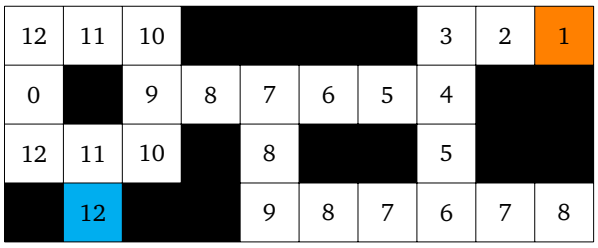


Figure 7.4: State of the maze array after BFS.

7.3.2. Path-tracing

The result of the BFS procedure is that every valid cell has one or more neighbors with a lower value. Using this fact a path can be traced from the start cell to the target cell.

After the start cell is added to the visitor queue, the path-tracing procedure follows similar steps to that of the BFS procedure:

1. The first element in the visitor queue is de-queued and is added to the path list.
2. Each valid neighbour is checked for a lower value.
3. The first neighbour with a lower value is added to the visitor queue.

The path tracing procedure ends when the visitor queue is empty. The resulting path in case of Figure 7.2 is shown in Figure 7.4.

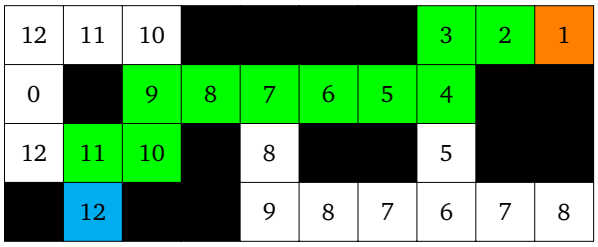


Figure 7.5: Resulting path.

8

Odometry of Differential-Drive Robots

8.1. Introduction

Odometry allows one to estimate the pose of a robotic agent — which we define as a vector that contains the position and orientation of the agent with respect to a reference frame — using its available motion sensors. It is a simple and cheap option for determining the momentary position of a mobile robot [20]. This chapter describes the application of odometry to a Duckiebot, alongside conflicts that arise over time.

8.2. Duckiebot Geometry

The Duckiebot DB21-M is a differential-drive robot consisting of one omni-wheel and two front-wheels of radius R (note that in reality, the radius of both wheels will vary). The reference frame of the Duckiebot is chosen such that its origin is at the center of the baseline B between both wheels. If L represents the distance between the reference frame and each wheel, the baseline can be expressed as $B = 2L$. The resulting geometric representation is shown in Figure 8.1.

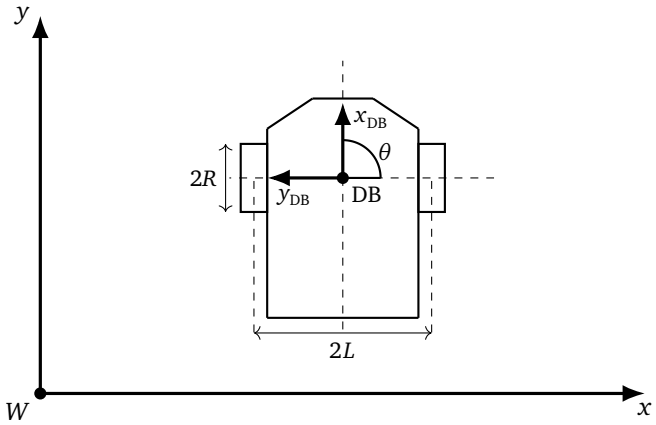


Figure 8.1: Relevant geometry of the Duckiebot for the application of odometry.

8.3. Duckiebot Application of Odometry

Each of the DC motors of the Duckiebot is equipped with a Hall effect sensor wheel encoder. The encoder contains a magnetic ring in which $2N$ magnetic poles are equally spaced. The Hall effect sensor then sends N pulses (which we call ticks) per motor turn [21]. In case of the Duckiebot, the Hall effect encoders have a resolution of $N_{\text{max}} = 135$ ticks per revolution.

Suppose that the wheel encoder sends $N_{\Delta t}$ ticks in a time interval of Δt seconds. Then the rotation of the

wheel becomes

$$\Delta\phi = \frac{N_{\Delta t}}{N_{\max}} \cdot 2\pi, \tag{8.1}$$

yielding an angular velocity of $\omega = \Delta\phi/\Delta t$, assuming that the wheel has a constant linear speed.

By measuring the angular velocity of each wheel, the linear and angular displacement can be determined from which the pose can be estimated [22]. The linear and angular velocity of the reference frame of the robot become

$$v_{\mathrm{DB}} = \frac{\omega_{\mathrm{L}}R + \omega_{\mathrm{R}}R}{2} \quad \text{and} \quad \omega_{\mathrm{DB}} = \frac{\omega_{\mathrm{L}}R - \omega_{\mathrm{R}}R}{2L}, \tag{8.2}$$

which yields the linear and angular displacements

$$d = v_{\mathrm{DB}}\Delta t \quad \text{and} \quad \Delta\theta = \omega_{\mathrm{DB}}\Delta t, \tag{8.3}$$

where the subscripts L and R distinguish the left- and right front wheels.

The resulting pose relative to the world reference frame W is estimated by

$$x(t+\Delta t) = x(t) + d\cos[\theta(t+\Delta t)], \tag{8.4}$$
$$y(t+\Delta t) = y(t) + d\sin[\theta(t+\Delta t)], \tag{8.5}$$
$$\theta(t+\Delta t) = \theta(t) + \Delta\theta. \tag{8.6}$$

8.4. Validity of Odometry

The application of odometry is based upon simple equations that assume that wheel revolutions can be related linear displacement of the robot. However, this assumption is of limited validity [23].

Errors that accumulate over time, such as the possible misalignment of the wheels, are known as systematic errors. Non-systematic errors are due to the environment that the robot is driving in, such as an uneven floor. The predominance of either systematic or non-systematic errors depends on the surface that the robot drives on [20]. In case of the Duckietown environment (which can be seen as a smooth even surface), systematic errors will be the predominant type of error source. Due to the accumulation of systematic errors over time, the odometry model requires a reset using an absolute position update [20] in order to maintain accuracy.

By calibrating the odometry model, the error propagation can be reduced [24]. By formulating the linear velocity of each wheel as being dependent of the linear and angular velocities of the robot reference frame, the corresponding angular velocity of the motors can be determined [25].

The calibration method proposed by the Duckietown project [26] relates the previous denoted linear and angular velocities to a trim variable. By adjusting the trim variable, the Duckiebot calculates the radius of each wheels and the baseline parameter. Initially, the wheels are assumed to be of equal radius, such that the trim variable has a value of zero.

Then the calibration is performed using the following procedure:

1. The motors are given the commands that correspond to the robot frame driving in a straight line.
2. Due to improper calibration, the Duckiebot drifts away from the straight line. By measuring the deviation from the line, the trim variable can be adjusted, which affects the driving command given in step 1.
3. Repeat the previous steps until a satisfactory result is achieved.

When the calibration has been completed successfully, the Duckiebot stores the parameters of the odometry model in memory.

Part II

Implementation

9

Introduction to the ROS package

9.1. Introduction

This chapter discusses the basic principles of the robot operating system (ROS). The Duckietown platform natively provides some functionality upon which the designed ROS package is build.

9.2. The Robot Operating System (ROS)

The robot operating system (ROS) is an open source software tool that is used to implement robotic applications. It forms the bridge between software and hardware. A ROS network consists of nodes – which are small executable programs – that can communicate with each other using ROS topics. A node can write specific data messages to a topic, which is called publishing. It can also subscribe to a topic. In that case, whenever the data message of the topic changes, the node can execute a function as a callback. Note that nodes can publish and subscribe to multiple topics.

9.3. Duckiebot Native Functionality

The following scripts that the Duckiebot provides are build upon by the designed package:

- `DeadReckoningNode` — applies odometry in order to estimate the pose of the Duckiebot with respect to its initial reference frame.
- `controller` — translates the error between centre of the lane into linear and angular velocities using a PID controller.

9.4. Overview of the Created ROS package

A ROS package is made to implement the robot behaviour. It is build up out of multiple ROS nodes which communicate which each other through multiple newly created topics. The ROS nodes can be divided into 3 categories: Image recognition, Navigation and Motion & SoC control. An overview of the how the categories are connected can be seen in Figure A.1.

9.5. The SoC Control

The State of Charge controller reads the battery info the battery publishes, extracts the SoC from the info and publishes it. Additionally it also receives the coil alignment error. If the error is within a threshold it means the coils are properly aligned and it sends the enable to the receiver side of the charging system in order for the robot to enable charging. If the SoC reaches the maximum mission SoC the robot disables the charging.

10

Image recognition

10.1. Introduction

The robot uses the camera to detect multiple objects. On the road it needs to detect the lines who indicate the edge of a lane. It also needs to detect intersection lines on the ground. This tells the robot when it is at an intersection in order to reset the homography and make turns. For charging the robot should also be able to detect a charge line. Then homography is applied and by comparing the distance to the line with a reference to distance it is able to align with the coil to enable charging. Some of the detection algorithms will use so called horizon masks, these masks use homography to filter out pixels which are more then a certain distance away from the Duckiebot.

Another thing the robot needs to detect is if there is a traffic light in the image. This is done by using a YOLO model which is trained for the different type of traffic lights.

10.2. Detecting Intersection Lines

Since odometry is only accurate for short distances it needs to be reset in order to limit the error. This is done at intersections since their locations are easily detected by the robot and the location in the world frame is known. The robot knows it is at an intersection if it detects a intersection line. Intersection lines are red and span across the road horizontally.

Other than the intersection lines there are no other red lines so only filtering for the road and that colour should result in no other objects being detected. They are detected by looking beneath the horizon for red patches. If a patch has 4 or more corner points and the centre of the patch is close to the centre of the lane then it assumed to be an intersection line. This filters out other intersection lines of the intersection or red objects which are not located on the road.

Each time the image is updated the node applies a colour filter for red HSV values resulting in a red colour mask. This mask is multiplied with a horizon mask set at a distance of 0.3 meters. This is done since the robot should only react to the intersection lines if it is close and to filter out other uninteresting lines. After this contours are detected in the combined mask. It then goes through each contour, if any are found, and determines if one could be a intersection line. This is done by checking if the contour is near the centre of the road. If so it will publish that it has detected a intersection line and will calculate the pose of the Duckiebot with reference to the intersection.

This is done by applying homography to the points in order to get the position with reference to the Duckiebot. Then the centre of the intersection line with reference to Duckiebot is calculated using those points. The angle of Duckiebot is determined by the angle of a line drawn between the top right and bottom right points of the intersection line or the top left and bottom left points of the intersection line. It decides if it looks at the left or right points on the basis of the y-value of the position of the intersection with reference to Duckiebot. If the y-value of the centre is larger then zero it will look at right points otherwise it will look at the left points. This is due to where the robot points in reference to the centre of the intersection line. As seen in Figure 10.1 the angle of the Duckiebot with reference to the intersection is the same as the angle of

the right 2 intersection line points with reference to the Duckiebot. The distance between the Duckiebot and the intersection line is used to find the pose of the Duckiebot with reference to the intersection.

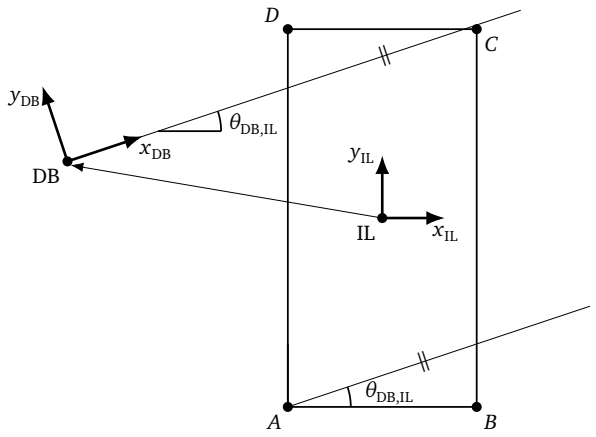


Figure 10.1: Duckiebot pose with reference to the intersection

10.3. Lane Detection

If the robot is not at an intersection it should drive forward and stay within its lane. However even if the robot starts in the middle of the lane facing perfectly forward it will still drift to the left or right. This is due to the wheels and motors not being perfect. The motor commands might tell the motors to both drive at exactly the same speed but there will still be some drift. Combined with the omnidirectional wheel, which has a lot of resistance and runs far from smooth, the robot needs to make corrections in the direction it is driving. Therefor a lane detector is implemented, the robot is able to see the lanes, from which it can determine where the middle of the road is. The robot does this by detecting the right edge of the left lane and left edge of the right lane. These edges are found by combining multiple masks, a mask which filters colour, a horizon mask and a edge detection mask. The colour masks are created by HSV colour filtering the image, which results in two masks, one for the yellow of the middle line and one for the white of the right line. The horizon mask is set at one meter.

For the edge mask multiple masks are used to get the edges which correspond with the lanes. First the horizontal and vertical edge gradients are calculated using Sobel edge detection on an Gaussian blurred image. These are used to make a gradient magnitude mask by calculating the gradient magnitude for all pixels and removing all magnitudes below 40. This is done since the contrast between the dark colour of the road and the light colour of the lanes results in high gradient magnitude, so all low values are considered noise. After this masks are made of only the positive horizontal and negative horizontal gradient values. The same is done for the vertical gradient values. This is done in order to filter for certain gradient angles. Lastly two masks are created which select either the left or right half of the image.

The left edge is detected by combing the yellow colour, negative horizontal and negative vertical gradient, gradient magnitude and horizon masks and looking at the left side of the image. The right edge is detected by combing the white colour, positive horizontal and negative vertical gradient, gradient magnitude and horizon masks and looking at the right side of the image. The masks and the resulting combined mask can be seen in Figure 10.2.

The ideal path is found by finding the points which lie in between these lines as seen in Figure 10.2. By plotting a line through the points and finding the angle of this line with the y-axis of the image can be calculated. This angle is published so it can be used to determine the robots velocities.

10.4. Detecting Coil Alignment Lines

For aligning the coil the detection is almost the same as the intersection detection, however now it is not filtered for the color red but blue. The robot detects the corner coordinates the same way it does for the intersection lines. It then publishes these coordinates for the motion control.

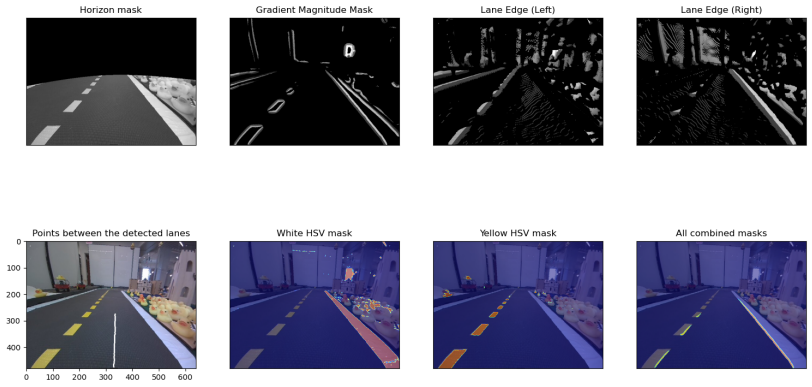


Figure 10.2: The different masks used for lane detection and the middle of the lane found using the masks

10.5. Object Detection

The Duckiebot needs to detect traffic lights in order to know if it can cross the intersection. It also needs to detect traffic lights to know if it is at a charging park or station and if they are occupied or not. This is done using YOLO object detection.

10.5.1. YOLOv5 Model

The YOLOv5 model is used for detecting traffic lights in images. YOLO is used since it is fast and easy to train. The only drawbacks of using YOLO is that the bounding box location can be inaccurate, however since the location of the traffic light in the images is unimportant this is not a problem.

10.5.2. Model Training and Performance

After the data set is created the model can be trained. The data set used for training the model consist of 90 images for training and 19 for validation. The batch size is set at 64 and the epoch amount is increased until a good model is achieved. This can be seen in the graphs for training and validation losses The resulting training and validation losses for 300 epochs can be seen in Figure 10.3.

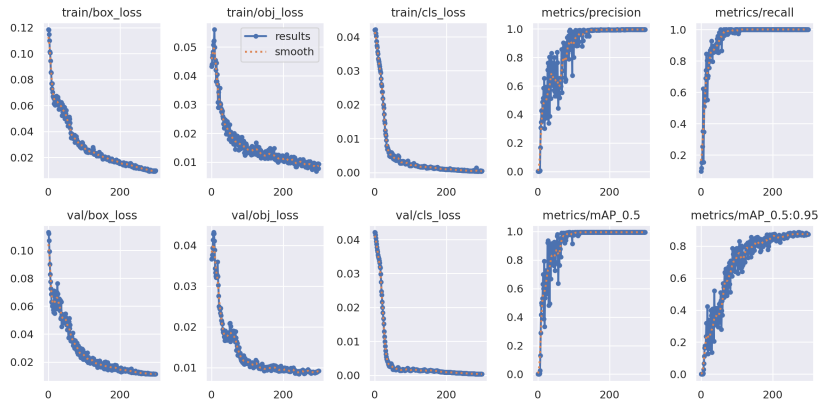


Figure 10.3: Training and validation losses, precision and recall and mAP graphs of the model.

The x-axis of the graphs show the result at each epoch. In Figure 10.3 it can be seen the training en validation classification loss `cls_loss` as well as the objectness loss align well and both bottom out at around 100 epochs. The loss will then only decrease slightly for each extra epoch. The localisation loss `box_loss` is not interesting to us since the location of the traffic light does not matter. Further model results can be seen in Appendix B.

The model with 100 epochs was tested using a webcam to determine the accuracy on new data. It was found that the model is good at detecting traffic lights. The detected traffic light had a confidence around 0.9 and the model was able to classify the traffic lights correctly.

11

Navigation

11.1. Introduction

The goal of this chapter is to describe the implementation of letting the Duckiebot navigate itself to the charging park. It combines the aspects of maze routing, odometry and computer vision to achieve this.

11.2. Lee Algorithm Adjustment

One factor that the Lee algorithm does not account for is the orientation of the Duckiebot. There exists a possibility that the algorithm determines a path for which the Duckiebot would need to switch lanes and reverse its direction. This would however result in the fact that requirement [2.1.7] would not be satisfied. To overcome this problem, the BFS part of the Lee algorithm has been adjusted such that the tile behind the Duckiebot cannot update the start cell. To avoid having to deal with complex cases, the Lee algorithm is only initiated when the Duckiebot is driving on a straight road tile.

Suppose that a Duckiebot is facing southwards, as shown in Figure 11.1. Since the tile at row 1 and column 7 is directly behind the Duckiebot, it is not allowed to update the start cell. The maze routing algorithm continuous and provides the path as shown in the figure.

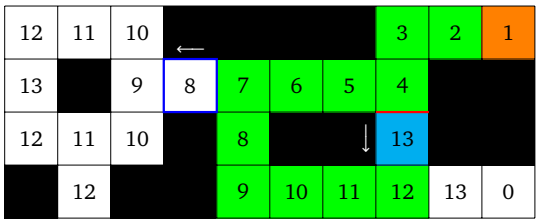


Figure 11.1: Preventing value assignment from behind the Duckiebot.

However, applying this adjustment can result in the algorithm not being able to find a path at all. In case of Figure 11.1, this problem would arise for the case that the Duckiebot is at the tile at $(1,3)$, which has been surrounded by a dark blue box. This issue can be resolved by taking one of three measures:

1. The structure of the Duckietown should be selected such that this problem never arrises, as is the case for the Duckietown present in Figure 7.1. This is the measure that was taken during the project.
2. In the case that no path is found, the Duckiebot should try to find a path when it arrives at a different straight tile.
3. Neglect requirement [2.1.7] and allow the Duckiebot to turn and switch lanes.

11.2.1. Intersection Instructions

Since the robot can drive along straight and curved road sections on its own, we decided to transform a determined path into a queue of instructions corresponding to specific intersections.

We distinguish the following intersection instructions:

- `STRAIGHT` — The Duckiebot should follow a straight path.
- `LEFT` — The Duckiebot should steer to the left with smaller steering angle.
- `RIGHT` — The Duckiebot should steer to the right with larger steering angle.

In Figure 11.1, the path consists of the intersections $I = \{(3,7),(1,4),(1,7)\}$. Given the direction of the Duckiebot and the path itself, the resulting instructions yield a queue $Q = \{\texttt{RIGHT}, \texttt{RIGHT}, \texttt{LEFT}\}$.

11.3. Applications of Odometry and Homography

The goal of applying odometry is to determine the pose of the Duckiebot with respect to the Duckietown, which is the information that the maze routing algorithm needs in order to find a suitable path.

11.3.1. Reference Frame Transformation

The Duckiebot is provided with an odometry package, which utilizes the wheel encoders for both wheels in order to estimate the pose of the Duckiebot. Figure 11.2a shows the pose that the odometry package publishes. In order to obtain the pose of the robot with respect to the Duckietown frame, as shown in Figure 11.2b, the odometry reference frame must be translated and rotated.

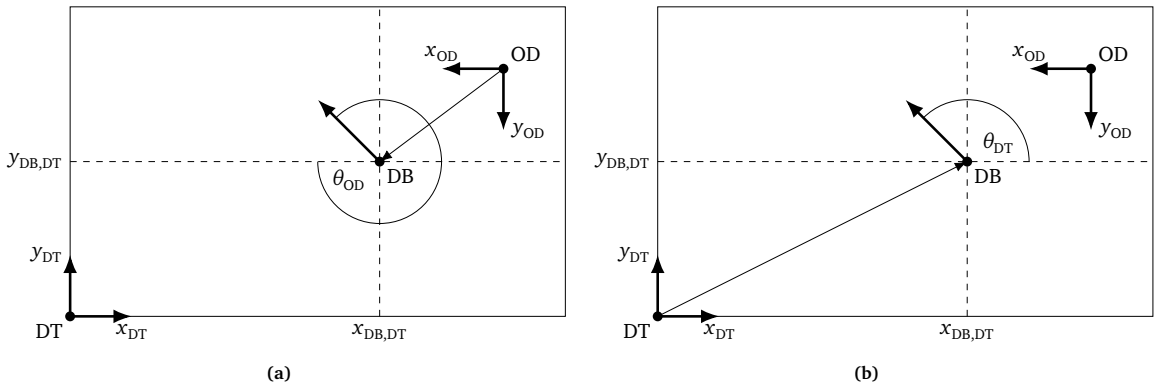


Figure 11.2: Poses of the Duckiebot with respect to (a) the odometry reference frame and (b) the Duckietown reference frame.

If we assume that the initial pose of the Duckiebot with respect to the Duckietown reference frame is known, then so is the pose of the odometry reference frame. By applying the theorem explained in Chapter 3, the transformation matrix corresponding to the pose of the Duckiebot with respect to the Duckietown reference frame becomes

$$T_{\text{DB}\to\text{DT}} = T_{\text{OD}\to\text{DT}}\, T_{\text{DB}\to\text{OD}}, \tag{11.1}$$

$$= \begin{bmatrix} \cos(\theta_{\text{OD,DT}} + \theta_{\text{OD}}) & -\sin(\theta_{\text{OD,DT}} + \theta_{\text{OD}}) & x_{\text{OD,DT}} + x_{\text{DB,OD}}\cos(\theta_{\text{OD,DT}}) - y_{\text{DB,OD}}\sin(\theta_{\text{OD,DT}}) \\ \sin(\theta_{\text{OD,DT}} + \theta_{\text{OD}}) & \cos(\theta_{\text{OD,DT}} + \theta_{\text{OD}}) & y_{\text{OD,DT}} + x_{\text{DB,OD}}\sin(\theta_{\text{OD,DT}}) + y_{\text{DB,OD}}\cos(\theta_{\text{OD,DT}}) \\ 0 & 0 & 1 \end{bmatrix}, \tag{11.2}$$

where $\theta_{\text{OD,DT}}$ is the angle of the pose of the odometry reference frame with respect to the Duckietown reference frame.

Then the resulting pose becomes

$$\mathbf{q}_{\text{DB}\to\text{DT}} = \begin{bmatrix} x_{\text{OD,DT}} + x_{\text{DB,OD}}\cos(\theta_{\text{OD,DT}}) - y_{\text{DB,OD}}\sin(\theta_{\text{OD,DT}}) \\ y_{\text{OD,DT}} + x_{\text{DB,OD}}\sin(\theta_{\text{OD,DT}}) + y_{\text{DB,OD}}\cos(\theta_{\text{OD,DT}}) \\ \theta_{\text{OD,DT}} + \theta_{\text{OD}} \end{bmatrix}. \tag{11.3}$$

11.3.2. World to Grid Conversion

Given that a certain $m \times n$ Duckietown grid contains tiles which sides have length s, the position $(x_{\text{DB,DT}}, y_{\text{DB,DT}})$ of the Duckiebot can be converted into the grid coordinates

$$\text{row} = \left\lfloor \frac{m \cdot s - y_{\text{DB,DT}}}{s} \right\rfloor, \tag{11.4}$$

$$\text{column} = \left\lfloor \frac{x_{\text{DB,DT}}}{s} \right\rfloor. \tag{11.5}$$

Since the maze routing algorithm only considers NESW-directions, the orientation of the robot must be mapped onto such direction. The approximations are shown in Figure 11.3.

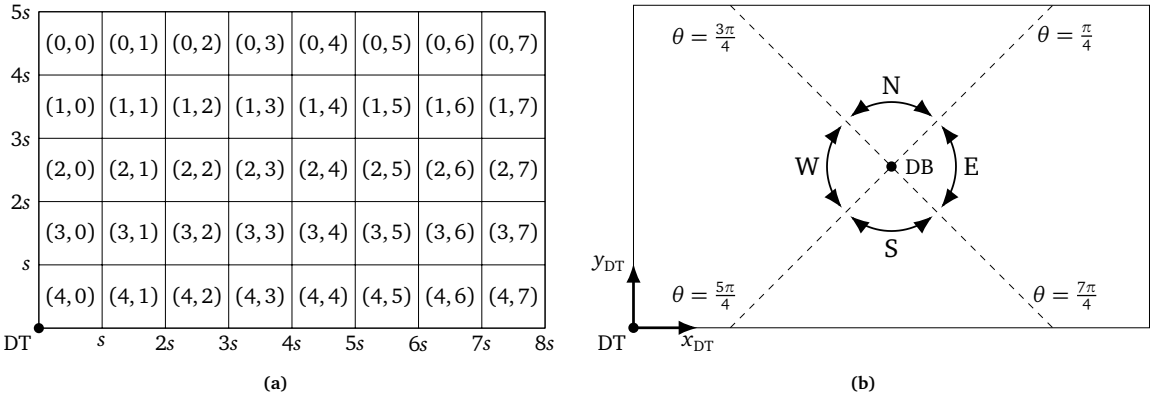


Figure 11.3: Grid approximations of (a) position and (b) direction.

11.3.3. Odometry Reset Procedure

As mentioned in Chapter 8, the accuracy of the dead-reckoning model decreases over time. To overcome this problem, the model must be reset using absolute position updates.

The following procedure, visualized in Figure 11.4, is followed in order to reset the odometry model:

1. When the robot detects a stop line, it applies a homography to determine the pose of the Duckiebot with respect to the reference frame of the detected stopline.
2. If the Duckiebot knows the pose of the intersection line with respect to the Duckietown reference frame, it can determine its own pose relative to the Duckietown reference frame.
3. The resulting pose is translated and rotated to the odometry reference frame.

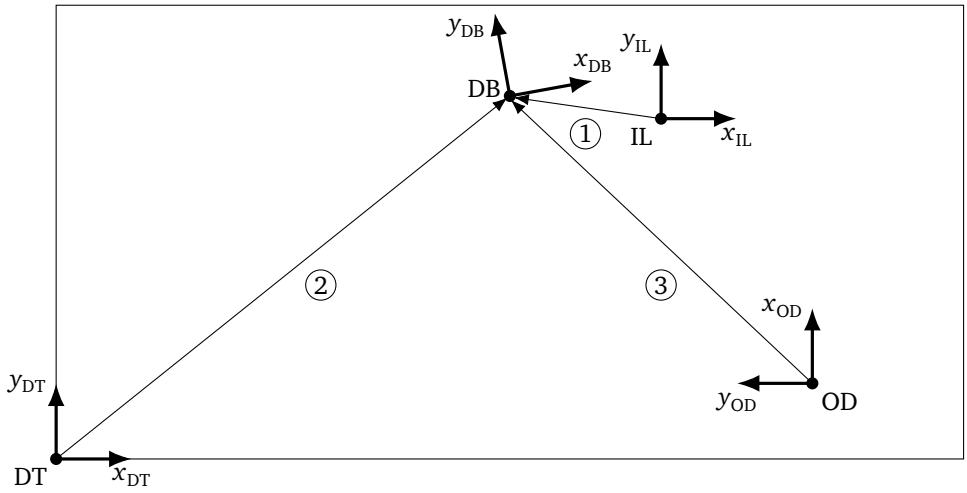


Figure 11.4: Relevant reference frames associated with the odometry reset procedure.

This procedure is performed using the transformation given by

$$T_{\text{DB}\to\text{OD}} = T_{\text{DT}\to\text{OD}}\,T_{\text{IL}\to\text{DT}}\,T_{\text{DB}\to\text{IL}}, \tag{11.6}$$
$$= (T_{\text{OD}\to\text{DT}})^{-1}\,T_{\text{DB}\to\text{DT}}, \tag{11.7}$$
$$= \begin{bmatrix} \cos(\theta_{\text{OD,DT}}-\theta_{\text{DB,DT}}) & \sin(\theta_{\text{OD,DT}}-\theta_{\text{DB,DT}}) & x_{\text{DB,OD}} \\ -\sin(\theta_{\text{OD,DT}}-\theta_{\text{DB,DT}}) & \cos(\theta_{\text{OD,DT}}-\theta_{\text{DB,DT}}) & y_{\text{DB,OD}} \\ 0 & 0 & 1 \end{bmatrix}, \tag{11.8}$$

where

$$x_{\text{DB,OD}} = [x_{\text{DB,DT}} - x_{\text{OD,DT}}]\cos(\theta_{\text{OD,DT}}) + [y_{\text{DB,DT}} - y_{\text{OD,DT}}]\sin(\theta_{\text{OD,DT}}), \tag{11.9}$$
$$y_{\text{DB,OD}} = [x_{\text{OD,DT}} - x_{\text{DB,DT}}]\sin(\theta_{\text{OD,DT}}) + [y_{\text{DB,DT}} - y_{\text{OD,DT}}]\cos(\theta_{\text{OD,DT}}). \tag{11.10}$$

Then the resulting pose becomes

$$\mathbf{q}_{\text{DB}\to\text{OD}} = \begin{bmatrix} [x_{\text{DB,DT}} - x_{\text{OD,DT}}]\cos(\theta_{\text{OD,DT}}) + [y_{\text{DB,DT}} - y_{\text{OD,DT}}]\sin(\theta_{\text{OD,DT}}) \\ [x_{\text{OD,DT}} - x_{\text{DB,DT}}]\sin(\theta_{\text{OD,DT}}) + [y_{\text{DB,DT}} - y_{\text{OD,DT}}]\cos(\theta_{\text{OD,DT}}) \\ \theta_{\text{OD,DT}} - \theta_{\text{DB,DT}} \end{bmatrix}. \tag{11.11}$$

11.4. State-of-Charge Based Behaviour

The Duckiebot uses a lithium-ion (Li-ion) battery, which is charged using the constant current constant voltage (CC-CV) method. In this method, charging takes place in two stages. In the first stage, the current of the battery cell is kept constant until the battery cell voltage reaches a threshold value. At stage 2, the battery is charged using a trickle current that is being applied by the constant output voltage of the charger. From the moment that stage 2 begins, the rate of charge decreases over time [27].

Now suppose that the robot has a mission — which consists of driving to a location, performing a task at that location, and driving back to the charging park — for which it does not require the battery to be completely charged. By letting the robots charge to a certain threshold rather than to maximum capacity, the efficiency of the charging method increases.

The navigational decisions that the Duckiebot has to make are based upon its State-of-Charge (SoC), of which the different modes of behaviour are shown in Figure 11.5.

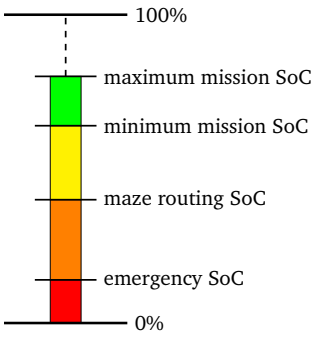


Figure 11.5: State of Charge behaviour distribution.

The Duckiebot acts differently depending on its SoC and the fact whether it is charging or not.

When the Duckiebot is charging:

- It wants to charge to at least the minimum mission SoC, but preferably to the maximum mission SoC.
- It is willing to leave the charging pad for other Duckiebots that are below their emergency SoC only if its current SoC is above the minimum mission SoC.

When the Duckiebot is not charging, it navigates to the charging park if its SoC is below the maze routing SoC. When it arrives at the charging park, their exist 3 possibilities:

1. In case that the entry traffic light of the charging park is off, it enters the charging park.
2. In case that the entry traffic light of the charging park is on, it will either
    - enter a waiting queue if its SoC is above the emergency SoC.
    - enter the charging park if its SoC is below the emergency SoC.

11.5. ROS Implementation of Odometry and Maze Routing

The navigation module combines the aspects of maze routing, odometry and computer vision in order to let the Duckiebot navigate towards a charging park when it wants to. This section describes the ROS implementation of the navigation module, which is made up of a module that handles the odometry aspect and a module that handles the maze routing aspect.

11.5.1. The Odometry Handling Module

The module that handles the odometry is shown in Figure 11.6. As stated before, the Duckiebot contains a package that applies the principle of odometry to determine the pose of the Duckiebot. This package has been slightly adjusted such that its pose variables can be updated.

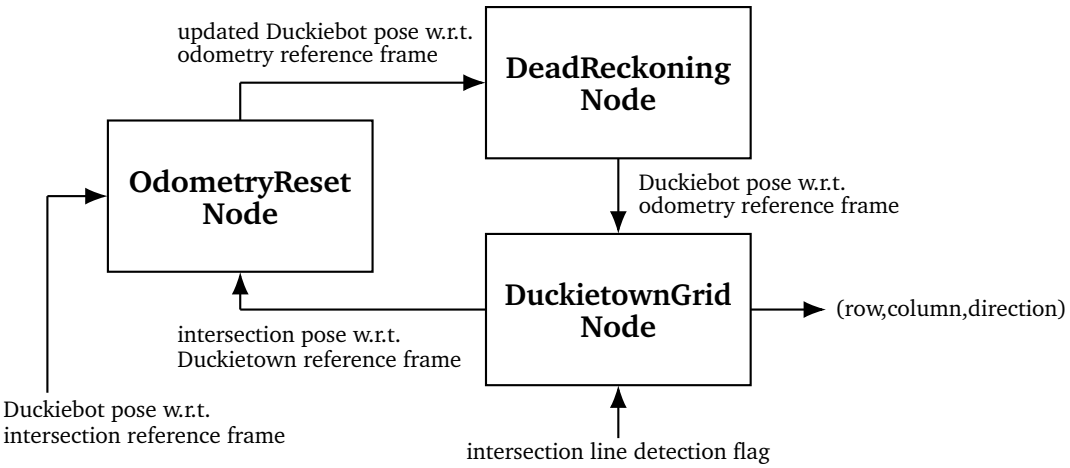


Figure 11.6: Odometry handling module.

Each time that the `DeadReckoning` node publishes an odometry message, the `DuckietownGrid` node produces a 2D pose with respect to the Duckietown reference frame. Using the characteristics of the Duckietown it can relate the pose to a certain tile within the Duckietown. The row and column of the specific tile and the NESW-direction of the Duckiebot are then published, which the maze routing module utilizes to determine the intersection instructions that are needed to navigate to the charging park.

When the Duckiebot detects an intersection line, the `DuckietownGrid` node publishes the known pose of the intersection line, which the `OdometryReset` node uses to determine the pose $\mathbf{q}_{\mathrm{DB}\to\mathrm{OD}}$. It then publishes its result to the `DeadReckoning` node, which effectively resets the odometry.

11.5.2. The Maze Routing Module

The maze routing module, shown in Figure 11.7, determines the instructions that the Duckiebot needs to follow to arrive at the charging park.

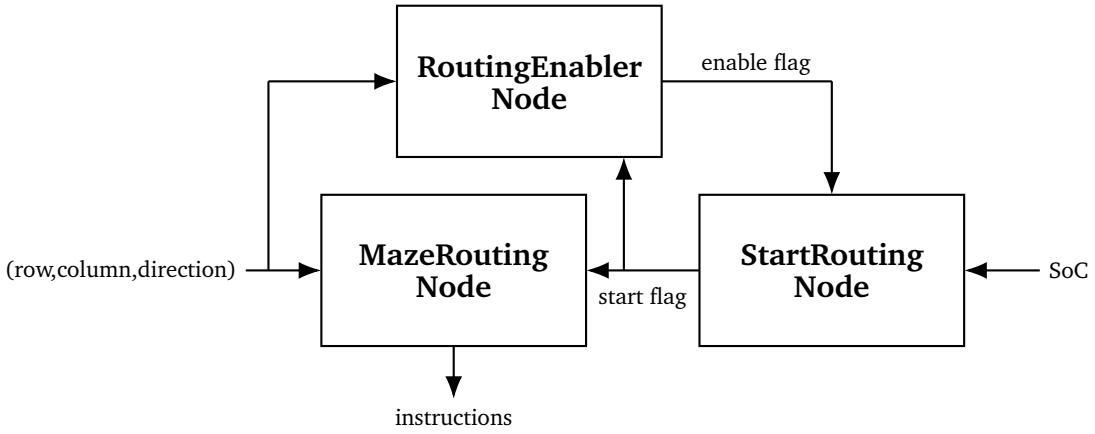


Figure 11.7: Maze routing module.

When the Duckiebot is below the predefined maze routing SoC, the `StartRouting` node sets its start flag to high. Then the `MazeRouting` node performs the Lee algorithm, in which the start cell is the current tile that the Duckiebot occupies and the target cell is the entry of the charging park. The `RoutingEnabler` node sets its enable flag to low, resulting in the `StartRouting` node being unable to update its start flag. This is done such that the Lee algorithm is only performed when the Duckiebot is not already following a predetermined path.

When a path is found, the `MazeRouting` node publishes the instructions that the Duckiebot needs to perform to arrive at the charging park. The `RoutingEnabler` node sets its enable flag to high when the Duckiebot drives out of the charging park, such that the maze routing start procedure can be initiated again when the Duckiebot's SoC is below the maze routing SoC.

12

Motion Control

12.1. Introduction

The motion control unit it responsible for processing the information the robot has and determining what the wheels should do. The motion control gets all the visual information, whether there is a intersection, coil lines, traffic lights and the lane error. It also gets information from the navigation part what it should do if it encounters a intersection. Lastly it also knows if the robot's SoC is at a critical state and the value of the ToF sensor. All this information is processed by the motion control and then publishes the angular and linear velocity.

12.2. Motion Control ROS node

The motion control node needs to process all it's different inputs. Firstly it is important that if the speed at which the robot drives depends on if the space in front of the robot is clear and no other objects are present. If the value of the ToF is below a certain distance the robot will drive slower so it will not hit a robot driving in front of it. Then the motion control has multiple modes depending on what the robot has detected and what instruction is has received from the navigation unit. An overview of the in- and output of the node can be seen in Figure 12.1

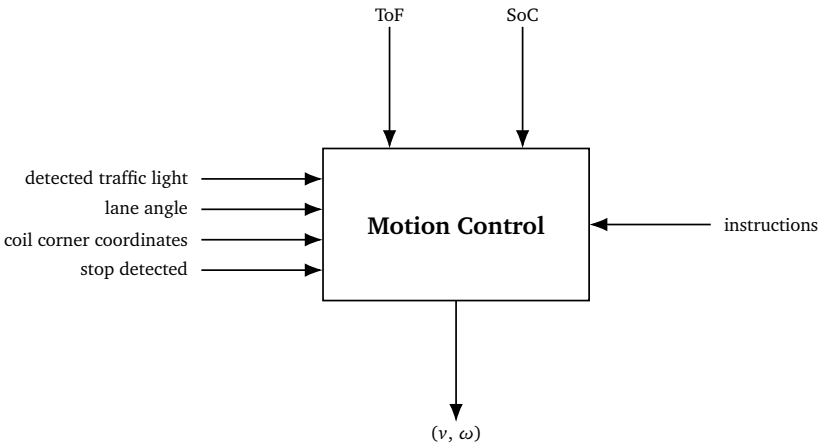


Figure 12.1: Overview of the inputs of the Motion Control ROS node

12.2.1. Lane Following Mode

The standard mode is the lane follower mode, in this mode will simply follow the road and adjust to stay in the centre of the lane. When the robot detects a traffic light it will update its next instruction. The next instruction tells the motion control what it should do at the next intersection.

12.2.2. Traffic Light Detection

If the robot detects a traffic light which indicates the entrance of a charging park it decides to go straight at the intersection if the light is off and the SoC is such that the robot wants to charge or if the light is on but the robot is below the emergency SoC. If these cases are not met the robot will turn left and enter the charging park queue in order to wait until a charging pad is clear.
If the robot detects a charging station traffic light it means it is in the charging park so if the light is on it will keep going straight since the charging pad is occupied. Otherwise the pad is free and it will turn right.
If the robot detects a normal traffic light and other traffic lights it will simply follow the instruction received by the navigation command. If the light is off it means no traffic is approaching and the robot can drive onto the intersection. When the light is on it means the robot needs to wait at the intersection line until the light turns off.

12.2.3. Intersection and Coil Alignment Detection

When a robot detects a intersection line it will see if it needs to wait for the traffic light or not. If it needs to wait for the traffic light it will go into wait mode, where it will stay until it detects that the traffic light has switched off. If this is not the case it will simply keep driving until a intersection line is no longer detected, this means the robot is now on the intersection. It will then go into the mode depending on the set instruction.

Lastly if the robot detect a coil alignment line it means it is in a charging station and needs to charge. Therefor it will then go into alignment mode and adjust it position until the required alignment is reached. This is done by applying homography to the points and comparing with reference points.

12.2.4. Motion Control Mode Overview

So the robot has six modes, the robot checks which mode it is in and publishes linear and angular velocity accordingly every time the lane error is updated. The way the robot transitions between modes can be seen in Figure 12.2.

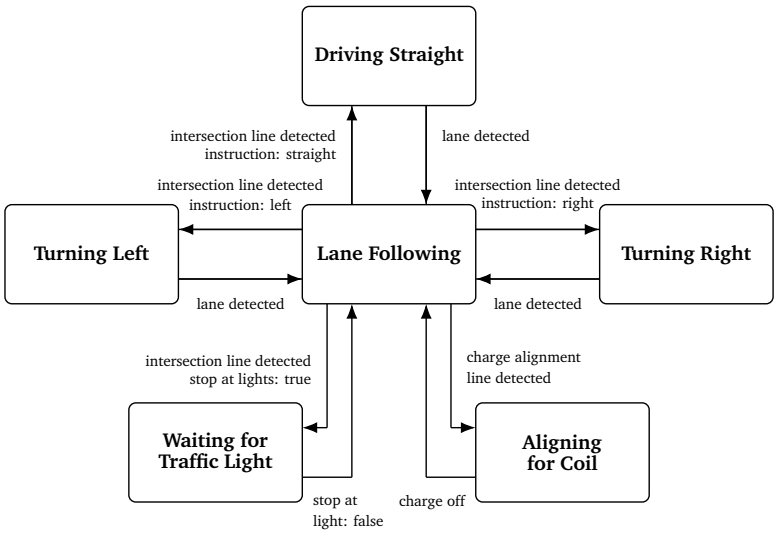


Figure 12.2: Overview of the different motion control modes and when the transition between modes happen

In each mode the angular and linear velocity the motion control publishes are determined differently. In the lane following mode it uses the lane angle and distance error in order to determine the velocities. In waiting for traffic light mode the velocities are zero. In the turning and driving straight mode it will continuously send out the same velocities until a lane is once again detected and it returns to lane following mode. Since all of the intersections are equal in size the radius of the turn is the same for each intersection. Lastly for the coil alignment the robot will drive according to the error between the found points and the reference points. If this error is within a certain range it will no longer drive since it is able to charge. Then it will wait until it is done charging after which it return to lane follower mode. Unless its SoC is above the minimum mission SoC and another Duckiebot wants to use the charging pad then it will return to the lane follower mode immediately.

Part III

Discussion and conclusion

13

Discussion

In this project different algorithms have been implemented in order to have the Duckiebot behave as according to the requirements. The algorithms that were used are thought to be accurate however in certain cases some algorithms may be inaccurate or fail entirely.

13.1. Discussion of the Path Finding Algorithm

Besides the potential issues with path finding algorithm described in Chapter 11, other problems could arise. Namely, in case of larger Duckietowns – think of Duckietowns whose grid consists out of hunders of rows and columns – the path finding algorithm may exceed the proposed maximum execution time. However, this will only affect the Duckiebot negatively if it is in very close proximity of an intersection the moment that the algorithm is initiated.

In case that the distance between intersections is large, the odometry becomes more inaccurate over time. This could potentially lead into the Duckiebot estimating its NESW-direction relative to the Duckietown incorrectly, resulting in the path finding algorithm producing an incorrect path, or is not even able to find a path.

13.2. Discussion of the Image filtering techniques

In the image the HSV values for colour change under different lighting conditions. In order to cover different lighting conditions the range for the HSV values needs to be larger then for one lightning condition. This could increase the change of potential error due to noise not being filtered out.

The line detection was found to be a method accurate for detecting lanes and detecting the centre of the path. It is only perceptible to errors on left side due to left line not being continuous. The gaps in the line could lead to errors if some points left of the line are in the masks.

However most importantly the system is very depended on the accuracy of homography. Homography is responsible for resetting the odometry and aligning the coil for example. They are essential for making sure the system functions. If the coil alignment is inaccurate the Duckiebot might charge very slow or not at all. Therefore the camera calibration needs to be able to very accurately determine the homography matrix.

13.3. Accuracy of the YOLO Object Detection

The YOLO neural network is very good at detecting and differentiating between different traffic lights with a confidence score around 0.9. However the model was not trained to detect whether the traffic lights were on or off. To detect this the model has to be retrained or the Duckiebot has to look at colour values in the centre of the bounding box. The first option depends on the ability of the newly trained model to distinguish between a traffic light which is on or off. The second option depends on the models accuracy to draw bounding boxes. The bounding box loss of the model was around 0.02 at 100 epochs as seen in Figure B.1. It is higher than the objectness and classification loss but still very low.

14

Conclusions and Recommendations

14.1. Conclusion

The goal of the project was to implement the robot behaviour of a wireless charging system to serve as a basis for further development. Different modules were made to implement the behaviour. The navigation part guides the robot to the charging station when its SoC is low. The image processing part extracts information from the camera images of the robot. Lastly the motion control drives the robot depending on the available information. These modules together full fill the requirements set beforehand.

The system is fully implemented using ROS so it can be used on other robots which also run ROS. The Duckiebot uses both YOLO object detection to know it has arrived at a charging park or charging station. It uses the camera images and homography to align itself with the charging pad. The image processing and object detection allow the robot to drive autonomously through the Duckietown without going outside of its lane. It avoid collision using the ToF sensor and intersection traffic lights. Additionally the Duckiebot only leaves a charging station if it is charged to level itself has deemed appropriate.

Currently the resulting processes of the input has not been tested on the Duckiebot therefore it can not confidently be said that the processing time of the inputs is below the frametime of the camera. Testing on a laptop however computing times were below the frame rate of the camera. Since the Jetson Nano located on the Duckiebot is designed to run neural networks and artificial intelligence algorithms it is believed that the Duckiebot's processing time would be below the frametime.

14.2. Recommendations

Since the project was meant as a basis the following potential improvements are recommended:

- The project should be performed with robots that can communicate bidirectional. This could lead to more of discussion between robots for priority of a charging pad.
- In stead of programming the structure of the Duckietown into the memory of the Duckiebot, it could map the structure on its own.
- The Duckiebot should have an ability to dynamically adjust its SoC threshold distribution based upon its current mission.
- Adding support for differentiating between multiple charging parks.
- Adding machine learning for modelling the time between encounters of charging station while performing a task in order to change its SoC threshold.
- Path finding can be changed so the Duckiebot can turn and travel in the opposite direction.
- Using machine learning to determine to which charging park the Duckiebot should navigate depending on the chance the charging park has an available charging pad.

Part IV

Bibliography

References

[1] J. Tani, L. Paull, M. T. Zuber, *et al.*, "Duckietown: An innovative way to teach autonomy", in *Educational Robotics in the Makers Era*, D. Alimisis, M. Moro, and E. Menegatti, Eds., Cham: Springer International Publishing, 2017, pp. 104–121, ISBN: 978-3-319-55553-9.

[2] N. du Plessis and M. Roos, "Autonomous wireless charging system for robot swarms: Wireless charging hardware", Bachelor's Thesis, Delft University of Technology, Jun. 2023.

[3] M. Ayoub and M. Versluis, "Autonomous wireless charging system for robot swarms: Charging park design", Bachelor's Thesis, Delft University of Technology, Jun. 2023.

[4] *Duckiebot founder's edition datasheet*, version 5, Duckietown, 2018.

[5] *Adafruit vl53l0x time of flight microlidar distance sensor breakout*, Adafruit Industries, Dec. 2022, pp. 6–7.

[6] J. Bloomenthal and J. Rokne, "Homogeneous coordinates", *The Visual Computer*, vol. 11, pp. 15–26, 1994.

[7] B. Poling, "A tutorial on camera models", *University of Minnesota*, pp. 1–10, 2015.

[8] H. A. Hashim, L. J. Brown, and K. McIsaac, "Nonlinear pose filters on the special euclidean group se(3) with guaranteed transient and steady-state performance", *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 5, pp. 2949–2962, 2021. DOI: 10.1109/TSMC.2019.2920114.

[9] J. Ćesić, I. Marković, I. Cvišić, and I. Petrović, "Radar and stereo vision fusion for multitarget tracking on the special euclidean group", *Robotics and Autonomous Systems*, vol. 83, pp. 338–348, 2016.

[10] V. Sonneville, A. Cardona, and O. Brüls, "Geometrically exact beam finite element formulated on the special euclidean group se (3)", *Computer Methods in Applied Mechanics and Engineering*, vol. 268, pp. 451–474, 2014.

[11] S. S. Beauchemin and R. Bajcsy, "Modelling and removing radial and tangential distortions in spherical lenses", in *Multi-Image Analysis: 10th International Workshop on Theoretical Foundations of Computer Vision Dagstuhl Castle, Germany, March 12–17, 2000 Revised Papers*, Springer, 2001, pp. 1–21.

[12] G. M. H. Amer and A. M. Abushaala, "Edge detection methods", in *2015 2nd World Symposium on Web Applications and Networking (WSWAN)*, 2015, pp. 1–7. DOI: 10.1109/WSWAN.2015.7210349.

[13] M. Juneja and P. Sandhu, "Performance evaluation of edge detection techniques for images in spatial domain", *International Journal*, vol. 1, pp. 614–621, Jan. 2009. DOI: 10.7763/IJCTE.2009.V1.100.

[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You only look once: Unified, real-time object detection*, 2016. arXiv: 1506.02640 [cs.CV].

[15] J.-a. Kim, J.-Y. Sung, and S.-h. Park, "Comparison of faster-rcnn, yolo, and ssd for real-time vehicle type recognition", in *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, 2020, pp. 1–4. DOI: 10.1109/ICCE-Asia49877.2020.9277040.

[16] N. Pereira, "Pereiraaslnet: Asl letter recognition with yolox taking mean average precision and inference time considerations", in *2022 2nd International Conference on Artificial Intelligence and Signal Processing (AISP)*, 2022, p. 3. DOI: 10.1109/AISP53593.2022.9760665.

[17] S. Permana, K. Bintoro, B. Arifitama, and A. Syahputra, "Comparative analysis of pathfinding algorithms a *, dijkstra, and bfs on maze runner game", *IJISTECH (International Journal Of Information System & Technology)*, vol. 1, p. 1, May 2018. DOI: 10.30645/ijistech.v1i2.7.

[18] C. Y. Lee, "An algorithm for path connections and its applications", *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961. DOI: 10.1109/TEC.1961.5219222.

[19] F. Rubin, "The lee path connection algorithm", *IEEE Transactions on Computers*, vol. C-23, no. 9, pp. 907–914, 1974. DOI: 10.1109/T-C.1974.224054.

[20] I. Bostan, I. Lita, E. Franti, M. Dascalu, C. Moldovan, and S. Goschin, "Systematic odometry errors compensation for mobile robot positioning", in *The Experience of Designing and Application of CAD Systems in Microelectronics, 2003. CADSM 2003. Proceedings of the 7th International Conference.*, 2003, pp. 574–576. DOI: 10.1109/CADSM.2003.1255162.

[21] J. Palacín and D. Martínez, "Improving the angular velocity measured with a low-cost magnetic rotary encoder attached to a brushed dc motor by compensating magnet and hall-effect sensor misalignments", *sensors*, vol. 21, no. 14, p. 4763, 2021.

[22] E. Ivanjko, I. Komsic, and I. Petrovic, "Simple off-line odometry calibration of differential drive mobile robots", in *Proceedings of 16th Int. Workshop on Robotics in Alpe-Adria-Danube Region-RAAD*, 2007, pp. 164–169.

[23] J. Borenstein and L. Feng, "Measurement and correction of systematic odometry errors in mobile robots", *IEEE Transactions on Robotics and Automation*, vol. 12, no. 6, pp. 869–880, 1996. DOI: 10.1109/70.544770.

[24] R. B. Sousa, M. R. Petry, and A. P. Moreira, "Evolution of odometry calibration methods for ground mobile robots", in *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2020, pp. 294–299. DOI: 10.1109/ICARSC49921.2020.9096154.

[25] R. B. Sousa, M. R. Petry, P. G. Costa, and A. P. Moreira, "Optiodom: A generic approach for odometry calibration of wheeled mobile robots", *Journal of Intelligent & Robotic Systems*, vol. 105, no. 2, p. 39, 2022.

[26] *The duckiebot manual, calibration - wheels*, Duckietown, 2022.

[27] M. Kisacikoglu, B. Ozpineci, and L. Tolbert, "Reactive power operation analysis of a single-phase ev/phev bidirectional battery charger", Jul. 2011, pp. 585–592. DOI: 10.1109/ICPE.2011.5944614.

Part V

Appendix

A

Overview of subsystems of the ROS implementation

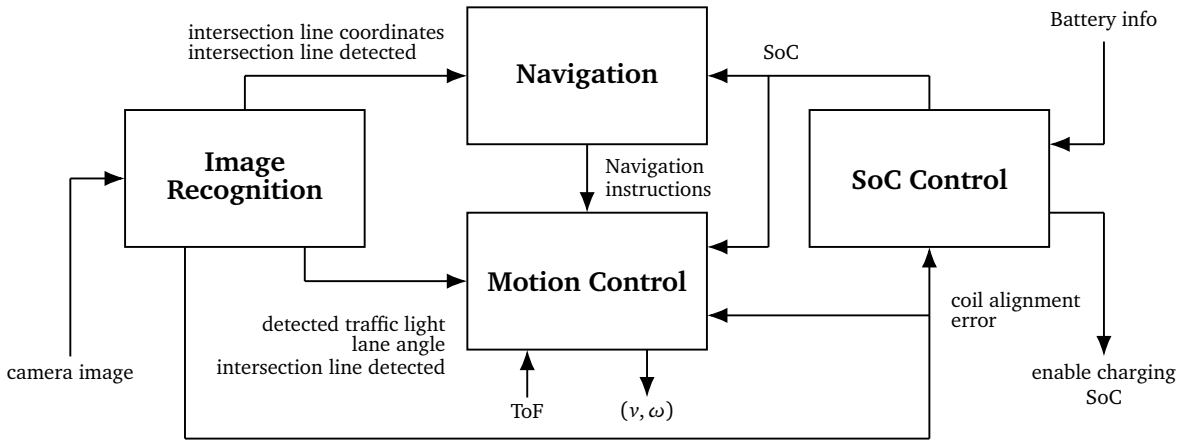


Figure A.1: Overview of subsystems of the ROS implementation

B

Graphics of the Trained Model

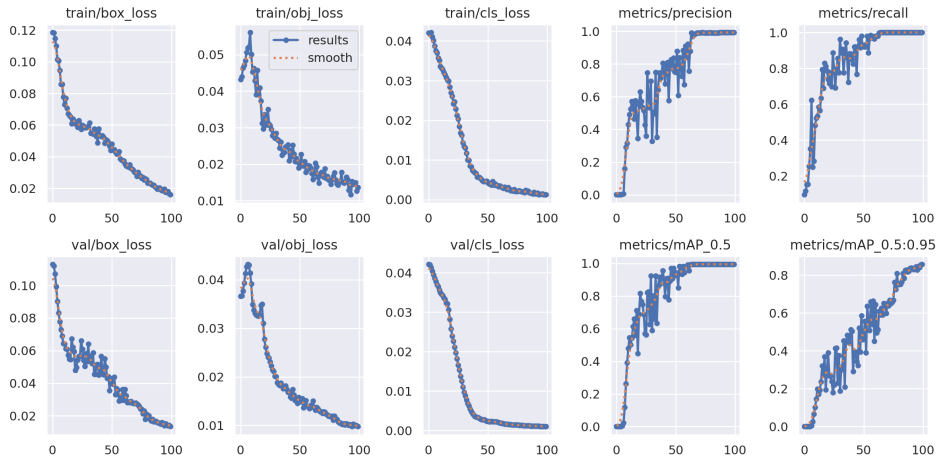


Figure B.1: Training and validation losses, precision and recall and mAP graphs of the model for 100 epochs.

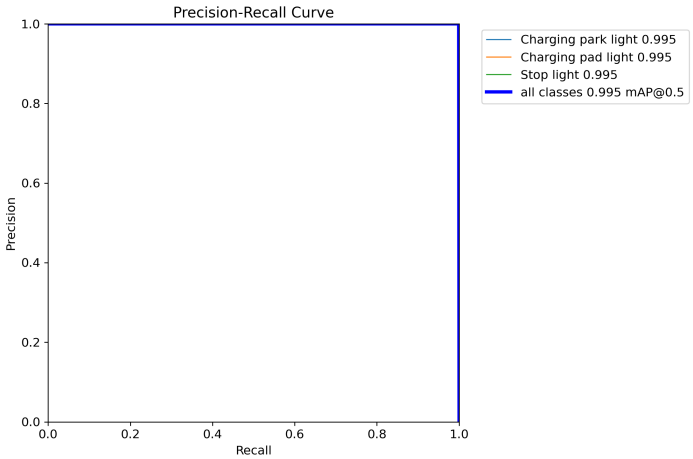


Figure B.2: PR curve for 100 epochs.

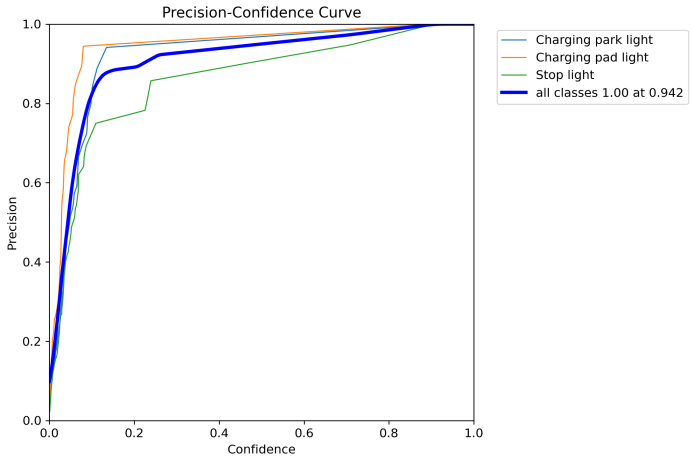


Figure B.3: P curve for 100 epochs.

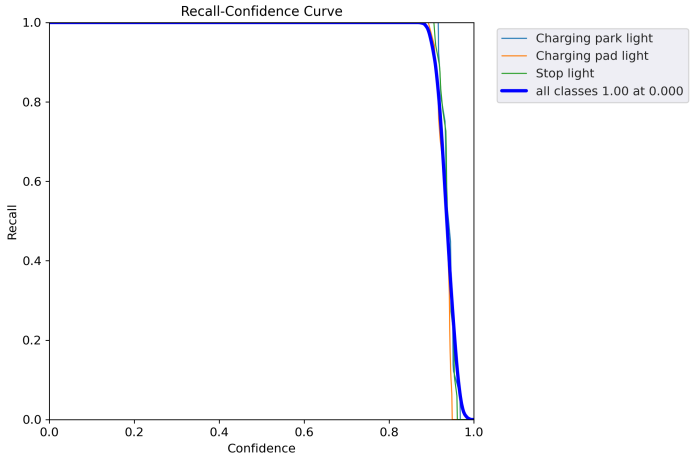


Figure B.4: R curve for 100 epochs.

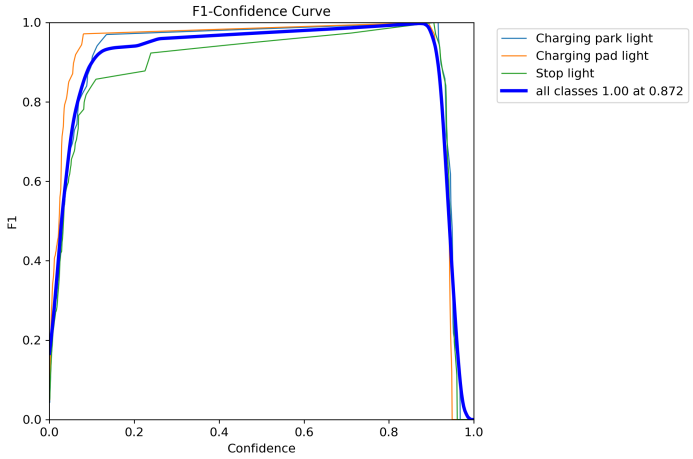


Figure B.5: F1 curve for 100 epochs.

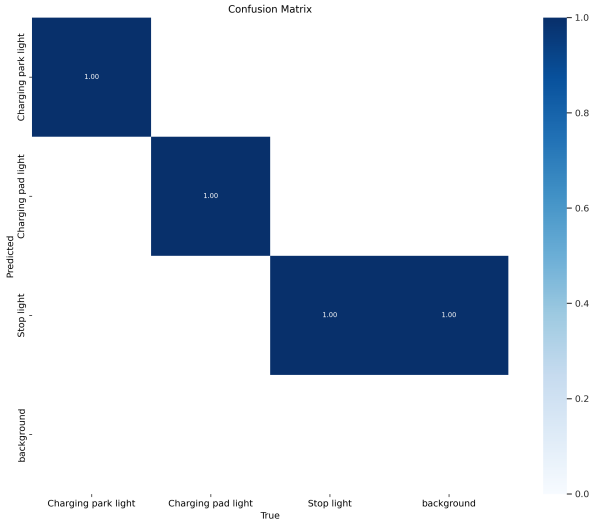


Figure B.6: Confusion matrix for 100 epochs.