České vysoké učení technické v Praze
Fakulta dopravní

Katedra aplikované matematiky
Obor: Inteligentní dopravní systémy

Kamerově ovládané autonomní robotické vozítko

Camera controlled autonomous robotic vehicle

BAKALÁŘSKÁ PRÁCE

Vypracoval:	Adam Fialka
Vedoucí práce:	Ing. Bohumil Kovář Ph.D.
Rok:	2022

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
děkan
Konviktská 20, 110 00 Praha 1

K611.. **Ústav aplikované matematiky**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Adam Fialka

Studijní program (obor/specializace) studenta:

bakalářský – ITS – Inteligentní dopravní systémy

Název tématu (česky): **Kamerově ovládané autonomní robotické vozítko.**

Název tématu (anglicky): Camera-controlled autonomous robotic vehicle.

Zásady pro vypracování

Při zpracování bakalářské práce se řiďte následujícími pokyny:

- V teoretické části zpracujte přehled metod používaných pro ovládání a navigaci autonomních robotických vozítek. Zaměřte se zejména na metody, které používají zpracování obraz z kamery. Vysvětlete ty nejpoužívanější a to včetně matematického popisu.
- V experimentální části (a), popište konstrukci testovacího vozíku a konfiguraci použitého počítače. Popište a experimentálně ověřte softwarové ovládání pohybu vozítka a snímání obrazu z kamery, (b) pro vozítko naprogramujte aplikaci, pomocí které vozítko autonomně projede testovací trať.

ČVUT FD

Rozsah grafických prací:	není specifikovaný
Rozsah průvodní zprávy:	minimálně 35 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)
Seznam odborné literatury:	dle doporučení vedoucího bakalářské práce

Vedoucí bakalářské práce: **Ing. Bohumil Kovář, Ph.D.**

Datum zadání bakalářské práce: **8. října 2021**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání bakalářské práce: **8. srpna 2022**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

[illegible]

L.S.

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
-3-

[illegible]

prof. Ing. Ondřej Přibyl, Ph.D.	**doc. Ing. Pavel Hrubeš, Ph.D.**
vedoucí Ústavu aplikované matematiky	děkan fakulty

Potvrzuji převzetí zadání bakalářské práce.

[illegible]

Adam Fialka
jméno a podpis studenta

V Praze dne .... 8. října 2021

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci, zpracovanou na závěr bakalářského studia na ČVUT v Praze, Fakultě dopravní.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškloských závěrečných prací.

Nemám závažný důvod proti užívání tohoto školního díla ve smyslu § 60 zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon)

[illegible]
....
Adam Fialka

Dne: 30. 11. 2022

Poděkování
Tímto bych chtěl poděkovat panu Ing. Bohumilu Kovářovi Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování bakalářské práce věnoval.

Název práce:	**Kamerově ovládané autonomní robotické vozítko**
Autor:	Adam Fialka
Studijní program:	Technika a technologie v dopravě a spojích
Obor:	Inteligentní dopravní systémy
Druh práce:	Bakalářská
Vedoucí práce:	Ing. Bohumil Kovář Ph.D.

Abstrakt: Tato bakalářská práce se zabývá tvorbou a programováním algoritmu pro detekci jízdního pruhu za pomoci záznamu z kamery, který může být použit pro řízení robotického vozítka, osazeného kamerou a jednočipovým počítačem. Cílem práce je otestování vyvinutého algoritmu na vozítku osazeném počítačem Nvidia Jetson Nano. V souladu s cílem práce bude do textu průběžně zařazován přehled a vysvětlení metod používaných pro navigaci autonomního vozítka, jež využívají zpracovaní obrazu z kamery. Pro naprogramování algoritmu byl zvolen jazyk Python, jelikož disponuje podporou knihovny OpenCV, jenž poskytuje množství předdefinovaných funkcí a metod pro zpracování obrazu. Při předzpracování obrazu bylo využito perspektivních transformací, které jsou v této problematice často používány pro usnadnění detekce. Byla využita známá binarizační metoda Otsu, jejímž výstupem je binární obraz, ze kterého detekuje hranice jízdního pruhu metoda zvaná Sliding window.
Výsledkem je kód pro algoritmus, který může být v budoucnu použit jako základ pro vývoj algoritmu na ovládání řídící nápravy vozítka. Vzhledem k okolnostem pandemie byla oblast testování algoritmu zúžena pouze na testovací videa z autokamery. Zároveň byla ověřena a vyhodnocena použitelnost některých konceptů pro detekci jízdního pruhu.

Title: **Camera-controlled autonomous robotic vehicle**

Abstract: This thesis deals with development and programming of a lane detection algorithm which, based on camera input, can be used for steering a robotic vehicle equipped with single–board computer and camera. The goal is to test the algorithm developed, using a vehicle equipped with an Nvidia Jetson Nano computer. Throughout the thesis, an overview of camera–based navigation methods will be included to correspond with the goal.

Python is the programming language of choice, thanks to the support of OpenCV library, which contains many predefined functions and methods used in image processing. The algorithm will utilize perspective transformations, which are often used in lane detection algorithms. The Otsu method was the binarization method of choice, which outputs a binary image. This binary image is then processed by the lane detection method called the sliding window. The output of the thesis is a source code for the algorithm, which can be used in future development of an algorithm for vehicle steering. Given the circumstances during the pandemic, the testing was reduced to sample test videos from a dash cam. Additionally, the usability of some lane detection concepts was proved and evaluated.

Contents

List of Figures

Listings

Introduction

As the subject of this thesis, I have chosen *Camera-controlled autonomous robotic vehicle*, because it opens the door to a world of computer vision, which is an exciting field, currently utilized in many areas of industry. The goal of this thesis is to develop an algorithm for lane detection, which could be used in the development of more complex autonomous vehicle software for a small toy car equipped with a camera and a single-board computer.

To accomplish this task, I will have to understand key principles that are the backbone of this entire field. Principles such as convolution, some topics of discrete mathematics, statistics, and many more. This knowledge base will then provide a sufficient foundation for understanding and, therefore, describing methods, typically utilized in solutions within the scope of the matter. This includes basic principles and methods used for image segmentation, color spaces, or lane detection algorithms.

As for the practical part, the developed algorithm will be put in Python code, which could be used later for testing on the vehicle. The code will be tested on a set of sample videos to prove its functionality. When the results of this testing will be satisfactory, testing on the toy car can begin. As a result, an edge detection algorithm will be developed. It should be possible to use this algorithm as a base for automating the steering of the toy car so that it can follow a simple track.

Motivation

As stated at the beginning, computer vision is a scientific field, which finds use in many areas of today's industry. From miracles of modern medicine, over trend setting in automated agriculture, to defining future of automotive industry, where autonomous driving technologies are one of the most discussed topics, with industry giants like BMW, Daimler, MobilEye or Tesla investing large resources into developing an autonomous vehicle systems.

Another popular field, closely related to computer vision and its use cases, is artificial intelligence and machine learning. It is safe to say that without combination of both, the probability of developing an autonomous vehicle would be low, maybe none.

Computer vision and relative areas are heavily used in ADAS (Advanced Driver Assistant Systems), which are supposed to prevent death and injuries by reducing the number of incidents. Key ADAS applications are, for example: Pedestrian detection, Lane departure warning, Automatic emergency breaking and more. It has to be said that these features are present in the vast majority of new cars and most of them utilize computer vision.

Earlier this year (2022), Mercedes-Benz unveiled the Level 3 Drive Pilot, which is the first-level 3 autonomous driving system with international valid certification. This means that the vehicle is capable of fully autonomous driving up to a speed of 60 kph in a highway environment. The same was achieved earlier by Honda, with their new Legend being also a level 3 autonomous vehicle; however, at the time of this thesis, it is not available outside of Japan. This leap in technology will surely result in further development.

List of Contributions

This thesis contributed to the project with the following:

- The use of Otsu binarization and sliding window methods was verified, showing strong positive results if certain conditions are met
- The source code, which is the output of this thesis, will be used by future students, providing a solid base for improvements and innovation
- future improvements were proposed.

Structure of the Thesis

In the beginning, a theoretical background (1) will be covered. Beginning with the introduction of color spaces, like RBG, CMYK, and others, which are heavily utilized in computer vision, continuing with the presentation of crucial mathematical concepts that make this thesis possible. These concepts include discrete derivatives, gradients, and more. Next, we will move to the introduction of traditional edge-detecting methods, such as the Sobel operator or Canny edge detector. All of those utilize mathematical knowledge, which was introduced prior. Moving on to concepts of thresholding and perspective transformation. Both will play a defining role in the forthcoming practical section. Lastly, we will introduce the current state of the art in the field of edge detection (2).

Everything has to start somewhere, and this part of the thesis will begin with the description of the first attempts at building the algorithm (3).I will show the concepts on which I tried to build the algorithm and which eventually failed, but the gained experience helped me on the way to the final version, which will be closely introduced with code samples later. In the following chapter (4), we will share the build of the algorithm and the vehicle itself. Here, I will also include the experience when I was assembling the vehicle, as well as provide some details of the setup. . Comming closer to the final conclusion, the testing of the algorithm will be covered (5), with possible improvements to the algorithm introduced and briefly described. And finally, the thesis will be concluded with an evaluation of the results.

Chapter 1

Background

In this chapter, I will introduce basic principles often used in computer vision. This includes color spaces, mathematical operators, or an overview of common edge detectors just to name a few. Understanding these principles is essential for the future development of more complex solutions.

1.1 Color Spaces

A color space is an organized set of colors. With a weighted combination of the principal color wavelength, it is possible to create almost any color [1]. These colors also form a coordinate system for color measurements. In addition, different color spaces favor different purposes. For example, the convenient RGB color space is valuable in programming [2], however, the resulting color does not always match the color the human eye perceives.

Many challenges of computer vision encounter difficulties with object detection under different lighting conditions. This is the area where color spaces become useful.

1.1.1 Turning Gray

For some applications, a grayscale image is needed. An achromatic image loses color information, and the gray level (intensity) becomes the only attribute [3]. This loss of information can be beneficial in further processing, where color does not play a significant role, e.g., edge detecting, smoothing, and so on.

Often, the user can find a need to convert a color image in RGB color space to a grayscale for further processing. It is done by using a weighted sum of color components, where it is necessary to take into consideration that human vision perceives each of the basic RGB colors differently. Therefore, if the weights are not used, the resulting image will appear dark in the ”green” and ”red” areas and bright in the ”blue” ones [2]. That is how we get the following formula for RGB to grayscale conversion.

$$Y = Luminance(R, G, B) = w_r \cdot R + w_g \cdot G + w_b B \tag{1.1}$$

Some specific values of each weight applied to equation (1.1) were developed, for example, to encode analog and digital color.

1.1.2 RGB

One of the most widely used linear color spaces is the RGB color space [4]. As the name suggests, the primary colors of this color space are Red, Green, and Blue. The space is represented by a normalized cube, a unit cube, where, at the origin, the black is located. The farthest point away from the origin represents white and on the connector of these two points is where the gray scale lies [3]. The number of bits that represent a pixel is called pixel depth.

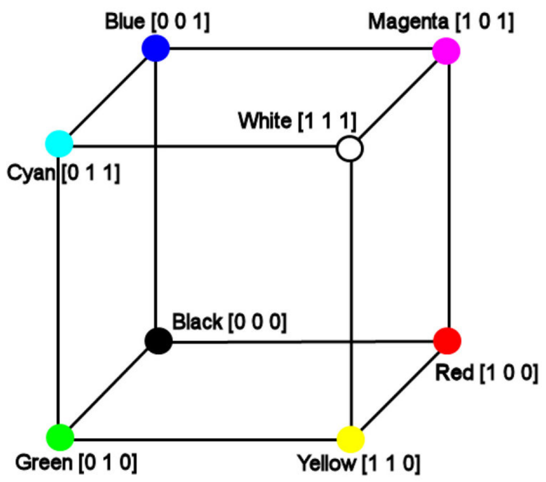


Figure 1.1: Weighted RGB cube[1]

1.1.3 CMY & CMYK

The CMY name comes from the three subtractive secondary colors, cyan, magenta, and yellow. It is a subtractive color model, which means that these primaries should be viewed as the colors that are subtracted from the white light. Secondary colors mean that it is possible to create them using primary colors: cyan is B + G (blue and green); magenta is R + B (red and blue), and yellow is R + G (red and green), as visible in figure 1.1 above [3].

[1] https://www.researchgate.net/figure/The-RGB-colour-cube_fig1_304240592

Because mixing colors causes poor black, most printing devices, for example, use the fourth ink - black, and so we have to introduce the CMYK color space, where "K" represents black. This color model also finds use in printing, where black(or lack of it) is noticeable and affects the overall output quality.

The color coordinates can be, due to their close relation, converted from RGB to CMY in a simple way:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 255 \\ 255 \\ 255 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{1.2}$$

1.1.4 HSV

In the real world, our sight does not follow these principles. Instead, the human eye can detect the color's brightness, saturation, and hue, and this is exactly what the HSV color space (hue, saturation, value) attempts to represent. Note that the HSV color space is sometimes referred to as an HSB, where B stands for brightness, but technically it is the same color space.

This color space is often presented as a cone [5], where the black is located at the tip. From here up, the height of the cone represents the value, the amount of light reflected by the color, so logically, at the other end, is where the white color is located. Changes in Hue are represented by the angle in the range $\langle 0, 360 \rangle$. Finally, the horizontal axis is a representation of saturation.

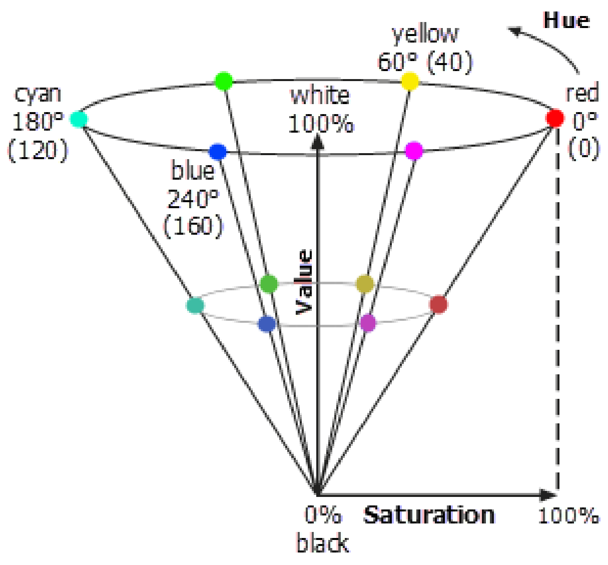


Figure 1.2: HSV cone[2]

[2]https://desktop.arcgis.com/en/arcmap/10.7/manage-data/raster-and-images/color-model-conversion-function.html

1.1.5 Lab

This color space (also known as CIELAB or L*A*B) is a 3-dimensional system that consists of lightness (L), and color dimensions (a,b). One of the key properties of this color space is its independence and exactness of the device [6]. These make the Lab color model suitable for carrying color information from device to device, i.e., use in many different industries.

Although the color representation is the most accurate, it is often converted to less accurate color spaces. The reason behind this is that computer screens, printers, and other devices use three or even four colors for image representation.

1.1.6 YCrCb

The YCrCb color space is derived from the RGB model, which was mentioned before. It is standardized for digital television and image compression [2] and is used for these purposes due to its lower bandwidth requirements. Because in the RGB model, each color component has to have equal bandwidth [7], YCrCb color model separates the brightness component Y (Luma component), Cr and Cb (chroma red and chroma blue). It was shown that the human eye is more sensitive to changes in brightness than to color, which is the reason behind this separation: It is possible to send Cr and Cb at a lower rate than Y, resulting in bandwidth and cost savings [7].

YCrCb can be represented as RGB as a cube. Unlike the RBG cube, where white and black are located in opposite corners (can be seen above (1.1)) making up a ”grayscale diagonal”, here white and black are located in the center of the cube’s sides.

In some cases, an interactive example is more useful than a whole book of theory. For these purposes, I can recommend a well-created website [5], which was created as a practical exercise of a bachelor’s thesis at Paderborn University. This website provides, in my opinion, excellent interactive representations of color spaces with a brief introduction.

1.2 Computer Vision and Digital Image Processing

1.2.1 Derivatives

The concept of a derivative is fundamental in image processing and is applied in many of its areas. Anyone, who participated in any calculus course should be familiar with the following notation:

$$\frac{\partial}{\partial x} f(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{1.3}$$

This is a definition of a first–order derivative. In computer vision, however, we are working with a discrete image. Therefore, it is necessary to reformulate this definition for discrete functions. To be able to apply the derivative to a discrete function, the concept of finite difference has to be introduced. Finite difference is, very simply, a distance between 2 points. It is possible to use the concept of finite difference to approximate the derivatives [3]. In the case of a digital function, Δx is the distance measured in pixels and defines the distance between samples of an image function. Because of all of this, we can reintroduce the first-order derivative in the following notation.

$$\frac{\partial f(x)}{\partial x} \approx f(x+1) - f(x) \tag{1.4}$$

The second–order derivative can be approximated in a similar way. It is possible to think of the second–order derivative as a 'difference of a difference', so at least 3 pixels are required [8].

$$\frac{\partial^2 f(x)}{\partial x^2} \approx f(x-1) + f(x+1) - 2f(x) \tag{1.5}$$

$$\frac{\partial^2 f(x)}{\partial y^2} \approx f(y-1) + f(y+1) - 2f(y) \tag{1.6}$$

Because finite differences have a strong response to sharp changes in pixel intensity, and therefore noise can destroy the output, we will need a smoothing tool of some sort to reduce its effect on the result (more on smoothing, etc. later).

1.2.2 Convolution

Firstly, it is necessary to introduce a couple of basic principles. The process of applying a pattern of weights for a linear filter (kernel) is usually called convolution [3]. A convolution is one of the fundamental operations in image processing [1]. In simple words, performing a convolution of two functions means rotating one of these functions by 180 degrees, taking the function at its origin and sliding it past the other function [3]. Then, at each sliding displacement, a computation is performed or, as Ballard said [1], the value of the convolution at any displacement is the integral of the product of the values of the functions (relatively displaced). Therefore, the formal notation of convolution is as follows:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \tag{1.7}$$

Convolution also , just like its less sophisticated brother multiplication, has some algebraic properties, such as commutativity, associativity, distributivity, etc.. (The notation below is in this order).

$$
\begin{aligned}
(f * g) &= (g * f) && (1.8)\\
(f * g) * h &= (h * g) * f && (1.9)\\
h * (g + f) &= (h * g) + (h * f) && (1.10)
\end{aligned}
$$

However, since the signal on which the convolution is performed is not a continuous signal but a discrete signal, it is necessary to take this fact into account and apply a discrete convolution:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{1.11}$$

The mechanics of convolution are applied in many areas of computer vision, for example, smoothing, and some edge-detecting techniques are based on kernel convolution, a simple convolution filter, which will be discussed later in this thesis.

1.2.3 Gradient

The gradient is a vector that points to the maximum rate of change of a function f at a single point. After computing the values for all points (x, y), $\nabla f(x, y)$ becomes a vector image, where each element is a vector calculated using the equation below [3].

$$\nabla f(x, y) = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] \tag{1.12}$$

Most edge detection methods start with estimating the magnitude of an image gradient [4]. We can compute the magnitude of the gradient with the Euclidean vector norm to determine the strength of the edge. In real-world applications, it is common to apply a threshold, so only edges that exceed the chosen value are taken into further processing [4].

$$M(x, y) = ||\nabla f(x, y)|| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{1.13}$$

We have obtained the magnitude, but we do not know in which direction the rate of change is the highest. The angles are measured counterclockwise, with respect to the x-axis [3].

$$\alpha(x, y) = \arctan\left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}}\right) \tag{1.14}$$

To opt for the image gradient, it is necessary to compute partial derivatives $(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$ on every pixel of an image. This is implemented in Gradient operators (Prewitt, Sobel, etc.), on which I will touch later, in other chapters.

1.2.4 The Gaussian

Sometimes, there is a need for fewer details in an image, so the resulting image (after edge detection, for example) is not too busy and contains only the important information. The most commonly used process is smoothing.

The Gaussian smoothing operator is a circularly symmetric convolution kernel [3]. Each pixel of the input array (the image) is convolved with the Gaussian kernel to create the output array (blurred image). When that happens, the output image loses details and noise compared to the input image. Although not the fastest filter, Gaussian smoothing is probably the most useful. [9].

The kernel is essentially a representation of a 2D Gaussian distribution, where σ stands for the standard deviation. The σ and the mean completely define the Gaussian [3], however, the mean of the kernels is zero.

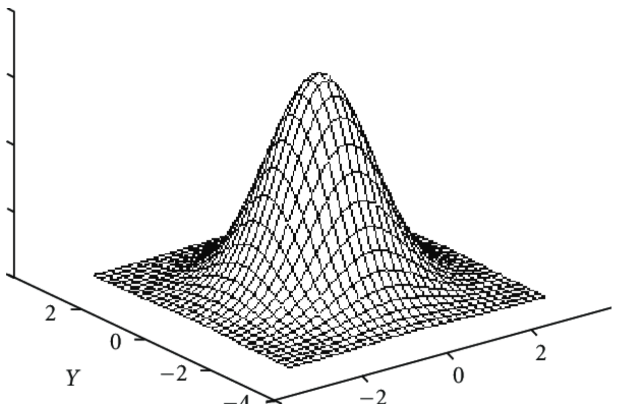


Figure 1.3: 2D Gaussian distribution[3]

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-x^2-y^2}{2\sigma^2}} \tag{1.15}$$

Now, just like in the case of the gradient, the resulting kernel, which is applied in computer vision, is an approximation. Otherwise, the kernel would be infinitely large. Theoretically, the values of the distribution are small enough that they can be ignored in the area past 3 standard deviations. This essentially means that using a kernel larger than 6σ x 6σ does not bring any benefits [3]. Thus, the dimension of a kernel is the smallest integer that satisfies this condition.

The two properties of Gaussian are that the product and convolution of two Gaussians are Gaussians as well. Because of that, the 2D Gaussian is separable into vertical and horizontal components. Then, the convolution is performed separately in the x and y directions.

[3] https://www.researchgate.net/figure/2D-Gaussian-distribution-with-mean-0-0-and-s-1-Source-http-wwwceehwacuk_fig1_26487220

1.2.5 The Laplacian

The Laplacian is a sum of second-order derivatives of a function. Because derivatives of any order are linear operations, the Laplacian is also a linear operator [3].

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{1.16}$$

However, in image processing, the Laplacian is used as a filter which, when applied to an image, can be used for edge detection [8]. And because the image is a discrete function, as in the case of convolution, a discrete version of the Laplacian operator has to be introduced [10]. To do that, we add the notation of discrete second-order derivatives mentioned earlier in (1.5).

$$\nabla^2 f = f(x-1) + f(x+1) - 2f(x) + f(y-1) + f(y+1) - 2f(y) \tag{1.17}$$

When applied, edges are represented with strong zero crossings in the Laplacian of an image [8]. There are two things that we must keep in mind: the Laplacian does not provide any information on the orientation of an edge and the sensitivity of the finite difference to noise [8].

1.2.6 The Laplacian of Gaussian (LoG)

As mentioned above, using a Laplacian for edge detection can be tricky due to its sensitivity to noise. Due to this fact, it is necessary to smooth the image first, convolving the image function with Gaussian, for example, and then applying the Laplacian. Because both the Laplacian and Gaussian operators are linear operators, it is possible to switch the order of the operations to get the resulting operator named the Laplacian of Gaussian. [11].

$$\nabla^2(G(x,y,\sigma) * f(x,y)) = (\nabla^2 G(x,y,\sigma) * f(x,y)) \tag{1.18}$$

The $\nabla^2 G(x,y)$ operator can be effectively approximated by the difference of two images convolved by two Gaussians with different variances σ [11].

1.3 OpenCV Library

OpenCV is an open-source library for computer vision and machine learning. It was built to provide a common infrastructure for computer vision and machine learning applications with more than 2500 optimized algorithms. The library is natively written in C++, however, there are interfaces for other languages such as Python, Matlab and others [12]. OpenCV library can run under Linux, MS Windows or even Mac OS X and Android [12].

For the purposes of this thesis, I will use the OpenCV library with the Python interface. Python was chosen for its intuitive syntax and relatively low requirements for computing power, which is useful for implementation on a single-board computer, such as RaspberryPI or Nvidia Jetson.

1.4 Edge Detection

Edge detection in general is one of the fundamental procedures in computer vision. The main objective is to detect sharp changes in pixel intensity[11], which calls for the implementation of a derivative of first and second order. In the following text, edge detection techniques and its components chosen for this project are explained.

1.5 Edges

As suggested above, the edge is defined by a abrupt change in the intensity of the pixels (or the gradient (eq. (1.12)) value is extremal). These groups of points (edge points / pixels) are a valuable source of information about the content of an image, because they can create regions boundaries[3]. If the input image is turned into grayscale scheme to remove excessive information, reserving only the information about pixel luminance, the computation is going to be less demanding, and, therefore, faster.

It is possible to classify edges by their intensity profile. A transition between 2 intensity levels is called a Step edge and represents "an ideal edge" [11]. A Ramp profile is slightly more realistic. As the name suggests, the transition runs over more than one pixel and the slope is proportional to the level of blurring of the edge[3]. The roof edge is another model of an edge, where the base of the "roof" is determined by the width and sharpness of the edge [3].

One of the special cases is a corner. Corners are interesting, because they can be localized, but edge detectors and gradient operators often fail at corner points [4]. It is due to the smoothing of the image, which covers the corner. These points are characterized by a sharp change in gradient orientation in a small neighborhood. There are many corner detectors; however, one of the most common is the Harris corner detector.

1.6 Prewitt mask

The Prewitt mask is a 3x3 convolution kernel, which computes the gradient using local averaging in horizontal direction with one kernel (G_x) and in the vertical direction with (G_y). This helps reduce noise compared to, for example, the Roberts operator, which implements a 2D kernel [1].

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \; G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.19}$$

1.7 Sobel mask

Sobel operator/mask is another one of many gradient-based edge detection operators. It utilizes two convolving kernels, one in the horizontal G_x direction and the other in the vertical (G_y) direction. Each of the kernels can be applied separately, resulting in two separate gradient images. The Sobel operator draws many similarities to the Prewitt operator, with the main difference being the value of 2 (-2 respectively) in the centre coefficients. The higher value of those pixels gives them more weight. Because of this, the resulting image computed with the Sobel operator is less noisy than the Prewitt image.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \; G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1.20}$$

1.8 Canny edge detector

The edge detector of choice for this project will be the Canny edge detector. It is considered to be one of the most widely used edge detectors in computer vision. John Canny proposed this edge detection method with three main goals: bringing the number of false edge pixels to minimum, improving edge localization, and delivering a single mark per edge [2]. To achieve these goals, multiple methods are combined. It implements Gaussian smoothing to reduce noise level, gradient operator to compute edge gradient and its properties, and zero-crossing to improve edge localization.

Because the definition of the Canny edge detector is complicated, the process of detection using the Canny edge detector can be described simply in the following steps:

- Use 2D Gaussian to smooth the image
- Use Sobel (or Prewitt) mask and compute the image gradient
- at each pixel, find the gradient magnitude
- at each pixel, find the gradient orientation
- at each pixel, compute Laplacian along the gradient direction
- Find zero crossings in Laplacian to find the edge location

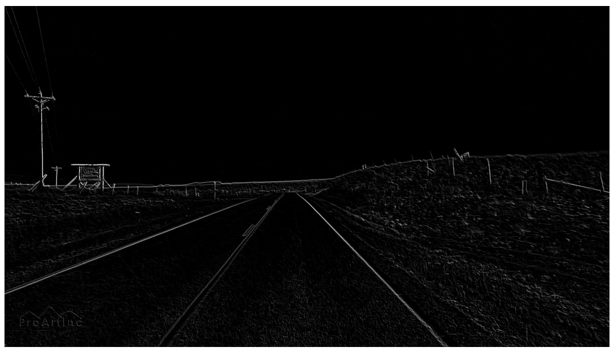
It is important to realize that the output of the application depends on the level of smoothing; more specifically, it depends on the choice of the sigma parameter σ (1.2.4). This effect could be represented by a simple example:

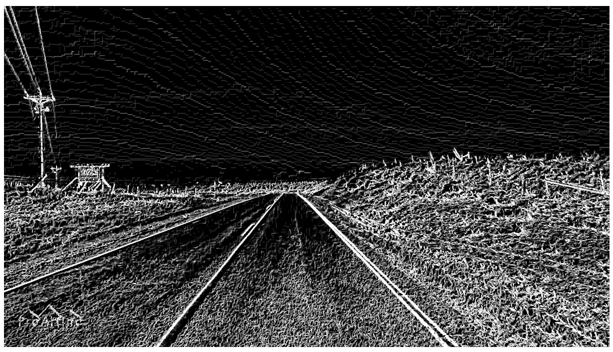
Imagine a very high-resolution image of an old asphalt or a tarmac road. If a Canny edge detector is applied, the result is going to be very busy, including damage caused by vehicles, weather conditions, etc., and differences in texture of already repaired areas and the original surface of the road. Although it might sound very desirable, it is the opposite: This image is very busy and contains edges, which have little to no use in practice. Instead, if we manage to blur the image, the result will be less and less busy. However, it is necessary to find a compromise between the sigma value, because we do not want our image to be too blurry [13].

It depends on what is important for the user, and the Canny edge detector allows us to change this via the change of one parameter, the sigma.

In Figure (1.4), you can see the difference in the output image using different operators. For the image processed by the Canny edge detector, the low and high thresholds are 50,150 (a recommended ratio of thresholds is between 1:2 and 1:3 [9])

(a)


(b)


(c)

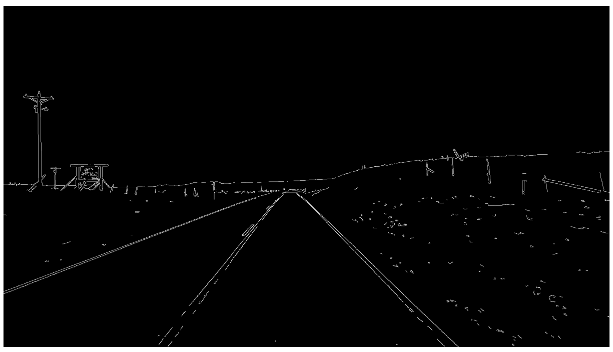

(d)

Figure 1.4: Comparison of edge detectors: (a) original image, (b) Prewitt, (c) Sobel and (d) Canny

1.9 Thresholding methods

Thresholding methods are used for image binarization, making further processing much easier, if not possible. It is possible to classify thresholding methods into two categories – fixed threshold methods, or adaptive thresholding. Fixed threshold compares the value in the gray hue image to the specified value, which stays constant [14]. These methods are the simplest way to create a binary image, but do not provide the flexibility needed for most applications. Adaptive thresholding methods improve methods with a fixed value and automatically compute the threshold value. In [15], the thresholding methods are classified into six groups according to the information they are exploiting.

- Histogram shape–based methods
- Clustering–based methods
- Entropy–based methods
- object attribute–based methods
- Spatial methods
- Local methods

1.9.1 Otsu Binarization

Otsu method is one of the most referred [15] clustering–based methods [14]. Being introduced in 1979 by Nobuyuki Otsu, the algorithm uses image histogram to define a suitable threshold value, which then separates pixels into two classes, background and object (two clusters). The ideal threshold is automatically selected by the discriminant criterion, namely the measure of separability of the resultant classes at gray levels [16]. Beneath this section, you can see the results of applying Otsu to a grayscale image in Python:

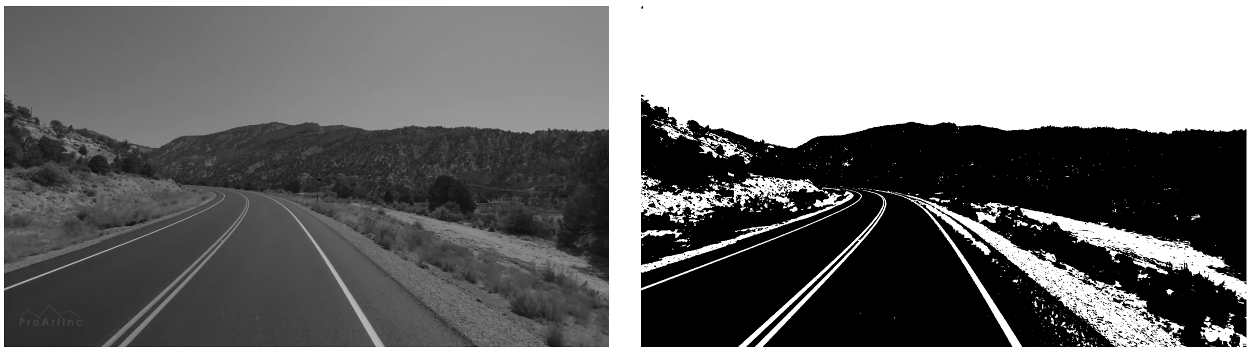


Figure 1.5: (a) Gray image; (b) Otsu applied to (a)

1.10 Perspective Transformation

The perspective transformation will play a key role in the algorithm. Because perspective effects, such as non–constant lane mark width and different distances make lane detection more difficult, it is beneficial to remove those effects by remapping the image to a bird–eye view.

Perspective transformation could be briefly described as applying a transformation matrix M to the points in the source image, which returns new points in the output image. This type of transformation does not preserve parallelism, length, or angle, and changes the viewpoint [17]. The basic principle is to represent the pixel of n dimensions in a $n+1$ dimension vector space, and with the use of homogenous coordinates [11]. For example, a typical image pixel has 2 coordinates – 2D, but for the calculations, a 3D representation of this pixel will be utilized. Thus, the transformation matrix M is a 3×3 matrix.

$$\begin{bmatrix} t_i x' \\ t_i y' \\ t_i \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \\ c_1 & c_2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.21}$$

Where (x', y') the coordinates of the transformed point and (x, y) is the source. Matrix M is the transformation matrix.
Since there are 8 degrees of freedom in M, first step is to select four points in the source image and map these points to the specified location in the output image [17]. Due to OpenCV (1.3), this is all that is necessary to get the matrix M.

1.11 Hough Transform

Let's imagine a self driving car. The car, for sure, uses some kind of camera to capture the road in front of the car, to keep it on the road. That usually means keeping the car in between some lanes. To achieve this, the computer needs to calculate the equation of these lanes to 'tell' the car to stay in between them.

The Hough transform is one of the feature extraction methods, which can be used to find features of a specific shape, usually one of the basic shapes, like ellipses, lines, etc., often called the primitives. It implements a transformation from regular image space to a parameter space, also called the Hough space, which in concept is an accumulation array. One of the common (and in my opinion one of the best) analogies of an accumulator array is a grid of voting buckets. [4] The sought outcome is the maxima in the parameter space - a bucket with the highest number of votes (just like the accumulator), which should represent the value of parameters of the searched structure. The question raised by the concept of an accumulator array is how large (or small) the cells should be. If the cells are too large, the lines merge, and if the cells are too small, noise will kill the line.

1.11.1 Finding a line

The algorithm of the Hough transform to detect a line is based on a parametrization of a line. Typically, there are two most commonly used mathematical representations for a line (let us assume two-dimensional x, y plain):

$$y = ax + q, \tag{1.22}$$

$$r = x\cos\theta + y\sin\theta \tag{1.23}$$

The choice of representation depends on the user; however, in this section, I would like to use equation (1.22), because it is easier to imagine, in my opinion, and therefore more suitable for demonstrating the main principle of the Hough transform. Each point (or so–called tokens) has its coordinates x, y, but the other 2 parameters, in this case, a and q, both of which are unknown, so there is no exact line specified. To find these values, the Hough space is created and each token is transformed to this space. The Hough space is a plain with an axis, where the variables are the searched parameters a and q. For each point in the original image space, a line can be created. As the algorithm goes through the tokens and the number of lines in the Hough space increases, crossings of these lines might occur. These intersections are the votes mentioned above. With each crossing, the color of the point of intersection might change, the number of votes in this bucket rises. The coordinates of the lightest point (or bucket with the highest number of votes) represent the values of the parameters, and thus a line can be constructed.

In the case of perfectly straight lines, all lines should cross one point, and the Hough space should display only one intersection, one bucket, which clearly wins the election. However, in a more realistic example, multiple intersections are going to occur and some lines might be separate. This means that the point in the image space does not form a perfectly straight line, and there are more possible parameter values, or, in other words, multiple lines.

Chapter 2

State of the Art in Lane Detection

Lane detection is an essential part of driving assistants and autonomous driving [18]. Generally, a computer vision process can have three main steps [19]:

- Image acquisition
- Image processing
- Image analysis + decision–making

For example, at first, the image has to be captured from a camera. Next, pre–processing methods (usually less complex operations [19]) are applied to improve the general suitability for further processing. And finally, information is extracted from pre–processed, analyzed image. On the basis of the results of this step, decisions can be made.

Lane detection methods could be divided into separate categories based on whether the method is classical or uses deep learning [20]. A classical approach involves the use of several computer vision and image processing methods to extract specialized features and identify the location of the lane segments. Subsequently, post-processing techniques remove false detections and join sub–segments to obtain the final positions of the road lane [20]. Although less complex, these methods do not cope well with sub–optimal illumination or complex road scenes [20].

A deep learning–based method presents an improved result and is embedded in the present autonomous driving systems. These methods can be further separated into two other categories: segmentation–based methods and adversarial generative network-based methods [20]. Deep learning–based methods use large datasets for learning. Some of the leading companies and universities in this field have developed such datasets, such as the AI trucking company TuSimple or CalTech (Californian institute of technology) [21].

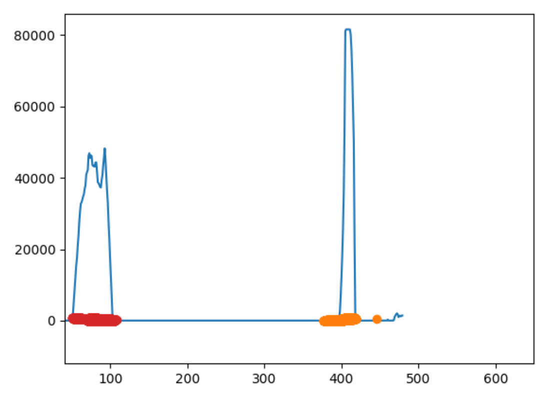
For the purposes of this thesis, a classical approach algorithm, the sliding-window algorithm, was chosen.

2.1 The sliding window approach

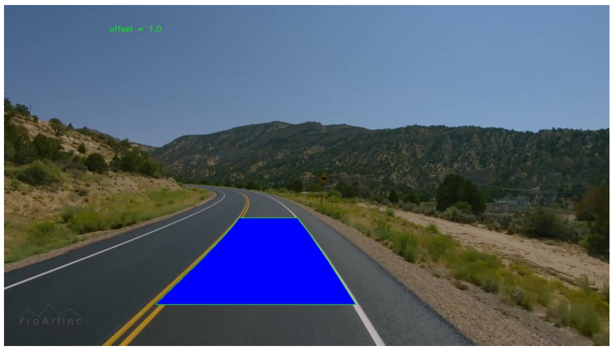
The sliding window approach, in general, can be used to reduce the search area by reducing the computational complexity [22]. An interesting application of this approach is the detection of railway lines.

Probably the most common way to determine the position of the sliding window(s) is to find peaks in the image histogram, where the horizontal axis represents the pixel columns, so that the peaks in the image can be located, while the vertical axis represents the intensity [23]. These computations are generated after converting the image to gray (1.1.1) and after warping (1.10), so the image is preprocessed for further calculations. Inside the window, a filter is applied to the potential edge points, which examines their gradient value and the mean squared error approximation [22], so the result becomes the center of the next sliding window. When enough points are obtained, it is possible to attempt to fit a polynomial line through these pixels.

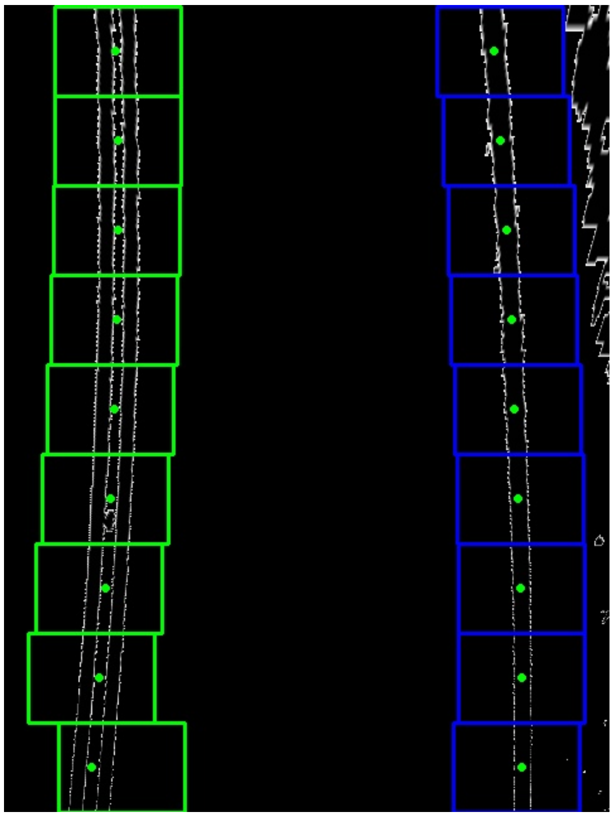
One of the challenges for this approach is the very sharp left or right turn, since this algorithm, in its standard form, assumes that the next sliding window will be "above" the current one. The article [22] from Northern Illinois University presents an improved version of this approach, reporting interesting results with a detailed explanation of their solution, which considers the previous lane as a starting point.



(a) image's histogram



(b) Resulting image



(c) Sliding window

Chapter 3

Concepts

Before explaining the final implementation of the algorithm and its results, I would like to spend some time introducing approaches that were assessed as nonoptimal at best or, at worst, not working at all. That said, if the approach mentioned in this chapter is described as not working, it is not meant, as it is impossible to implement. I speak from my point of view, from my own experience.

3.1 Use of Canny detection and Hough transform

With both algorithms described in detail earlier (1.8, 1.11), I decided to use both in the first attempt to solve this problem. This version of the algorithm is the least complex, taking only the warped image, with the applied Canny edge detector (1.8) in OpenCV. Combining the resulting image with the line detection algorithm, the Hough transform (1.11), and warping the image with detected lines back to its original perspective, ended up being sensitive to lighting conditions, such as shadows and incorrectly chosen threshold values. With ideal lighting conditions and a straight road ahead, the output was acceptable, but once the vehicle approached a curve, the result fell apart and was unusable for further calculations. Seeing those results, I decided to abandon this concept and find a different solution.

3.2 The histogram

This time, instead of using the Canny edge detector, I opted for the Otsu (1.9.1) thresholding, which created a binary image of the warped region. The warped binary image was then sent to a function, which sliced the image into horizontal strips and calculated histograms from each of them. With a defined middle point, I managed to detect the left and right lane lines separately. Each point on the lane line was the result of finding the maximum value in the slice histogram. The results of this solution were a significant improvement over the previously mentioned one. However, it still had its limitations. Firstly, it was difficult to find the balance of number of slices and sensitivity to shadows and other sources of inaccuracies - if the number of slices was too high, the output image was affected even by tiny defects in the image, the histogram was calculated for a slice so thin it couldn't cope with the smallest of errors, like shadows or poor lightning. A clear example was that when the middle lane line was doubled, the lane line points for this side of the road often jumped from one lane to another, leading to an inconsistent line.

On the other hand, when I reduced the number of slices, the resulting polygon lacked any smoothness, and there the sensitivity problems were still present. I ended up spending many hours trying to fix this solution and make it work, creating many support functions to find any improvement. This was not achieved. However, many of the principles applied in this approach and some of the functions created made their way into the final version of the algorithm, which I will introduce in more detail in a separate chapter.

Chapter 4

Implementation

In this chapter, I will walk through the experience of building the toy vehicle's kit, with advice for possible future users included. In addition, the idea behind the algorithm for lane detection will be shared. Core functions with their source code will be included with a more in–depth description of their principles.

4.1 The build

This project focuses on the topic of an autonomous vehicle. So far, everything mentioned was just theory. In this chapter, I would like to introduce the practical side from building the actual test vehicle (which due to the current situation brought us many surprising problems) to the details of testing the software.

The testing vehicle consists of two main parts, the chassis, and the computer. The chassis is the JetRacer AI racing robot by Waveshare, provided to me by the university. And here comes the first problem: due to the ongoing pandemic of COVID–19 (status 2021), the university uses an online school concept, and all students and most of the lecturers stay at home. So, picking up the kit meant that someone had to give it to me; fortunately, after a short exchange of information, everything was agreed and it was possible to pick up the kit.

Right after opening the box, I noticed two things: There are many screws of different sizes and shapes, and the manual is small, which makes it hard to follow. Anyway, following the instructions, the 'car' started to shape. A small piece of advice for anyone building this kit is this: Keep everything in place and follow the instructions. The screws are tiny and look almost identical with few differences between them. Also, for the included screwdriver and wrench, a regular size quality screwdriver is proposed; however, do not

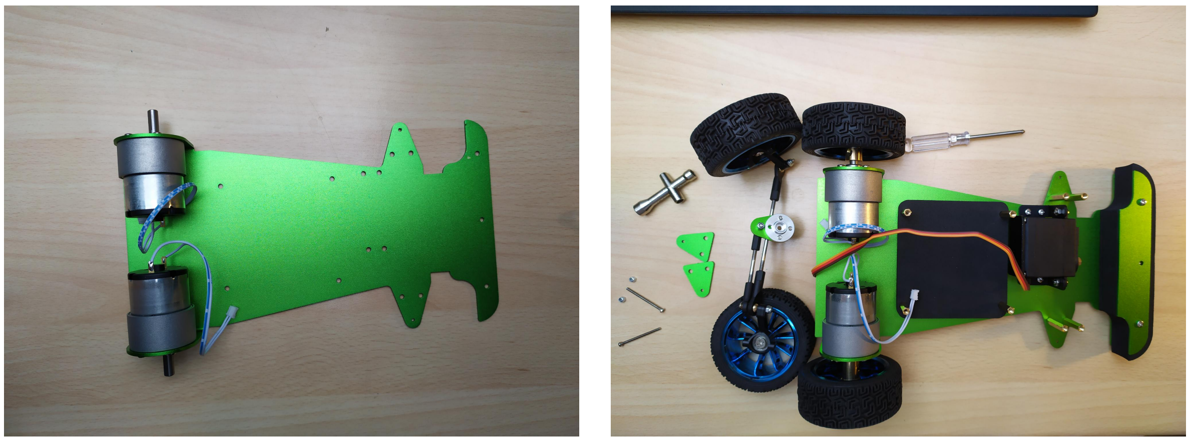


Figure 4.1: The vehicle build

put the included one away. Some areas are tight, so the small screwdriver helps.

At a certain point, you will find some aluminum parts that are sometimes sharp and do not feel quality, so you must be careful. With this being mentioned, the overall quality of this kit is, in my opinion, questionable - it looks very nice in the picture; however, the aluminum body parts are, as said, sharp, and the screws have to be treated with care, to not damage its head. Then there are, for example, the parts of the steering rack. The front axle's screw–threads are poorly finished, as are the ball joints, to which the wheels are mounted. Speaking of ball joints, during the build-up, I was careful not to damage anything. However, when my father came to inquire about my progress and fiddled with the steering, the right front ball joint cracked, releasing the wheel. After some research, it was found that it is not going to be easy to replace this part. I checked if there are any OEM parts available; however, I was unable to order this specific part. So, the only solution was to go to the modeling store. Successfully, the ball joints arrived, but after a couple of attempts, I realized that due to some manufacturing inaccuracies, the axis will not fit the joint, so the joints had to be adjusted to the correct diameter. Fortunately, everything past this went well, but it just confirms the point that this whole kit is not of the best quality. However, the only thing that matters is functionality.

This AI kit is powered by Nvidia's Jetson NANO developer kit. It is a small single–board computer specially developed for running AI. This vehicle is fitted with the more powerful 4GB version (more details of the specification list are included later).

Another popular choice for these types of projects is usually a RaspeberryPI. There are some differences between the Jetson and the RaspberryPI, one of them being that they use different charging connectors - whereas the RaspberryPI uses a USB-C charger, the Jetson is powered either by a single–pin charger or micro USB. This isn't a problem unless you unpack your order and you find out that the charger included in the box is a

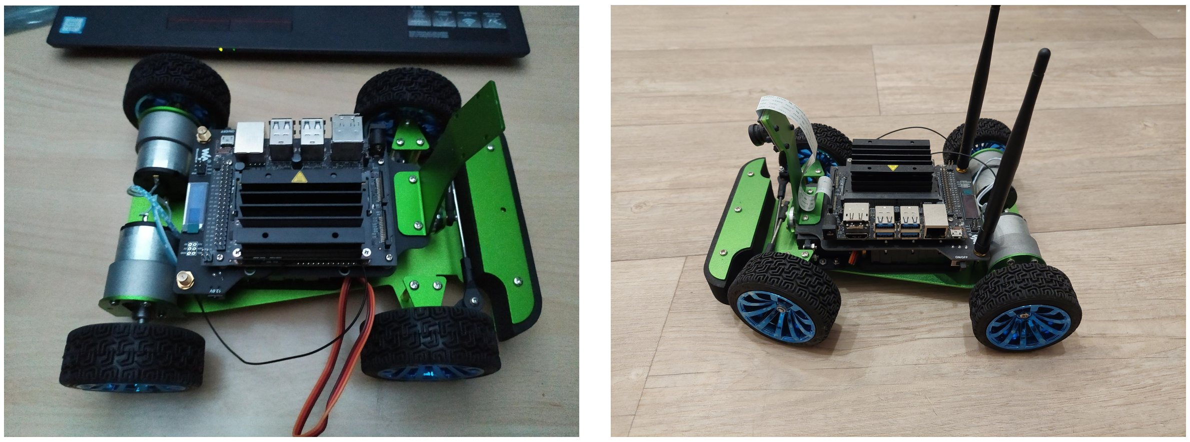


Figure 4.2: a) nearly finished vehicle, b) finished vehicle

Raspberry charger, which isn't compatible with the Jetson, so it had to be replaced.

When the Jetson was mounted on the vehicle, everything was ready for the installation of the operating system on the vehicle. Jetson uses a micro SD card as a storage solution, so a card reader and 2 micro SD cards were ordered just in case. However, probably by accident, one of the two ordered SD cards was missing. After further investigation, I realized that the missing SD card was replaced with a second card reader.

Nvidia's software support for Jetson is exemplary. Everything is easy to setup, or at least described by a detailed guide, so that anyone can install the required software. Simply download the recommended SD card formater and OS, format the SD card and load the OS onto it, with the recommended software featured on the Nvidia website. The installation itself is just a very simple 5–minute process. (the SD card included in the package ended up being faulty and failed to install the OS, so it had to be replaced with another, higher–performance SD card) The OS, provided by Nvidia, is essentially Ubuntu, with some pre–installed applications for the Jetson, and a signature graphic theme.

Next, all the necessary software has to be installed. First, Samba allows easy access to files from another computer, so there is no need to use a portable drive of some sort. Second, for remote desktop access, the NoMachine software is installed. NoMachine was chosen because it offers low latency, a nice set of features, and support, while being a freeware. For coding, Visual Studio Code by Microsoft was chosen, due to its relatively low hardware requirements and intuitive interface. Later in the project, I switched to PyCharm by JetBrains, which provides a more complex environment with GIT support to improve the project's efficiency.

The vehicle was installed and ready for testing. Following the guide on the manufacturer's website, I decided to test whether everything works as it should. Unfortunately, I was unable to connect to the car computer. Even when following the guide step by step and contacting support, I did not receive the expected result. Later, when the manufacturer published a new version of the software, this problem was resolved. At the time of publishment, I was already short on time to do further testing with the actual vehicle. The fact that the vehicle was equipped with a 64 GB microSD card did not help either; since the new software was around 67 GB before installation, a new card had to be purchased, which shortened the time for testing even more.

4.2 First software implementations

Technically, a vehicle capable of reacting to the situation "ahead" and also if it manages to keep the vehicle on the road could be called autonomous to some degree.

In this thesis, the goal is to create and implement an algorithm to keep the vehicle on the road. There are several solutions available on the Internet, of which few are widely used and are then trained by many to achieve the best results possible.

4.2.1 Building the algorithm

With concepts described in previous chapters of this thesis, the build of the algorithm can start.

From my personal experience, most of the solutions available on the Internet are built with the use of object-oriented programming. Although it has unquestionable benefits, I decided to opt for classic C-style functional programming. The algorithm could be broken down into the following steps:

- the image to grayscale,
- create a Otsu binary image,
- warp a defined region in the binary image,
- detect starting points and perform a sliding window algorithm,
- warp lane points to the original perspective,
- calculate offset, and lane curvature.

First, a video must be loaded. The image is then prepared for edge detection. Firstly, to remove any irrelevant information from the image, grayscale and otsu thresholding is applied.

```
def Gray_scaNumPyg):
    return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

def Otsu_thresh(img):
    otsu_threshold, img_resulthe = cv2.threshold(
        x-axis, 100, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU
    )
    return img_result
```

Listing 4.1: Grayscale and otsu

The grayscale function is very trivial – takes the image as an argument and returns the same image, just turned into a grayscale scheme. The Otsu threshold, still relatively simple, has more attributes available for configuration; for example, we can adjust the threshold value, which decides whether the pixel is evaluated as black (0) or white (255). The poorly chosen threshold value can affect the resulting image, which may later lead to incorrect calculations.

The next step is to perform a perspective transformation, which is a key component to extract the data necessary to detect lines. Again, the OpenCV library provides predefined functions, which create a transformation matrix based on two user-defined sets of points. In the code, I created a function Perspective_transform, which takes the binary image, the set of quadrangle vertices from the binary image, and the width and height of the warped image. Firstly, it defines the warped image's quadrangle vertices and transfers those points from lists to a single NumPy array, which is accepted by the OpenCV function getPerspectiveTransform(), which calculates a 3x3 transformation matrix. This matrix is applied to the binary image in the next line with warpPerspective(). The warped image, as well as the transformation matrix and the array of vertices of the warped image, is then returned by this function, as seen in the code below:

```python
def Perspective_transform(img, points, width, height):
    output_left_top = [0, 0]
    output_right_top = [width, 0]
    output_left_bottom = [0, height]
    output_right_bottom = [width, height]
    output_points = np.float32(the [outpleft_top, output_right_top, output_right_bottom,
    output_left_bottom])
    perspective_matrix = cv2.getPerspectiveTransform(points, output_points)
    warp_img = cv2.warpPerspective(img, perspective_matrix, (width, height))
    rcopiedarp_img, perspective_matrix, output_points
```

Listing 4.2: Perspective_transform

With the transformed binary image, the sliding window algorithm (2.1) can be used. Because the introduction is included in the separate chapter referenced earlier, lets look at the core sections of the code.

```python
def Window(img, nwin, win_width, recenter_th):
    histogram = np.sum(img[img.shape[0] // 2:, :], axis=0)
    out_img = np.dstack((img, img, img)) * 255
    midpoint = int(histogram.shape[0] // 2)
    left_bottom = np.argmax(histogram[:midpoint])
    right_bottom = np.argmax(histogram[midpoint:]) + midpoint
    win_height = np.int(img.shape[0] // nwin)
```

Listing 4.3: Window

First, the function takes as arguments the warped binary image, the number of windows per lane nwin, width of a window win_width and the recenter threshold recenter_th. When the frame is loaded, the function calculates a histogram of the bottom half of the loaded frame, since this is the region. If we performed a histogram of the whole frame, the resulting peaks could be wider, and therefore we could risk potential inaccuracies. To divide the histogram into two parts, left and right, it is necessary to define the value of the midpoint, which is simply the maximum value of the x–axis divided by 2. To find desired peaks for each lane line, we searched for the maximum value in each of the histogram's two parts. Note that if the center lane is doubled, as in one of the sample videos, it can result in two peaks on the left part of the histogram. Then the height of a single window is defined by dividing the height of the frame by the argument nwin, the number of windows per lane.

Second, with the peaks found, it is useful to find the location of nonzero pixels in the image. This can be done in multiple ways, but probably the most efficient is to use the NumPy function np.nonzero, which returns, in this case, two arrays, where one carries the x–coordinate of each nonzero point, and the other one carries the y–component. This idea was inspired by [24]. Then, a list is initialized for each lane to store lane points.

The third step is finally to use the sliding windows. In a *for* loop, each iteration defines opposite corners for a single window in the left lane and for one in the right lane. The rectangles are then plotted for visualization. As mentioned in (2.1), the recentralization of the following window is based on the number of nonzero pixels in the current window region. Here again, I implemented the np.nonzero method:

```
        l_pix = ((nonzero_y >= y_min) & (nonzero_y < y_max) & (nonzero_x >= xl_min) & (
    nonzero_x < xl_max)).nonzero()[0]
        r_pix = ((nonzero_y >= y_min) & (nonzero_y < y_max) & (nonzero_x >= xr_min) & (
    nonzero_x < xr_max)).nonzero()[0]
```

Listing 4.4: Window_region

If the number of nonzero pixels in the window's area exceeds recenter_th, the next window will be recentered to the mean of the x coordinates of detected nonzero points. The y-value is calculated by the algorithm.

Next, the center of each window is appended to the list poly_points_left / poly_points_right, which is then copied to the list poly_points with the use of list comprehension. I want to point out that the poly_points_right is copied in reverse order - this is due to the fact that poly_points will be used for visualization, and the OpenCV function fillpoly is sensitive to the order of the points. In other words, if I did not use the reverse order of poly_points_right, the polygon could be displayed with defects, as I described in (pokusy). The code implementation can be seen in the listing below

```
        if len(l_pix) > recenter_th:
            left_bottom = np.int(np.mean(nonzero_x[l_pix]))
        if len(r_pix) > recenter_th:
            right_bottom = np.int(np.mean(nonzero_x[r_pix]))
        l_point = [left_bottom, (y_min + win_height//2)]
        r_point = [right_bottom, (y_min + win_height//2)]
        l_circle = (l_point[0], np.int(np.abs(l_point[1])))
        cv2.circle(out_img, l_circle, 2, (0, 255, 0), 2)
        poly_points_left.append(l_point)
        poly_points_right.append(r_point)

    poly_points = np.asarray([j for i in [poly_points_left, reversed(poly_points_right)]
    for j in i], dtype="float32")
```

Listing 4.5: Window_region

In real vehicles, lane-keeping assist is almost always supported by lane-centering assistant. This assistant helps keep the vehicle in the center of the lane. Although not within the scope of this thesis, a simple offset calculator was created.

```
def car_position(points, img):
    car_position = np.array([img.shape[1] / 2, points[0][0]])
    right_bottom = points[-1]
    left_bottom = points[0]
    lane_width_px = math.dist(left_bottom, right_bottom)
    increment = lane_width_px // 2 + left_bottom[0]
    lane_center = np.array([increment, car_position[1]])
    offset = math.dist(car_position, lane_center)
    return offset, lane_width_px
```

Listing 4.6: Car_position

It utilizes the coordinates of two bottom points of the detected lane region and assumes that the vehicle is in the center of the image (the camera is in the middle of vehicle). The width of the lane is calculated as the distance of the points of the detected lane region and then calculates the offset as the distance between the center of the lane and the vehicle's position.

Chapter 5

Testing and Evaluation

Testing was firstly done on a couple of short sample videos, which were cut from a dashcam recording available on YouTube. Each of the samples was of a different complexity for processing – driving in straight, driving in a slight curve, and driving in more prominent curve. In addition, the lighting conditions differed from one sample to another.

The recording of driving straight with solid lighting conditions did not cause any problems, with the algorithm showing good results even when a light shadow occurred.

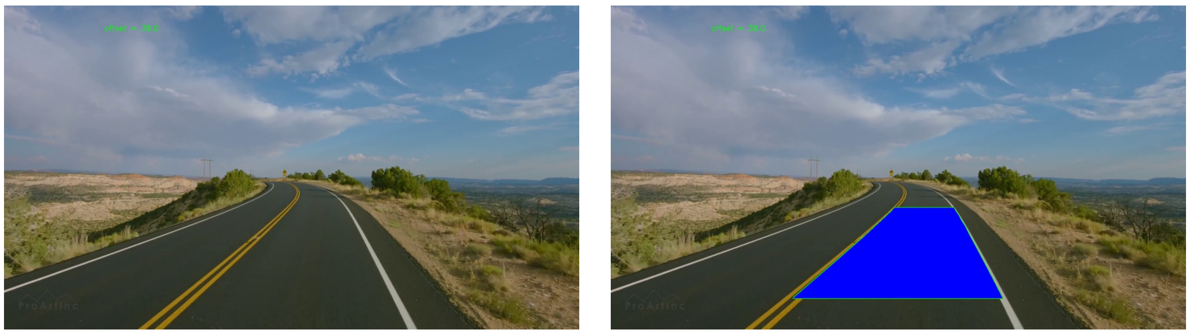


Figure 5.1: straight source, straight output

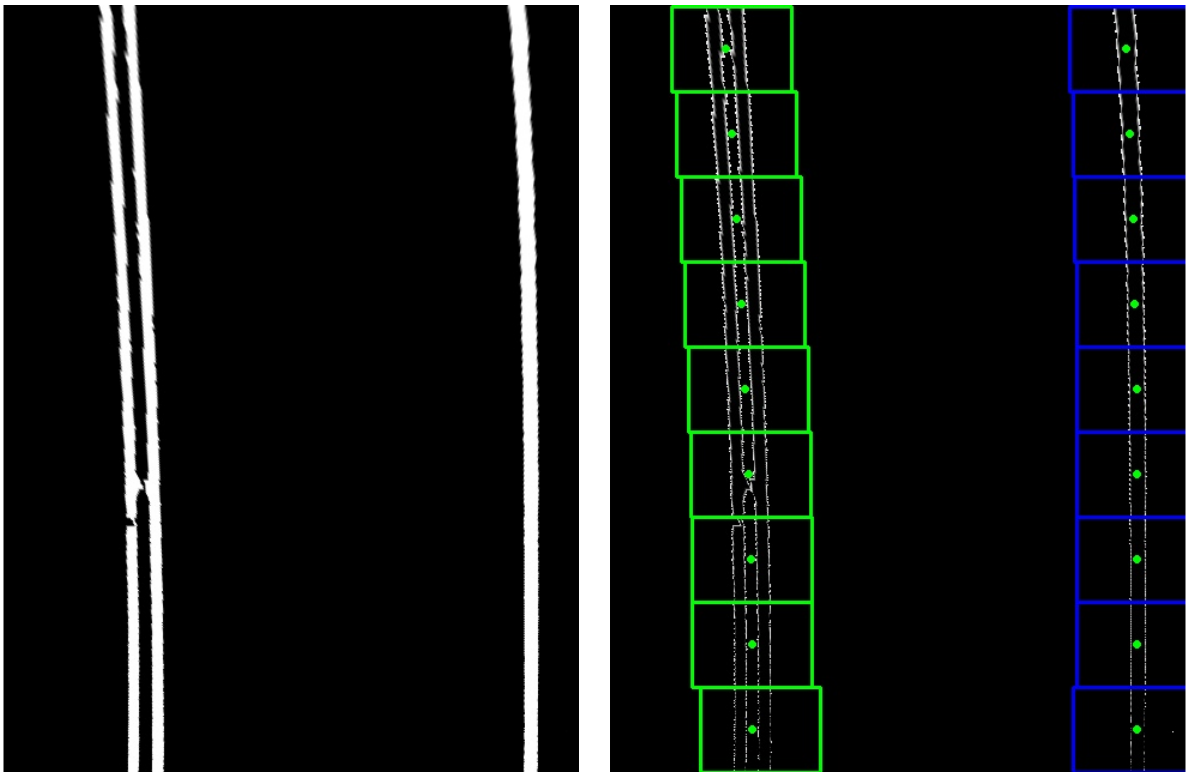


Figure 5.2: straight binary, straight sliding windows

While going through a slight curve, the algorithm performed decently in most situations. The only problem that occurred in those scenarios had to do with the highly contrasting edge of the road or lighting imperfections, which caused a slight deformation of the detected region. This problem is demonstrated by the following figure:

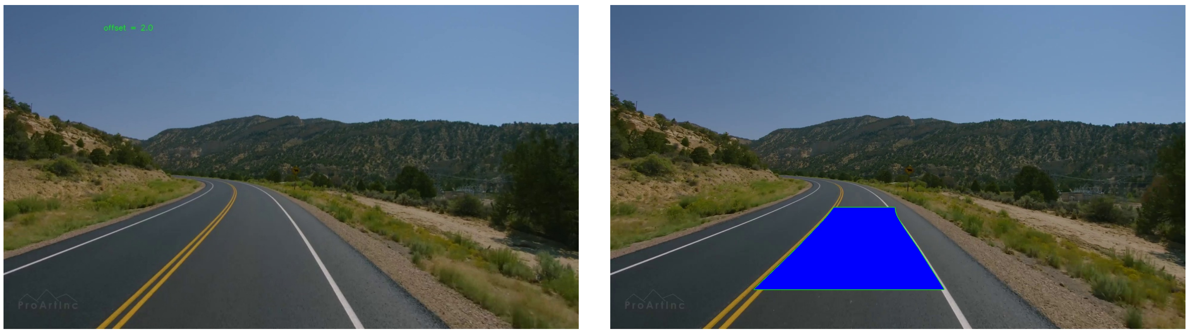


Figure 5.3: slight curve source, slight curve output

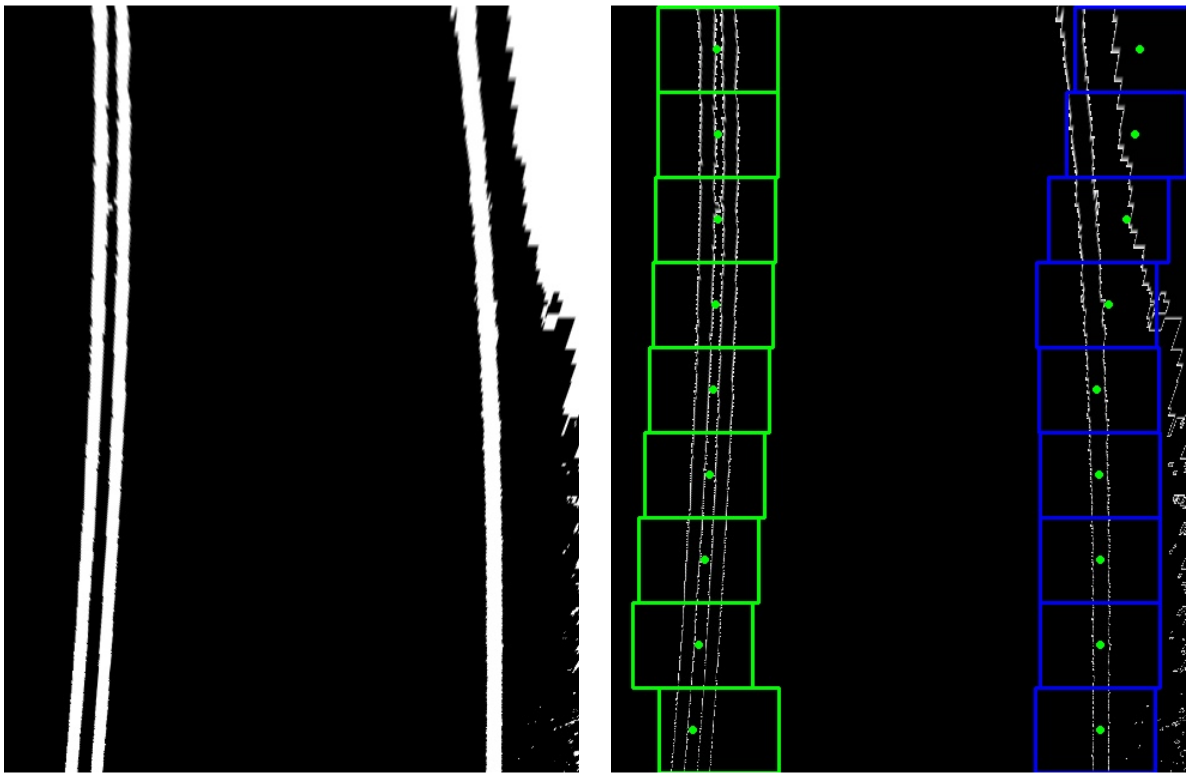


Figure 5.4: binary image, sliding windows

It should be noted that this was often corrected by a more precise selection of corner points of the source image for perspective transformation (1.10).

Dealing with more prominent curves we discovered significant drops in performance. In a few frames of testing videos, under specific lighting conditions, the algorithm returned decent outputs with little to no defect of the blue polygon:

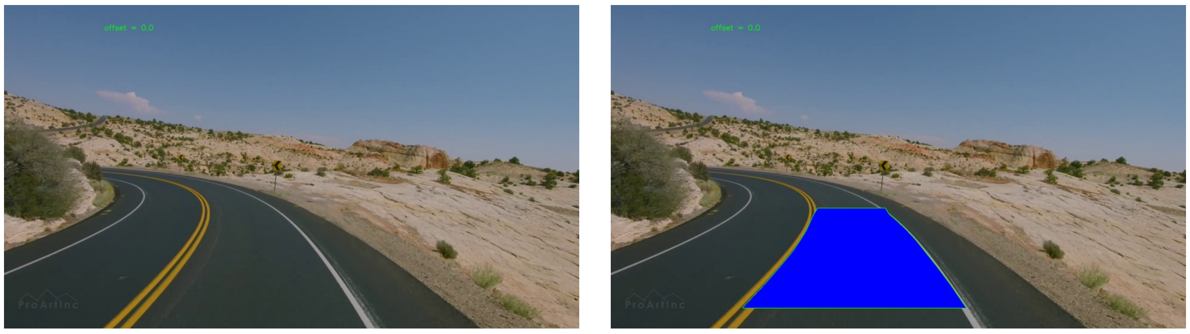


Figure 5.5: curve source, curve output

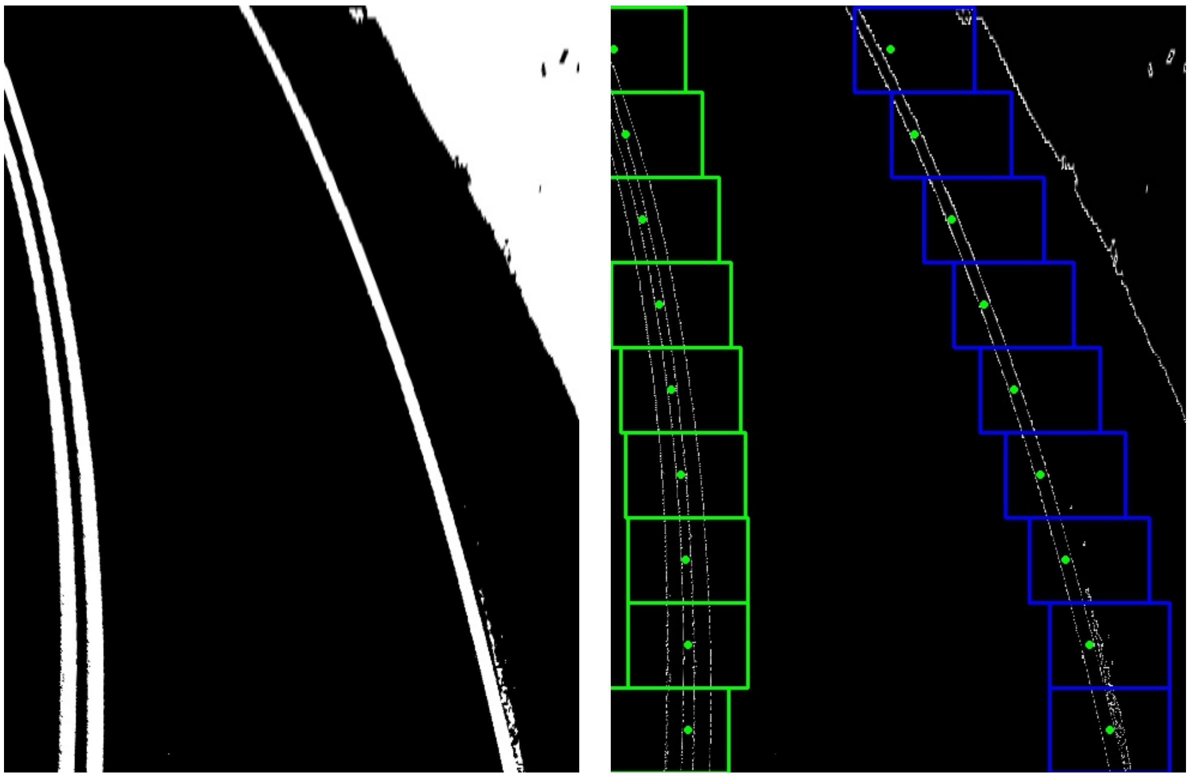


Figure 5.6: curve good binary, curve good sliding windows

However, in general, the increase (technically loss) of the curvature caused the upper points of the trapezoid to not capture the road, causing the position of sliding windows in this part of the image to be evaluated incorrectly, resulting in an insufficient output image, as shown in the figure below:

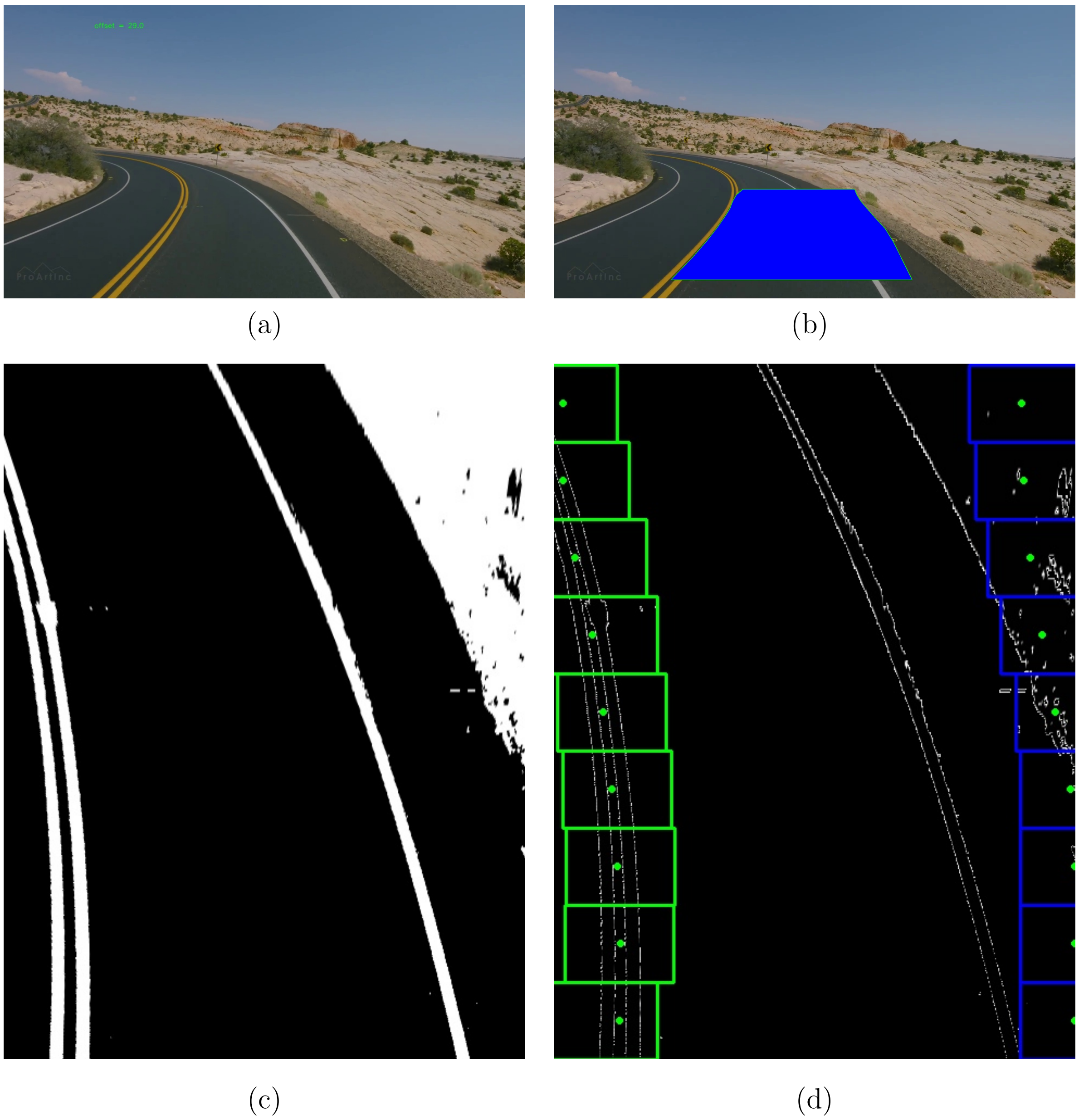


Figure 5.7: Lane detection on a curved road: (a) original image, (b) output image, (c) Transformed binary image (d) Sliding window

Unlike the previous case, this could not be improved enough by a better selection of warping points. The algorithm does not provide a sufficient tool to cope with this type of situation. One of the possible solutions to this problem could be the use of more complex binarization method, as in the prior case, in combination with an adaptive distance of detection – the main idea being that the vehicle approaching a sharp turn will not travel at the same speed as on the straight or mild turn, therefore, lowering the viewing distance for the detector should not bring issues. The decrease in distance would have to be tied to the angle of the road, which means something like comparing the offset of individual windows, which would, if conditions are met, decrease the distance.

The language of choice for this project is, as stated before, Python. While Python is very popular for its relatively simple syntax and well–established community providing countless libraries, the performance is not on par with C–style languages, like C++ or Java, which are also supported by OpenCV (1.3). Python still provides decent performance, if supported by powerful hardware.

One iteration of the algorithm in its final form was measured using the `time` module. This test was done on a laptop with the following specification:

- CPU: AMD Ryzen 7 5800h
- GPU: AMD RX Vega 8
- RAM: 16 GB DDR4

On this hardware specification, the algorithm runs at average 17 FPS. Comparing the hardware specification of the laptop to the specification of the Jetson Nano, it is certain it would take longer; therefore, to improve the overall performance of the algorithm, a conversion from Python to one of the other supported languages is recommended.

If there were no desire to convert the program to a different language supported by OpenCV (1.3), some performance gains could be made by optimizing the code. It is known that the efficiency of Python code can be improved if predefined functions are utilized. In the source code, it is possible to see that in some instances I implemented `for` loops. While this solution is functional, it is not exactly pythonic. More performance could be found by replacing these loops with predefined functions.

Conclusion and Future Work

The goal of this thesis was to develop a lane detection algorithm and test it on a test track for a toy vehicle. In this text, I depicted a selection of popular color spaces, showing their advantages and disadvantages, further, I described why it is beneficial to convert the image to grayscale.

Next, the key mathematical operators and concepts were introduced. I moved to the discrete world to define discrete derivatives, which are essential for edge detection. To be able to understand the basics of image processing, ideas like convolution or Gaussian had to be presented.

To utilize some of the mathematics briefly mentioned above, the subject of edge detection could be brought up. Together with it, I looked back in the past to show different convolution masks, which were utilized for edge detection. This enabled me to present the Canny edge detector, as well as other concepts essential for this thesis.

Furthermore, the current state of the art in lane detection was introduced, introducing the current state and trends in this field.

Moving to the practical part of the thesis, I briefly presented a couple of approaches I tried before deciding on the final solution.

Later, I shared the experience of building the toy robotic vehicle and made thoughts on the whole process, including the installation of software. With the vehicle built, I was able to move on and present the specifications of the final version of the algorithm, including code samples of key functions, and I described them in detail.

Finally, the results and evaluation of the testing were presented. Here, I showed the results of the testing and come up with possible improvements to the algorithm.

The lane detection algorithm was developed. Utilizing the Sliding window method, it performs well, if certain conditions are met. The areas where the algorithm in this state lacks, are poor lighting conditions, like shadows or lack of contrast of the road and lane lines. Also, it was proven that the sliding window approach does not produce constant results if the road is of significant curvature. The solution to this problem might be the adaptive ROI, which was briefly presented in this thesis, and might be worthy of further investigation. For future work, I propose adding a curvature calculation function, which can be later used to operate the steering rack, as well as improving the Car_position function and adding conversion to meters from pixels. Also, depending on the test results, it might be necessary to convert the source code to another programming language.

Unfortunately, it was not possible to test the performance of the algorithm on the toy vehicle. This happened for a couple of reasons. One is the difficulties with shipment and handover during the pandemic. The other was corrupt software released by the manufacturer of the vehicle kit. The software was fixed too late and the time conditions did not allow me to perform future testing with the vehicle.

I believe that the content of this thesis as well as the source code can be used as a starting point for future development of the software to make the vehicle autonomous. For readers, who are interested in the source code, please contact Department of Applied Mathematics, Faculty of Transportation Sciences, CTU in Prague.

Bibliography

[1] D. H. Ballard and C. M. Brown. *Computer Vision.* Prentice Hall, 1982.

[2] Mark J. Burge Wilhelm Burger. *Digital Image Processing An Algorithmic Introduction Using Java, Second Edition.* Springer, 2016.

[3] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing.* Prentice Hall, 2008.

[4] David A. Forsyth and Jean Ponce. *Computer Vision - A Modern Approach, Second Edition.* Pitman, 2012.

[5] Lukas Stratmann. *Color.* 2017. URL: https://color.lukas-stratmann.com/.

[6] Inc. MathWorks. *Representing color with the Lab color space.* URL: www.mathworks.com/discovery/lab-color.html.

[7] Balkrishan Ahirwal, Mahesh Khadtare, and Rakesh Mehta. "FPGA based system for color space transformation RGB to YIQ and YCbCr". In: *2007 International Conference on Intelligent and Advanced Systems.* 2007, pp. 1345–1349. DOI: 10.1109/ICIAS.2007.4658603.

[8] First Principles of Computer Vision. *Edge Detection Using Laplacian — Edge Detection.* Youtube. 2021. URL: https://www.youtube.com/watch?v=uNP6ZwQ3r6A&t=468s&ab_channel=FirstPrinciplesofComputerVision.

[9] Adrian Kaehler Gary Bradski. *Learning OpenCV.* O'Reilly Media, Inc., 2008.

[10] Stefania Cristina. *A gentle introduction to the Laplacian.* Aug. 2021. URL: https://machinelearningmastery.com/a-gentle-introduction-to-the-laplacian/.

[11] Miloš Sedláček Václav Hlaváč. *Zpracování signálu a obrazu.* FEL ČVUT, 1999.

[12] OpenCV team. *About.* 2022. URL: https://opencv.org/about/.

[13] X. Li M. Wang S. Zheng and X. Qin. "A new image denoising method based on Gaussian filter". In: (2014).

[14] Prathima Guruprasad. "OVERVIEW OF DIFFERENT THRESHOLDING METHODS IN IMAGE PROCESSING". In: (June 2020).

[15] M. Sezgin and Bulent Sankur. "Survey over image thresholding techniques and quantitative performance evaluation". In: *Journal of Electronic Imaging* 13 (Jan. 2004), pp. 146–168. DOI: 10.1117/1.1631315.

[16] Nobuyuki Otsu. "A Threshold Selection Method from Gray-Level Histograms". In: *IEEE Transactions on Systems, Man and Cybernetics* 9 (1979).

[17] TheAILearner. *Perspective Transformation.* 2020. URL: https://theailearner.com/tag/cv2-getperspectivetransform/.

[18] Minhyeok Lee et al. "Robust lane detection via expanded self attention". In: (2022), pp. 533–542.

[19] Mehdi Mahmoodpour et al. "An affordable deep learning based solution to support pick and place robotic tasks". In: (Dec. 2019), pp. 66–75. DOI: 10.22213/2658-3658-2019-66-75.

[20] Nima Khairdoost, Steven Beauchemin, and Michael Bauer. "Road Lane Detection and Classification in Urban and Suburban Areas based on CNNs". In: (Jan. 2021), pp. 450–457. DOI: 10.5220/0010241004500457.

[21] Wajahat Kazmi. *Lane detection: Brief history and the road ahead.* Nov. 2019. URL: https://medium.com/motive-eng/lane-detection-a-quick-review-and-the-way-forward-dd7a43353da.

[22] Keerti Chand Bhupathi and Hasan Ferdowsi. "An Augmented Sliding Window Technique to Improve Detection of Curved Lanes in Autonomous Vehicles". In: *2020 IEEE International Conference on Electro Information Technology (EIT).* 2020, pp. 522–527. DOI: 10.1109/EIT48999.2020.9208278.

[23] Addison Sears-Collins. *real-time-lane-detection.* 2021. URL: https://automaticaddison.com/the-ultimate-guide-to-real-time-lane-detection-using-opencv/.

[24] liwangGT. *CarND-Advanced-Lane-Lines.* URL: https://github.com/liwangGT/CarND-Advanced-Lane-Lines.