České vysoké učení technické v Praze
Fakulta dopravní

Katedra aplikované matematiky
Obor: Inteligentní dopravní systémy

Detekce překážek před autonomním vozidlem zpracováním kamerových dat

Detecting obstacles in front of the autonomous vehicle by processing the images

BAKALÁŘSKÁ PRÁCE

Vypracoval: Šimon Jelínek
Vedoucí práce: Ing. Bohumil Kovář, Ph.D.
Rok: 2023

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):

Šimon Jelínek

Studijní program (obor/specializace) studenta:

bakalářský – ITS – Inteligentní dopravní systémy

Název tématu (česky): **Detekce překážek před autonomním vozidlem zpracováním kamerových dat**

Název tématu (anglicky): Detecting obstacles in front of the autonomous vehicle by processing the images

Zásady pro vypracování

Při zpracování bakalářské práce se řiďte následujícími pokyny:

- Zpracujte přehled metod pro detekci překážek před autonomním vozítkem zpracováním obrazu z kamery. Algoritmy vysvětlete, včetně matematického popisu.
- Zpracujte přehled metod pro prahování šedotónového obrazu. Některé z nich implementujte a vyberte vhodnou metodu pro segmentaci objektu v jízdním pruhu.
- V jazyce Python naprogramujte aplikaci pro detekci překážek v jízdním pruhu před autonomním vozítkem zpracováním obrazu z kamery.
- Ve spolupráci s autorem bakalářské práce J. Zmátlo: "Detekce překážek před autonomním vozidlem zpracováním dat z lidaru" vytvořte robustní systém pro detekci překážek před autonomním vozítkem fúzí dat z lidaru a kamery.
- Funkčnost systému ověřte a vyhodnoťte na reálných datech.

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: **24. října 2022**
(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: **7. srpna 2023**
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

L. S.

.....
RNDr. Magdalena Hykšová, Ph.D.
vedoucí
Ústavu aplikované matematiky

.....
prof. Ing. Ondřej Přibyl, Ph.D.
děkan fakulty

Potvrzuji převzetí zadání bakalářské práce.

.....
Šimon Jelínek
jméno a podpis studenta

V Praze dne ..... 24. října 2022

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 souvisejcích s právem autorským a o změně některých zákonů (autorský zákon)

Podpis: ..... Dne: .....

Poděkování

S úctou si vážím pravidelných schůzek s panem Ing. Bohumilem Kovářem, Ph.D., kde jeho cenné rady a vedení mi výrazně pomohly v porozumění dané problematiky. Děkuji.

Název práce:	**Detekce překážek před autonomním vozidlem zpracováním kamerových dat**
Autor:	Šimon Jelínek
Studijní program:	Technika a technologie v dopravě a spojích
Obor:	Inteligentní dopravní systémy
Druh práce:	Bakalářská
Vedoucí práce:	Ing. Bohumil Kovář, Ph.D.

Abstrakt: Předmětem této bakalářské práce byl vývoj algoritmu pro detekci překážky zpracováním dat z kamery, následná jeho implementace a ověření na reálných datech. První část práce se zaměřuje na teorii, kde se rozebírají témata spjaté se segmentací obrazu a s metody pro detekci překážek. Ve vývoji programu byla aplikovaná jedna z metod pro segmentaci obrazu.

Funkčnost algoritmu byla ověřena na vozítku JetRacer osazené Nvidia Jetson Nano. Pro vývoj algoritmu byl zvolen programovací jazyk Python. K zaručení spolehlivosti programu byla vybraná data přímo z kamery vozítka modelovaná v různých situacích.

Výstupem této práce je funkční kód pro detekci překážek fúzí dat z kamery a lidaru. Algoritmus mimo jiné zpracovává data i k detekci jízdního pruhu, pro takovou úlohu však není koncipovaný. V budoucnu by se sjednocením algoritmů pro detekci jízdních pruhů a pro detekci překážek mohlo mluvit o autonomním systému řízení vozítka.

Title: **Detecting obstacles in front of the autonomous vehicle by processing the images**

Abstract: The subject of this bachelor's thesis was the development of an algorithm for obstacle detection by processing camera data, its subsequent implementation, and verification on real data. The first part of the work focuses on theory, where topics related to image segmentation and obstacle detection methods are discussed. In the development of the program, one of the image segmentation methods was applied.

The functionality of the algorithm was verified on a JetRacer vehicle equipped with an Nvidia Jetson Nano. Python was chosen for the development of the algorithm. To guarantee the reliability of the program, we modeled selected data directly from the vehicle's camera in different situations.

The output of this work is a functional code for obstacle detection by fusion of camera and lidar data. Among other things, the algorithm processes data for lane detection, but it is not designed for such a task. In the future, by unifying algorithms for lane detection and obstacle detection, we could talk about an autonomous vehicle control system.

Contents

1 Overview of image segmentation; thresholding **10**

1.1 An introduction to computer vision . . . 10
1.2 Image segmentation algorithms . . . 11
1.3 Segmentation based on thresholding . . . 11
1.3.1 Simple thresholding . . . 12
1.3.2 Adaptive thresholding . . . 13
1.3.3 Otsu Method . . . 14
1.3.4 Triangle thresholding . . . 15
1.4 Comparison of methods . . . 15

2 Object Detection **17**

2.1 General overview behind the object detection . . . 17
2.2 An Overview of math used in algorithms . . . 18
2.3 List of methods to detect an object . . . 19
2.4 R-CNN . . . 20
2.5 Fast R-CNN . . . 20
2.6 Faster R-CNN . . . 21
2.7 SPP-net . . . 21
2.8 YOLO . . . 22
2.9 SSD . . . 23

3 Proposed method in theory **24**

3.1 Steering system . . . 26

4 Implementation **30**

4.1 JetRacer . . . 30
4.2 Testing video . . . 30
4.3 The source code . . . 32
4.3.1 move.py . . . 32
4.3.2 main.py . . . 32

4.3.3 testHist.py 34
4.3.4 curve.py 36
4.3.5 motion.py 37
4.4 Improvements 37

5 Testing and Evaluation **38**

List of Figures

1.1 Types of simple thresholding 13
1.2 Thresholding triangle 15
1.3 Thresholding results 16

2.1 AI 18
2.2 R-CNN 20
2.3 Fast-R-CNN 21
2.4 SPP 22
2.5 PASCAL VOC 2007 23

3.1 Capture from camera 24
3.2 Defining points on the road 25
3.3 Perspective view 25
3.4 Perspective view from above in grayscale 26
3.5 Different approaches to image segmentation 27
3.6 Principle of the steering 28

4.1 JetRacer 31
4.2 Test track 31

5.1 Principle of the algorithm 39

Listings

3.1 Source of the grayscale image . . . 25
4.1 Commands for terminal . . . 32
4.2 Camera settings . . . 32
4.3 Cropping the image and calculating histograms . . . 33
4.4 Finding peaks; lanes and objects . . . 33
4.5 Calculation of radius . . . 34
4.6 Cropping image; detailed . . . 34
4.7 Histogram; detailed . . . 34
4.8 Finding peaks; detailed . . . 35
4.9 Lane detection; detailed . . . 35
4.10 Object detection; detailed . . . 36
4.11 calculating curvature; detailed . . . 36
4.12 calculating radius of the turn; detailed . . . 36
4.13 Driving function; detailed . . . 37
4.14 Driving function; detailed . . . 37

Introduction

The assignment of this Bachelor's thesis was "Detecting obstacles in front of the autonomous vehicle by processing the images". This project was ahead of its time, and in the near future the world will focus on the issue of autonomous vehicles. The world is moving towards a technical revolution where other common activities will be automated. Apart from the downside of many people losing their jobs, this opens up a wide range of opportunities to contribute to the process.

To be able to understand object detection, it is important to know the steps that precede it such as image acquisition, image preprocessing, and image segmentation. This work discusses the basis of thresholding, which is part of image segmentation, and describes the most common algorithms used for object detection and recognition.

The second part outlines the practical approach to this task in more detail. Describe and explain the individual parts of the code. The object detection algorithm is also documented, and its outputs on real data are located in the last section.

Motivation

Information technology in the field of vehicle control is very exciting, especially with the use of artificial intelligence. The development of AI is moving forward rapidly, so I consider it necessary to have at least a general overview of these things, which may change the future.

List of Contributions

Here are the outputs of this work that can contribute to the creation of the "autonomous vehicle" concept:

- Object detection and lane detection using the histogram analysis
- Polynomial approximation of curve and creation of circular arc
- Basic motions of the vehicle

- The source code for deeper development in the future of this project
- Insights and suggested improvements

Structure of the Thesis

The thesis could be distinguished into two parts; theoretical, which consists of image processing and object detection, and practical part. In image processing, the topic of image segmentation is covered, especially thresholding. The second part introduces the most common algorithms for object detection. In the final part, we describe the final source code in more detail with testing outputs and evaluation.

Chapter 1

Overview of image segmentation; thresholding

1.1 An introduction to computer vision

Computer system can be defined as a collection of hardware and software. Hardware includes physical components such as the motherboard, the central processing unit (CPU), the random access memory (RAM), the monitor, the graphics card, the sound card, or other input/output devices. In terms of software, there are two main types of software: system software and application software. The software of the system deals with the internal functioning of a computer through an operating system. The Application System is a set of commands that the user commands to run [1].

Object detection is a process that allows the computer system to recognize predetermined classes. In our case, classes can be humans, cars, signs, or other objects in the scene. The system is usually built to know what the classes are and where to find them.

This thesis focuses on detecting obstacles by processing images. The image as a virtual representation on the digital screen is made from a matrix with representative numbers. This brings us to the two-dimensional world, where the picture is built with samples or pixels. Each pixel carries information that defines what color it will have. On the basis of this, images can be distinguished into binary, grayscale, or color ones. Binary images are made from pixels that are fully black or white. Grayscale has its own interval of color between fully black and fully white. For pixels in color images, the most common representation is RGB, which refers to the colors red, green, and blue [2].

Detection of luminosity and color change between pixels is an essential process, without which edge detection would not exist. Edge detection can be used in object detection to identify the boundaries of an object. Basically, the algorithm looks for a large color

or luminosity gap between surrounding pixels. There are two ways to solve whether the neighbor pixel is different:

The first is to calculate the difference between each color component of pixel p_1 and p_2 via [2]

$$|p_1 - p_2| + |p_1 - p_2| + |p_1 - p_2| \tag{1.1}$$

The second one deals with color codes. RGB scale starts from 0 to 255 for each color, in binary code from 00000000 to 11111111. Determine the difference is done by shifting and OR-ing the codes.

1.2 Image segmentation algorithms

Image processing has basic steps, which are image acquisition, image pre-processing, image segmentation, and object recognition [3]. Image segmentation can be defined as a procedure, that is, extracting areas from an image. The extraction is performed on the basis of common attributes, such as color, intensity, or other visual properties. Thus, it builds a foundation for further image image analysis including object recognition, object tracking, etc.

There exist various types of algorithm for segmentation. Most common approaches go through:

- Region-Based Segmentation

  Thresholding Segmentation

  Regional Growth Segmentation

- Edge Detection Segmentation

  search-based (Sobel Operator)

  zero-crossing based (Laplacian Operator)

- Segmentation Based on Clustering
- Segmentation Based on Weakly-Supervised Learning in CNN

1.3 Segmentation based on thresholding

One of the most common methods in image segmentation is thresholding. It can be distinguished into local or global approach. The global is returning area of interest and background, as for local is basically the same, but on a larger scale. This method is simple

and fast; for input with an object that has a different color than its background, it can even be accurate [4].

1.3.1 Simple thresholding

The **simple thresholding** looks up at the pixel value and decides whether the value is higher or lower than the set threshold [5].

In binary thresholding it is looking whether the value is higher than the threshold, it changes the pixel to the maximum value and otherwise to the minimum. Decision-making is shown in 1.2.

$$\texttt{dst}(x,y)_{(binary)} = \begin{cases} \texttt{maxval}, & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0, & \text{otherwise} \end{cases} \tag{1.2}$$

For binary inverse the output switched as shown on 1.3.

$$\texttt{dst}(x,y)_{(binary-inverse)} = \begin{cases} 0, & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval}, & \text{otherwise} \end{cases} \tag{1.3}$$

The other method is thresholding trunc, which returns the threshold value if the condition is true; otherwise, it gives the input as an output. Logic is shown in 1.4.

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold}, & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y), & \text{otherwise} \end{cases} \tag{1.4}$$

The last two methods combine approaches from trunc with binary. For successful conditions, it returns the input; otherwise, it gives a zero shown in 1.5. If the output is switched as it was for the binary inverse, we get the threshold to the zero inverse shown in 1.6.

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y), & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0, & \text{otherwise} \end{cases} \tag{1.5}$$

$$\texttt{dst}(x,y) = \begin{cases} 0, & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y), & \text{otherwise} \end{cases} \tag{1.6}$$

A graphic representation of simple thresholding types is shown in figure 1.1.

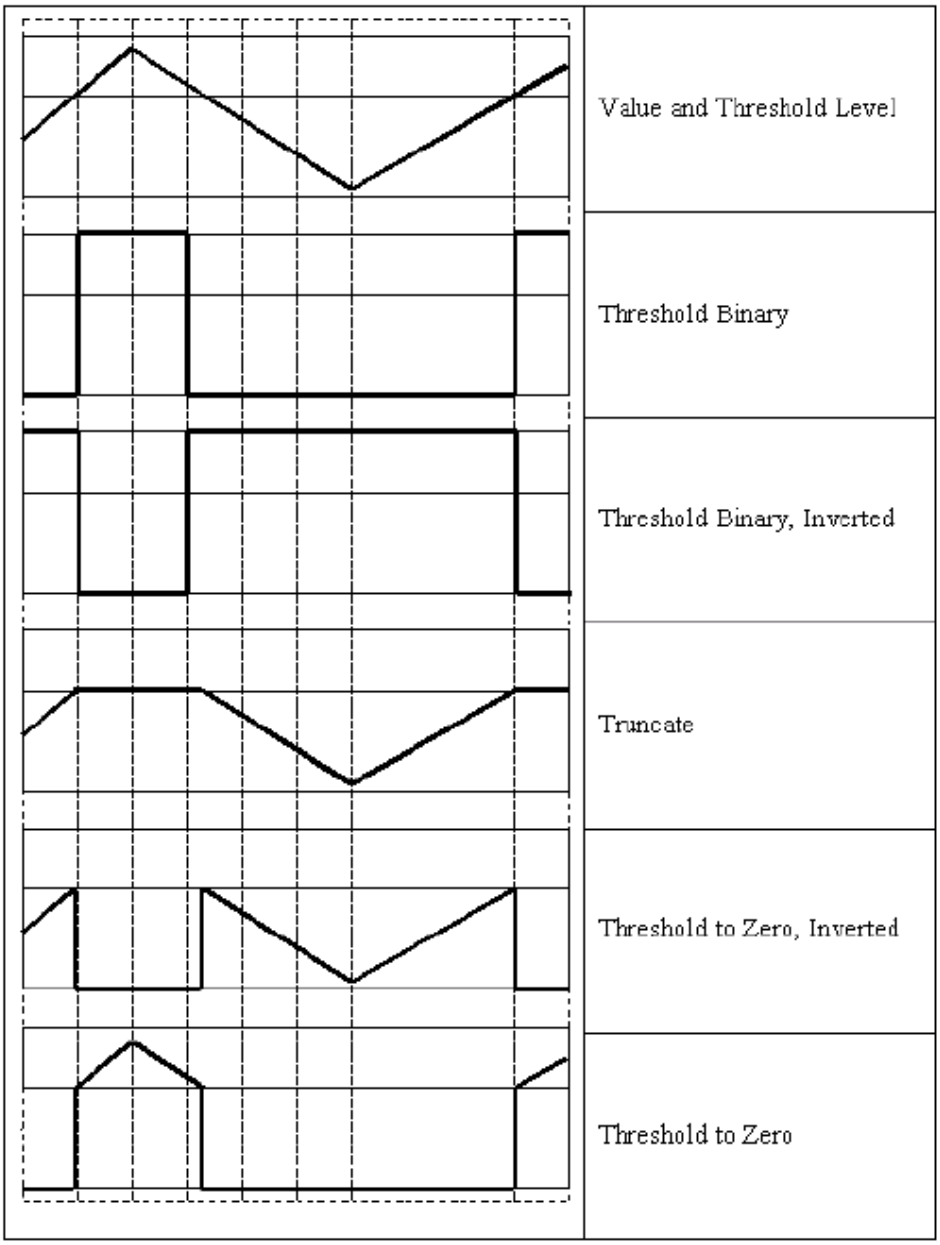


Figure 1.1: Types of simple thresholding [1]

1.3.2 Adaptive thresholding

There is a different way than a simple thresholding, where the threshold value is set globally, which is compared to the input. In the picture there can be areas where the intensity of the pixel is much higher than in other places and that would cause serious damage to the simple thresholding; therefore, the alternative method is to divide the whole picture in smaller regions. In those regions, is calculated local threshold value that the input is compared to.

[1] Source: https://docs.opencv.org/3.4/threshold.png

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxValue}, & \text{if } \texttt{src}(x,y) > T(x,y) \\ 0, & \text{otherwise} \end{cases} \tag{1.7}$$

The logic behind the adaptive thresholding is shown in the equation 1.7. The value *thresh* changed to $T(x, y)$, which is the threshold value calculated in the local region.

The $T(x, y)$ can be calculated in two possible ways. Behind the first one is a simple mean of all the pixels in the region minus the constant c. C is a constant that is subtracted from the mean or weighted sum of the neighborhood pixels [5]. The second is more complex; $T(x, y)$ is calculated as the Gaussian weighted sum minus c.

$$G_i = \alpha \cdot \exp\left(-\frac{(i-(ksize-1)/2)^2}{2\sigma^2}\right) \tag{1.8}$$

In equation 1.8 it shows the procedure for counting the Gaussian coefficient. Where $i \in N, i \in \langle 0; ksize-1)$, $\alpha = \sum G_i$, $ksize = inputValue$, $\sigma = 0.3 \cdot (0.5(ksize-1)-1)+0.8$ [5].

1.3.3 Otsu Method

The main idea behind the Otsu method is that there exist two classes of pixels; in front and in the background. Then calculate the optimal *thresh* for each class, so that the variance between them was minimal [6]. The algorithm calculates the variance for the inside and between classes. The variance in the inside class is shown in equation 1.9. The variance between classes is shown in equation 1.10.

$$\sigma_W^2 = \sum_{k=1}^{M} w_k \sigma_k^2 \tag{1.9}$$

The gray-level probability distributions are w_1, w_2. u_1 and u_2 are the means. The total mean of the gray levels is represented by u_t.

$$\sigma_B^2 = w_1(u_1 - u_T)^2 + w_2(u_2 - u_T)^2 \tag{1.10}$$

The resulting variance is then expressed by 1.11 [7].

$$\sigma_T^2 = \sigma_W^2 + \sigma_B^2 \tag{1.11}$$

1.3.4 Triangle thresholding

There exist one simpler method than otsu, which is called the triangle method. Simple and elegant method that calculates the number of pixels with their brightness. It intersects a straight line between the peek and the minimum, and in half creates a perpendicular determining the bin, which is set to threshold value.

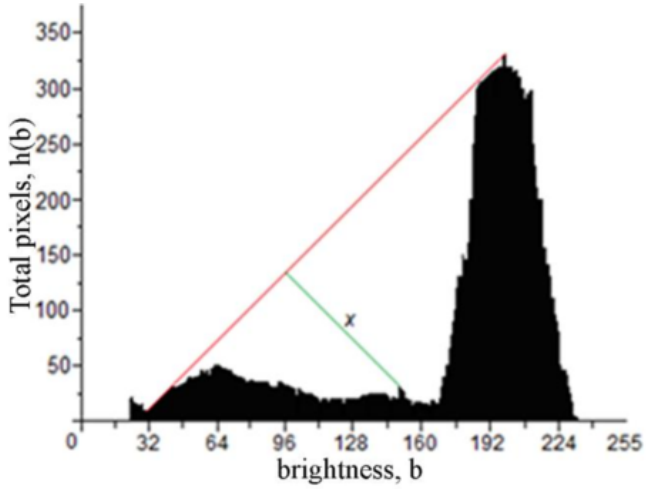


Figure 1.2: Thresholding triangle [2]

1.4 Comparison of methods

Author of this thesis made a simple script in Python to compare methods, which are mentioned above and more, to select the appropriate method to segment the object in the lane.

The original picture is on the left side of the plotted images; above each frame is the name of the implemented method. The otsu method was used in the implementation part; the reasons for this are described in more detail in this thesis.

[2]Source: https://www.researchgate.net/publication/337214579/figure/fig2/AS:824756525858819@1573648687533/Thresholding-by-the-triangle-method.png

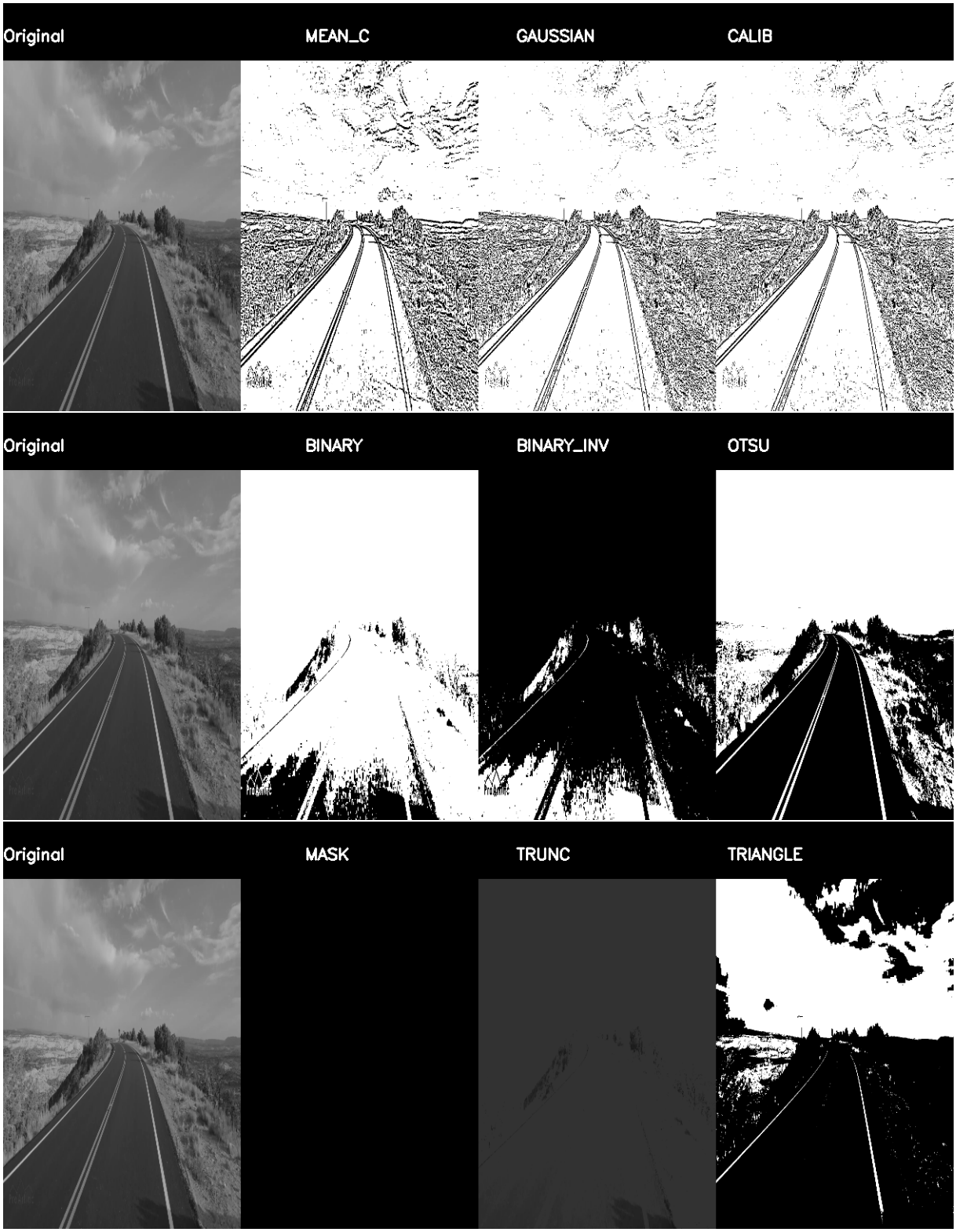


Figure 1.3: Thresholding results

Chapter 2

Object Detection

2.1 General overview behind the object detection

The world is surrounded by many obstacles of various shapes, colors, tastes, smells, and sounds as we describe them using basic senses such as sight, smell, taste, touch, and hearing. The most important might be the sight, where the sensor is the eye. The human eye perceives light with wavelengths of approx. 380 - 720 nm, which is called "visible light".

Color is the visual perception of a combination of light rays of different wavelengths coming from the same place. Color vision is the ability of an organism or device to distinguish objects on the basis of the wavelength of light they emit or reflect. Most mammals have two types of cones (including the dog that can see red and yellow well), but primates have three types of cones (including humans), some birds and insects have 4 or 5 types (also sensitive to UV radiation), butterflies see ultraviolet light but cannot see red, snakes see a wide range of colors from ultraviolet to infrared [8].

Human beings are capable of detecting and recognizing objects of varying distances. To test the speed of human detection and recognition is depending on various factors, however, some studies have investigated the exact duration. The time is around 100 ms [9], which would be 10 fps, if the input takes only the pictures after processing it. In the study "Detecting meaning in RSVP" it is mentioned that the brain processes the image, which the eye saw for only 13 ms, which would make it almost 77 fps [10].

One way to make life easier is to get machines to do our job. In most cases, human error is to blame, the only fault in the entire system that cannot be fixed, but replaced. Artificial intelligence was born with a digital computer, where it was shown to be used to solve difficult tasks. AI essentially learns and evolves like a human being. In some specific areas, AI have already outperformed humans by now. In figure 2.1 are graphically displayed layers of AI.

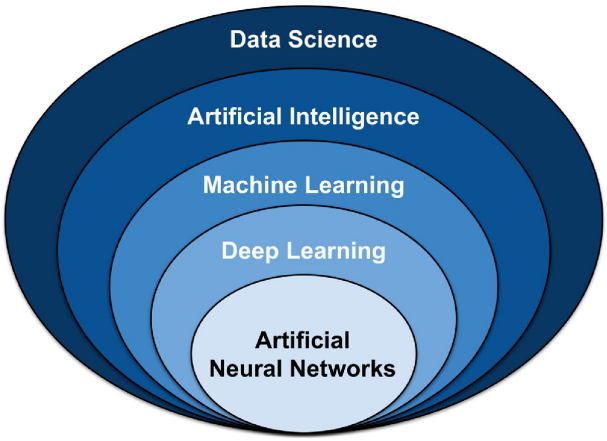


Figure 2.1: AI [1]

Object detection has numerous use cases, for example in autonomous driving, surveillance systems, medical image analysis, e-commerce, robotics, and many others. Algorithms are modified according to the kind of field for which they are used. In many cases, it seeks a balance between speed and accuracy, which are inversely proportional.

2.2 An Overview of math used in algorithms

The algorithms further described are based on deep learning methods with neural networks Fig. 2.1. They are made of multiple layers, which have an element-wise nonlinearity function [11] f with an inside linear combination of input x, a learning parameter w and a basis b [11] displayed on 2.1.

$$y = f(wx + b) \tag{2.1}$$

The function f acquires a sigmoid function or a restricted linear unit [11]. Multiple layers are deeply connected to each other, the various complexity behind it is called architecture. There exist different architectures in deep learning methods, such as stacked autoencoders *(SAEs)*, deep belief networks *(DBNs)*, convolutional neural networks *(CNNs)*, recurrent neural networks *(RNNs)* and generative adversarial networks *(GANs)* [11]. SAEs and DBNs possess a fully connected network, which means all elements are connected between themselves. This has bad influence on performance, therefore CNNs came up with weight sharing strategy, which basically focuses on the area of interest.

$$X_k^{l+1} = \sigma(W_k^l \cdot X^l + b_k^l) \tag{2.2}$$

X_k refers to the "feature map", which is generated by putting "receptive fields" consisting of learnable kernels as W_k added by basis b_k to the nonlinear elemental function $\sigma(\cdot)$ [11].

[1] Source: https://viso.ai/wp-content/uploads/2021/01/data-science-artificial-intelligence-machine-learning-vs-deep-learning-768x554.png

2.3 List of methods to detect an object

The development of object detection can be broken down into two milestones, first before the deep learning method was discovered and then. Deep learning detection is distinguished into one/two-stage technique of detection.

Generally, deep learning-based object detectors extract features from input image or video frames.The object detector solves two subsequent tasks: tasks 1 and 2:

1. Give a number of objects in the frame
2. Classify each object into its category and estimating its size using a boundary box

To simplify the process, these tasks are divided into two phases. Essentially, one-step object detection simply combines two-step objects together in a single step to generate much higher speed at the expense of accuracy. Here are the most common algorithms for each period [12].

- Before deep learning technique

  Viola-Jones Detector

  Histogram of Oriented Gradients (HOG)

  Deformable Parts Model (DPM)

- Two-stage

  Region-based Convolutional Neural Networks (R-CNN)

  Fast R-CNN

  Faster R-CNN

  Mask R-CNN

  Region-based Fully Convolutional Network (R-FCN)

  Spatial Pyramid Pooling (SPP-net)

- One-stage

  Single Shot Detector (SSD)

  YOLO (You Only Look Once)

  YOLOv3

  YOLOR

  YOLOv7

The start of object detection could be dated in 2001 with the Viola Jones Detector, which was mainly focused on face detection with a speed of 15 frames per second on a 384x288 pixel image [13]. The work focuses on algorithms that can possibly be used in the detection of objects in front of an autonomous car; therefore, only some of them are explained in more detail in the following.

2.4 R-CNN

The procedure of R-CNN comes through small boxes that are created after the picture is processed [14]. The algorithm checks if there is an object in the frame using a selective search, which is used to remove those images called regions. The technique is shown in figure number 2.2.

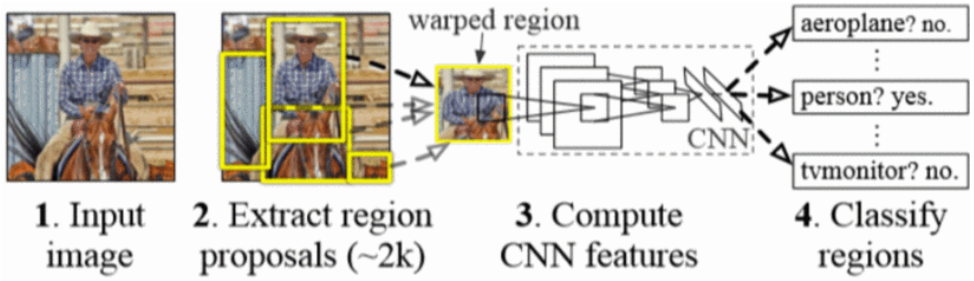


Figure 2.2: R-CNN [2]

To form an object, there generally exist four regions: varying scales, colors, textures, and enclosure. When correctly combined, the final layer should contain only the main frames with recognizable objects inside. These final boxes are reshaped, and each is sent to the convolutional neural network (Conv Net), which is designed to recognize these objects.

R-CNN can be considered a successful method, but the algorithm itself has its own problem, it is slow and demanding in high performance. Up to 2000 regions are extracted from one picture, which means that for n pictures that would be n times more. It takes about 40 to 50 seconds to make the prediction for each new frame [15].

2.5 Fast R-CNN

In short R-CNN decomposes the picture into smaller regions, which are individually examined. Ross Girshick came up with a new model, that sends the whole picture to Conv Net, not individual pieces [14]. This returns regions of interest. Each region comes through a pooling layer to secure a similar size. The last step is connecting the regions to a "Fully

[2]Source: https://viso.ai/wp-content/uploads/2021/03/r-cnn-region-based-convolutional-network-1-1060x346.jpg

connected layer", which is later used for creating bounding boxes around detected objects using the fusion of linear regression and SoftMax (inear classifier).

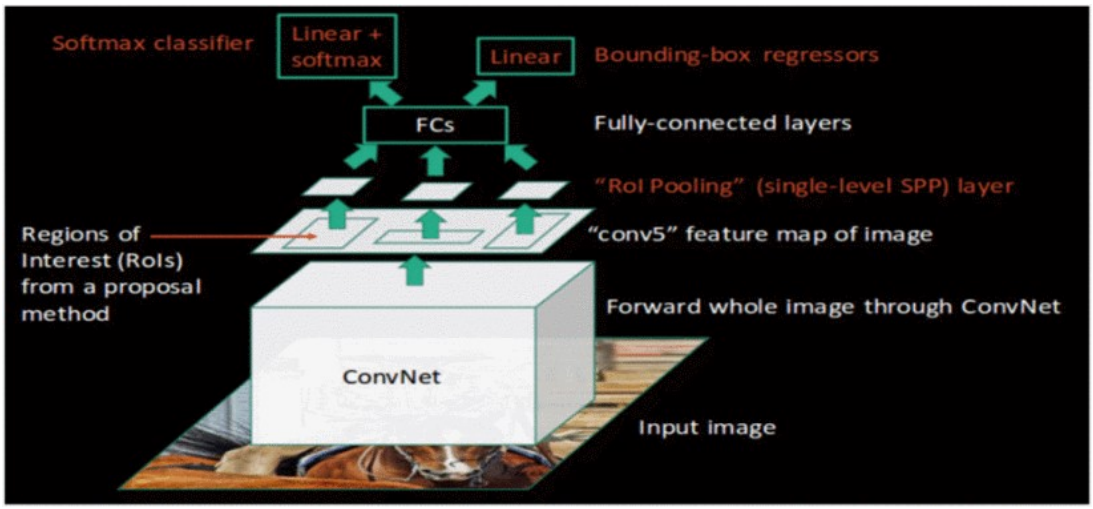


Figure 2.3: Fast-R-CNN [3]

The author states that this method is nine times faster than simple R-CNN [14]. However, the method uses a selective search for the regions of interest, which is not ideal. The time is approximately 2 seconds to identify objects in an image.

2.6 Faster R-CNN

Same year, Ross Girschick released his idea on how to improve the method, a team from Microsoft came up with a slightly better way. They added the feature called RPN (region proposal network). The RPN comes after the image is processed by Conv Net, creating proposals for each box. First it tells what is the chance that there is an object innit, and second it gives a regressionor, which is used for bounding later to fit the object better.

2.7 SPP-net

There was an obstacle in CNN. CNN can be distinguish primarily into convolutional layers and fully connected layers. Fully-connected layers require a fixed size frame. Fixed size can be done by two possible ways:

1. by cropping the frame,
2. by warping the image.

[3]Source: https://ieeexplore.ieee.org/mediastore_new/IEEE/content/media/9993809/9993813/9993862/9993862-fig-12-source-large.gif

[4]Source: https://ieeexplore.ieee.org/mediastore_new/IEEE/content/media/34/7175116/7005506/he1-2389824-large.gif

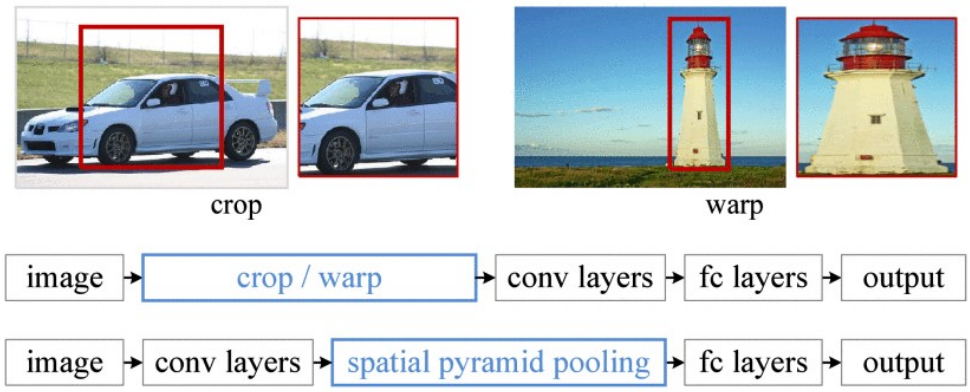


Figure 2.4: SPP [4]

Both ways has its problems, as for cropping there could be missing the whole object in the selected area and with warping comes geometric misinterpretation. Here comes the handy spatial pyramid pooling layer. As mentioned above, the problem appears at the deeper level of the network. The SPP works as a substitute for the last pooling layer; with SPP the input image can be any size, ratio, or even any scale [16]. That brings much higher speed. (R-CNN takes 14.37s per image for convolutions, while our one-scale version takes only 0.053s per image) [16].

2.8 YOLO

For real-time detection it is necessary to deal with the latency of the detection itself; therefore, speed and accuracy are required. Concisely, the two–stage algorithm was rebuilt into one-stage. In year 2016 version of a different approach was represented by Joseph Redmon called "You Only Look Once: Unified, Real-Time Object Detection".

YOLO divides the input into smaller cells, where it returns the chance of finding an object by giving multiple frames. In a R-CNN selective search, the number of multiple bounding boxes per image is around 2000 besides by YOLO this number is limited to 98 [17]. Not only YOLO is fixing maximal amount of boxes, but even their spatial location to prevent finding multiples in the same area, which rapidly decrease the final time.

Several of the algorithms mentioned here are shown in the figure 2.5. The table is divided into 3 columns; the Train data used for the test, precision of the single algorithm expressed by mAP (mean Average Precision) and FPS, which stands for frames per second. YOLO compared to Fast R-CNN was 90 times faster for the price of only 6.6 mAP.

[5]Source: https://ieeexplore.ieee.org/mediastore_new/IEEE/content/media/7776647/7780329/7780460/7780460-table-1-source-large.gif

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [30]	2007	16.0	100
30Hz DPM [30]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [37]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[27]	2007+2012	73.2	7
Faster R-CNN ZF [27]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Figure 2.5: PASCAL VOC 2007 [5]

2.9 SSD

In 2016 another one-stage method to detect an object was presented. SSD stands for Single Shot (MultiBox) Detector. In particular, SSD uses fixed bounding boxes that predict the object's category and also give an offset to those boxes to fit the object better.

The image goes through CNN, which returns the feature map. The feature map is modified to different scales, according to which they are added with different separate predictions, and are processed by many small convolutional filters [18].

Overall SSD performs better with detecting larger objects rather than smaller once, because the smaller once don't even care information in upper layers of Conv Net. The larger input size could be used to improve the performance in recognizing slighter objects.

In figure 2.5 have been shown the results of different algorithms. The results on Pascal VOC2007 showed that SSD got 74.3 mAP with 46 FPS, which outperforms YOLO in speed and accuracy [18].

Chapter 3

Proposed method in theory

The assignment was to evaluate whether there is an obstacle in front of the car or not. As an input is a capture from a camera, an example is shown in Figure 3.1.

Figure 3.1: Capture from camera [1]

To simplify the process of finding the obstacle, the frame is transformed into a perspective view. To achieve that, two functions built in OpenCV called *getPerspectiveTransform* and *warpPerspective* are used. First, we need to specify four points in the picture 3.2, which will be future corners of the new frame. The perspective transform function uses the coordinates of the prespecified points and transforms them into the matrix shown in 3.1.

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{3.1}$$

$$\texttt{dst}(i) = (x'_i, y'_i), \quad \texttt{src}(i) = (x_i, y_i), \quad i = 0, 1, 2, 3$$

[1] Source: https://trebicobcanum.net/wp-content/uploads/2017/11/23283504_1490663254355211_1710127173_n.jpg

Figure 3.2: Defining points on the road

The space between the selected points is now warped by using the second function shown in 3.2. M is output from 3.1 3×3 transformed matrix [5].

$$\texttt{dst}(x, y) = \texttt{src}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right) \tag{3.2}$$

The new perspective transformed frame is shown in figure 3.3.

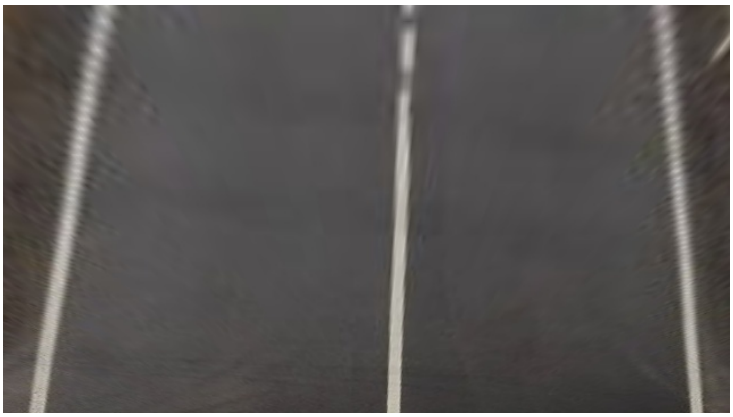


Figure 3.3: Perspective view from above

The image is converted to grayscale using the openCV predefined function for it called *cvtColor*. The source of the output is an array of values ranging from 0 to 255 depending on the intensity of the pixel at a specific location shown below.

```
[[218 218 218 ... 130 128 128]
 [220 220 220 ... 129 128 127]
 [222 222 222 ... 128 127 126]
 ...
 [ 70  71  72 ...  37  37  37]
 [ 68  72  76 ...  37  37  37]
 [ 68  77  86 ...  37  37  37]]

```

Listing 3.1: Source of the grayscale image

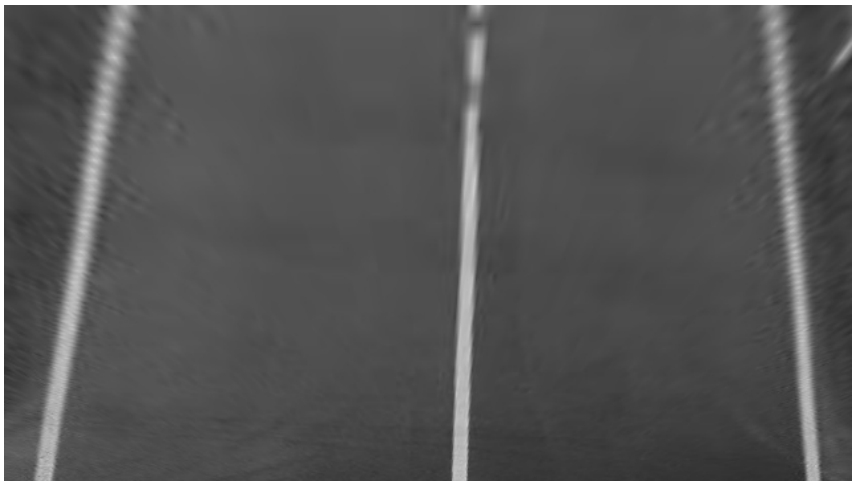


Figure 3.4: Perspective view from above in grayscale

The core idea is in the array of those values. The values are sorted from left to right and from top to bottom. If the input had 640×480 resolution, that would mean that the output array would have 640 elements on the $x-$axis and 480 on the $y-$axis.

The idea is to sum the columns with the same distance x. On the whole picture, it would not make sense, but if the picture is divided into thinner layers, it has a more informative value. Cropping the image or creating a mask in the image creates a layer that can be inspected for any obstacle. The idea of cropping the image was chosen, although it may appear that creating the mask could lead to better speed performance.

The whole picture is cropped from top to bottom in thin layers that were set to 30 pixels each. In each layer, the algorithm finds peaks between the two lanes. The detection of those lane is described in more detail in 4.3.3. A point with a value of half the intensity of the detected lane is considered an object. The principle is shown in 5.1.

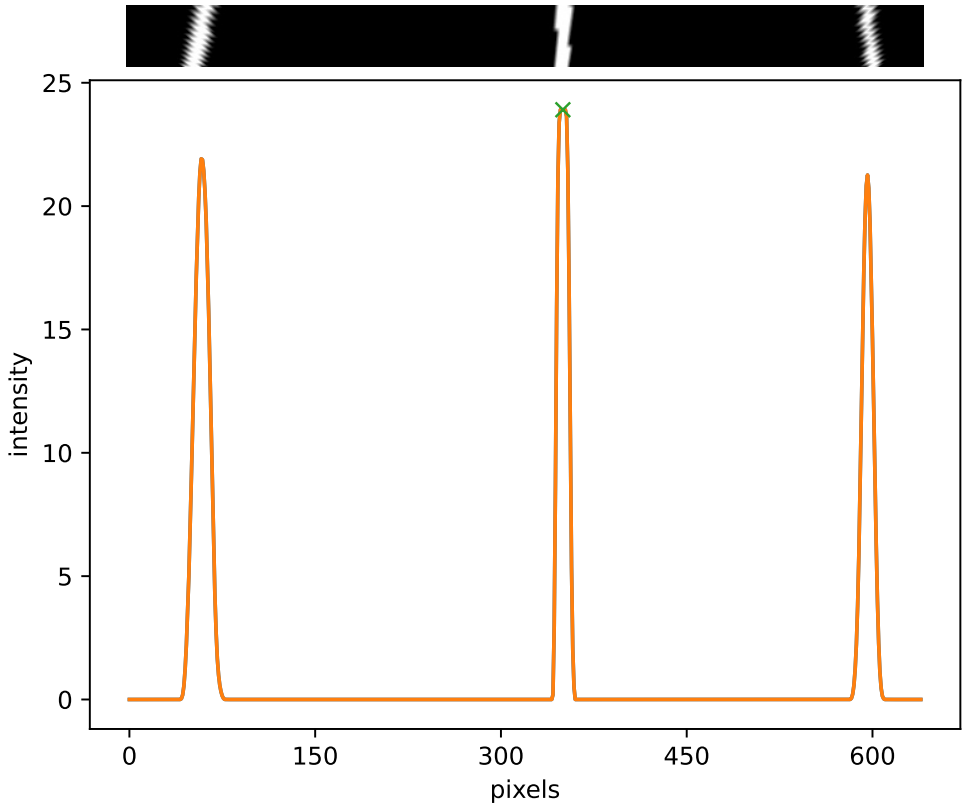
In the next Figures 3.5 are shown cropped images with its calculated histograms. There are shown two aproaches; one is converted to otsu and the second one is left in grayscale.

The conversion into otsu and gray has its pros and cons. Otsu method is really good for lane detection, but in terms of object detection, it rarely detects any change. As for the grayscale, it is more sensitive to changes in pixel intensity; therefore, it is much better at detecting obstacles.

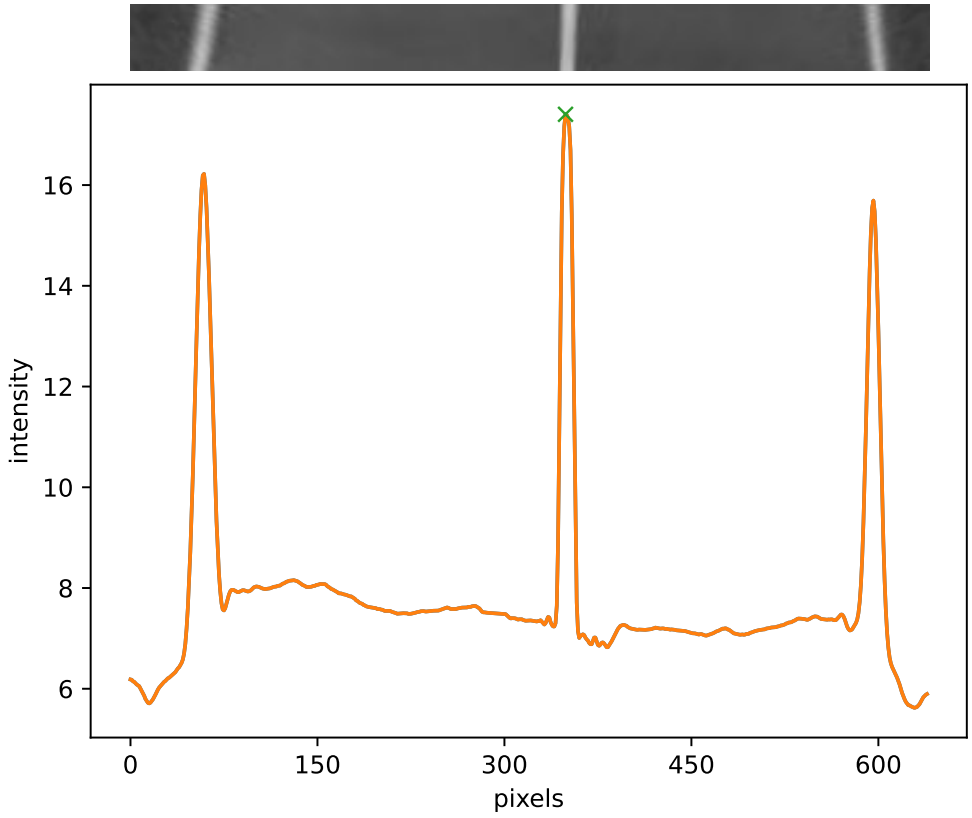
3.1 Steering system

The provisional solution for driving the JetRacer consists of these steps:

- collecting coordinates of midpoints
- calculating polynomial curve



(a) Cropped image otsu



(b) Cropped image grayscale

Figure 3.5: Different approaches to image segmentation

- selecting 2 points (first and the last)
- fitting the points with a circle
- recalculating radius of the circle to steering scale

In section with lane detection is calculated point between the two lanes i.e midpoint for each cropped image. Knowing the coordinates of these points, a *numpy* function called *polyfit* is used, which returns a vector of coefficients p that minimizes the squared error shown in equation 3.3.

$$\sum_{i=1}^{n} |p(x_i) - y_i|^2 \tag{3.3}$$

Then are selected two points; first is right in the center of the vehicle and the second is the last point of the new curve created. With a trigonometric equation using these two points, the radius of possible trajectory is calculated. The principle is shown in figure 3.6. The green dots are graphically represented as the midpoints, the blue curve as the polynomial curve, and the pink color is for the circle (ideal trajectory) with the center of S_k.

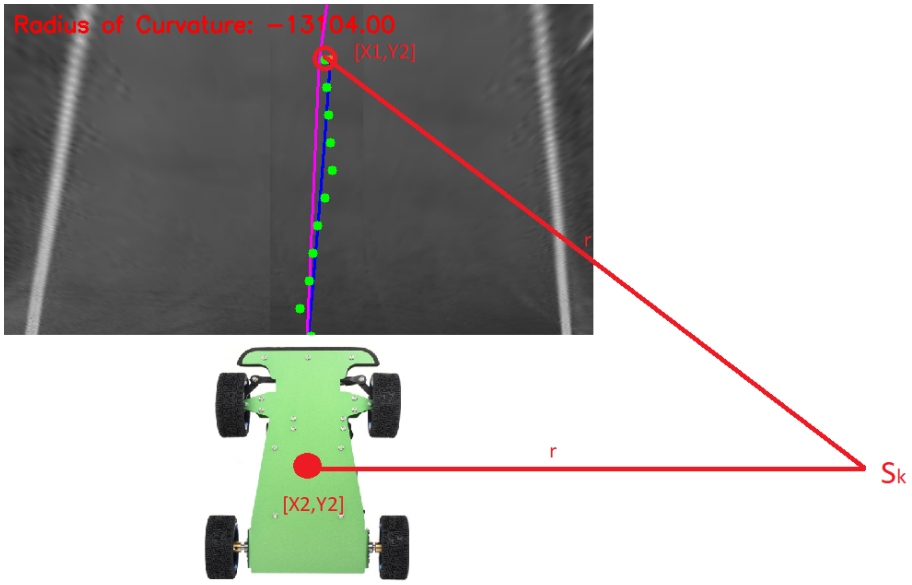


Figure 3.6: Principle of the steering

By default, the steering ratio is set from -1 to 1. The maximum turn of 1 was set to drive on a trajectory of a nonideal circle with a radius of 50 cm. Calculations of the minimum radius of turn theoretically resulted in 60 cm, but for practical purposes, the 50 cm radius was chosen instead.

The length of perspective view was measured, and thus the conversion between pixels and

centimeters was found as shown in 3.4. The height of image is set to 480, 55 cm was the length of the perspective window and 50 cm as for minimal radius of the turn.

$$\text{turn} = -\frac{50}{\text{radius} \cdot 55/480} \tag{3.4}$$

Chapter 4

Implementation

This part of the thesis describes the steps of the practical part. It contains the creation of the testing videos, the adjustment of physical modifications to the vehicle, the setting of basic motions of the model, and the writing of the final code in Python. The development of the code took place under constant testing and modification.

4.1 JetRacer

The task of self-driving vehicle is for simplification applied on small rc model, called JetRacer[1]. It was lent to the project purposes by CTU, the Faculty of Transport.

In the beginning, it was necessary to purchase an SD card adapter to USB. The SD card needed to be flashed and uploaded by JetRacer ROS Image[2], which is basically Ubuntu extended by ROS functions. After setting up the connection, the basic motions were tested in Jupiter Lab on *localhost*:8888. Few things had to be adjusted, for example fixing the center steering on the servo.

4.2 Testing video

In the next step, testing videos had to be created to make some progress in the project. In cooperation with Jan Zmátlo a test track in AutoCad was created, which was then printed in one of the CTU's buildings.

Using a simple script written in Python, a simple script was executed to capture the video from the camera. Many scenarios were modeled; car driving straight through the

[1]Source: https://www.waveshare.com/product/jetracer-ai-kit.htm
[2]Source: https://drive.google.com/file/d/16OBLRNlrZaSkhVcC4xJ6VugtChmZw1_B/view
[3]Source: https://www.waveshare.net/photo/development-board/JetRacer-2GB-AI-Kit/JetRacer-2GB-AI-Kit-3.jpg

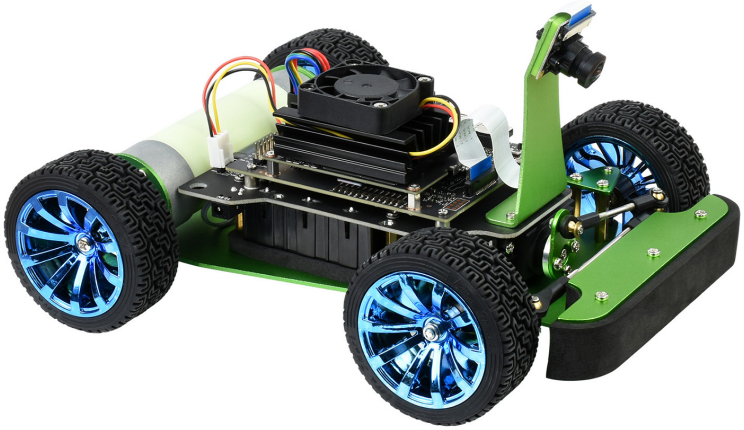


Figure 4.1: JetRacer [3]

track, car driving side to side, car driving over a black rectangle, and car approaching an obstacle.

The task was to detect an obstacle. Using only camera raises the problem of whether the stimulus detection is an object or just a splodge on the track. To simulate this kind of problem, paper with black rectangle innit was used. The test track is shown in figure 4.2.

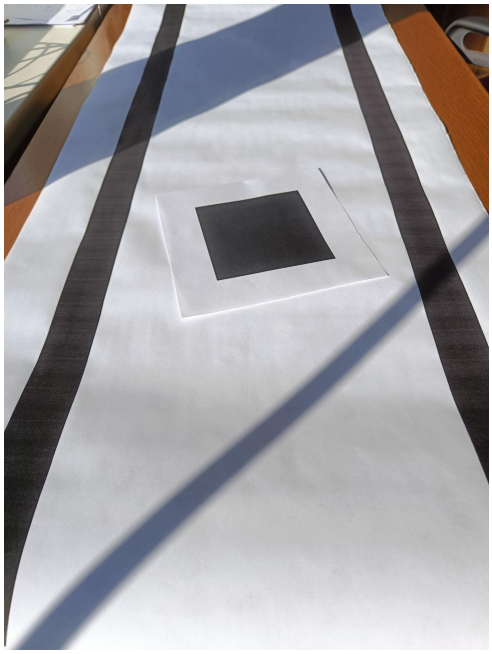


Figure 4.2: Test track

4.3 The source code

To build up the following code, one needs to install some libraries. It can cause some syntax errors, if the final script is executed in the windows environment; therefore it is highly recommended to run it on Linux platform.

```
        pip install opencv-python
        pip install --upgrade numpy
        pip install --upgrade scipy
        pip install nose
        pip install --upgrade imutils

```

Listing 4.1: Commands for terminal

The implementation of the algorithm consists of five files; *main.py*, *testHist.py*, *motion.py*, *curve.py*, and *move.py*. Development started on windows with testing on an image that contained only main.py and testHist.py, over time, new problems appeared that needed new solutions.

4.3.1 move.py

The main file that heads the other files is move.py. In this file, the footage from camera is captured and sent to main.py. Camera is defined with the following settings:

```
            def __gstreamer_pipeline(
                    camera_id,
                    capture_width=640,
                    capture_height=480,
                    display_width=640,
                    display_height=480,
                    framerate=30,
                    flip_method=0,
                ):

```

Listing 4.2: Camera settings

The program is calling the function `obj.video_test` in case there appears a problem, there is one function called mo.kill('bye'), which stops the motors and sets the steering in the middle. After that, the camera is turned off.

4.3.2 main.py

The main.py can be broken down into several areas. At the beginning of the program the image is transformed into perspective view to better analyze the environment in front of the vehicle.

The math behind the perspective transform is shown in 3.1. The image is converted to grayscale and then to Otsu binarization. The Otsu binarization has very good performance

in detecting lanes; however, when testing on a track, a problem with light from the Sun, which caused blindness.

The binary image is cropped into smaller regions (rectangles throughout the width of the image), from which local histograms are calculated. In an array `midpoints_y` the y distances are stored.

```
        cropped_images,midpoints_y = th.cropping_img(result_input)
         histograms = []

         for cropping_img in cropped_images:
             #img_cropping = cv2.rotate(result_input, cv2.ROTATE_90_CLOCKWISE)
             histVar = np.array(th.calculate_histogram(cropping_img), dtype=object)
             histograms.append(histVar)

```

Listing 4.3: Cropping the image and calculating histograms

In calculated histograms are found a peaks and objects. Two peaks in curtain distance are considered as lanes, objects are peaks that are found between the two lanes. Between two lanes are calculated midpoints for each cropped image, which are stored in `midpoints_x`. Along with `midpoints_y` it is possible to later create a curvature of the track in front of the vehicle.

```
    #peaks in histogram + adding objects
    objects = np.array([])
    midpoints_x = np.array([])
    for i in range(len(histograms)-1,-1,-1):
        temp = histograms[i]
        inv_data = np.invert(temp[4])
        histograms[i]=np.append(temp,inv_data) #add inverted data to histogram array [7]

        peaks = th.find_peak(inv_data,temp[6]) #invert for minimum
        histograms[i]=np.append(temp, peaks) #add peaks to histogram array [8]

        middle,last_lane, alert = th.lanes_detection(np.asanyarray(peaks[0]),(last_lane))
        objects = np.append(objects, th.object_detection(middle,inv_data,temp[6],
    last_lane)).astype(int)
        if alert != 1:
            midpoints_x = np.append(midpoints_x, middle).astype(int)
        else:
            midpoints_y = np.delete(midpoints_y, i)

```

Listing 4.4: Finding peaks; lanes and objects

If an array called "objects" is empty, the algorithm continues unchanged. In case there is an object, it is called a function made by the collaborator Jan Zmátlo that uses Lidar to evaluate the situation. If Lidar sees an object, then it calls the mo.kill('bye') function, which turns off the engines and sets the steering in the middle. In case there are no objects detected by Lidar, the elements in array 'objects' should be erased.

After validation, there is no object; the midpoints are sent to create a curvature. There is the condition that in case there are no midpoints, which can happen, the engines turn down. The curvature is sent to the steering function.

```
        # making a curve from data + getting raidus of curve
        if midpoints_x.size != 0 and midpoints_y.size != 0:
         radius = th.show_curve(birdseye,midpoints_x,midpoints_y)
         mo.steering_20(radius)
        else:
            mo.kill(objects)

```

Listing 4.5: Calculation of radius

The while cycle ends with function `mo.drive(objects)`, which may appear strange, but in case Lidar did not erase the elements, there is a second chance.

4.3.3 testHist.py

In this file is are many functions called from main.py, the main are functions for cropping the image, finding peaks, finding lanes or objects, and then plotting the results.

```
    def cropping_img(img):
        image_height = img.shape[0] image_width = img.shape[1] images = [] distance = []
    j = 30 cutout = image_height//3
        #cropping image from top to bottom into rows of 30 pixels for i in range(0,
    image_height-cutout, 30):
            distance = np.append(distance, image_height-j).astype(int)
            cropped = img[image_height-j-60:image_height-j, 0:image_width]
            images.append(cropped)
            j += 30 return np.array(images), np.array(distance)

```

Listing 4.6: Cropping image; detailed

In this function the input image is cropped into smaller rectangles with width of the image, height of thirty pixels, and the distances according to which they were cut are stored. From the top, a third of the height is cut out. The reason is that further away from the vehicle, when the lanes point to the side, an obstacle would be detected. Third, the provisionally constant for turns that do not acquire ninety degrees works just fine.

The histogram for every region is then calculated. The histogram adds values of intensity of each column, the columns are separated by pixels. Then the largest and smallest pixels are found, which, by arithmetic mean, will give the threshold value for finding an object.

```
def calculate_histogram(img):
    image_height = img.shape[0]
    image_width = img.shape[1]
    pixel_sums_x = [sum(row) for row in img]
    pixel_avgs_x = [s / image_height for s in pixel_sums_x] #horizonatl sum

    pixel_sums_y = [sum(col) for col in zip(*img)]
```

```
    pixel_avgs_y = [s / image_width for s in pixel_sums_y] #vertical sum

    midline = -((max(pixel_sums_y))+(min(pixel_sums_y)))//2
    return  image_height, image_width , pixel_sums_x, pixel_avgs_x,pixel_sums_y,
    pixel_avgs_y, midline

```

Listing 4.7: Histogram; detailed

There is a built-in function to find peaks. To detect lanes, an attribute called "distance " is used; set to 500 pixels, which is 22,2 cm.

```
def calculate_histogram(img):
def find_peak(input,mid_height):
    input = np.array(input)
    peaks, _ = find_peaks(input, height=mid_height, distance=500)
    return peaks,input,mid_height

```

Listing 4.8: Finding peaks; detailed

It can accrue, that is only one peak found, therefore the other lane added up based on last lane position. If it lost both sides, than it return variable called warning.

```
def lanes_detection(lane_peaks, last_lane):
    #space bewteen two peeks, handle if one dissappears to remember on which side it was
    if(len(lane_peaks)<1):
       print("Lost lanes!")
       #last_lane = [24,542]
       space_between=  (last_lane[0]+last_lane[1])//2
       warning = 1
       return space_between, last_lane, warning

    if len(lane_peaks)<2:
     if (lane_peaks[0] > 600//2): #if (left) lane is (lost), than cant be greater than
    middle
        space_between = (lane_peaks[0] + last_lane[0])//2 #left
        last_lane = [last_lane [0], lane_peaks[0] ]
        #warning = 1
     else:
        space_between = (lane_peaks[0] + last_lane[1])//2 #right
        last_lane = [lane_peaks[0] , last_lane [1]]
        #warning = 1
    else:
        space_between=  (lane_peaks[0]+lane_peaks[1])//2 # between two lanes
        last_lane = lane_peaks

    warning = 0

    return space_between, last_lane, warning


```

Listing 4.9: Lane detection; detailed

Object detection looks for peaks between the two lanes using the set threshold value of the

previous function. For reduction of false object announcements, the detection is shifted by a few pixels to the middle because around lanes could appear inaccuracies.

```
def object_detection(mid, data, midheight,edges):
    mid_dist=(edges[1]-edges[0])//2-95
    pot_prob, _ = find_peaks(data,height=midheight)
    object_idx=np.asanyarray(np.where(np.logical_and(pot_prob>=(mid-mid_dist), pot_prob
    <=(mid+mid_dist))))[0]

    if(len(object_idx)>0):
        pot_prob = pot_prob[object_idx]
        return pot_prob
    return [] #cannot return 'NoNe'

```

Listing 4.10: Object detection; detailed

In the end of the file are plotting functions, which are not important to understanding the principle of the algorithm.

4.3.4 curve.py

The curvature of the turn and the calculation of the radius that needs to be set are explored in the cuve.py file. For curvature estimation, the built-in function $np.polyfit(x, y, 2)$ is used, that is, calculating the second order polynomial, then to get the directive, we use $np.polyder(coefficients, 2)$ to derivative it. The radius of curvature is an inverted value of it.

```
    coefficients = np.polyfit(x, y, 2)
    derivative = np.polyder(coefficients, 2) # Calculate the second derivative of the
    polynomial
    curve_x = np.linspace(x.min(), x.max(), 1000)
    curve_y = np.polyval(coefficients, curve_x)
    curve_points = np.column_stack((curve_y, curve_x)).astype(np.int32)
    radius = (1 / derivative[0])

```

Listing 4.11: calculating curvature; detailed

To set the appropriate radius of turn through which the vehicle must go, two points were selected. First, the middle of the car, and second, the last point in the curve (made by polyfit). The center of the circle on which the car drives is on sides, is set on side, on which is up to the operator in front of curvature.

```
    # Coordinates of the center of the circle (y)
    x1 = endpoint[0]
    y1 = endpoint[1]

    x2 = startpoint[0]
    y2 = startpoint[1]

    height = abs(y1 - y2)
```

```
    width = abs(x1 - x2)

    temp = math.sqrt(width**2 + height**2)
    alpha = math.atan2(height, width) # radians
    x_circle = round((temp/2)/math.cos(alpha))
    print(x_circle)
    retio = (50/(55/480))/x_circle
    #retio = (50/(22.5/500))/x_circle
    print(f"Pomer zatoceni {retio}")
    r_turn =  x_circle *(abs(radius)/radius) #to decide which side should it be
    center_x =width//2 - r_turn
    center_y =height + start_offset #30 cm from middle of the car

```

Listing 4.12: calculating radius of the turn; detailed

4.3.5 motion.py

The car throttle and steering are running in motion.py file.

```
        def drive (warnings):
        if(len(warnings) > 0):
          kill('bye')
        else:
          car.throttle_gain = 0.7
          car.throttle = -0.5

```

Listing 4.13: Driving function; detailed

If the car doesn't have an incentive to stop, it keeps going.

In the steering function, the radius of the turn to be made is adjusted. Steering servos are given values from -1 to 1 and the maximum turning radius is 50 cm. Radius value is being recalculated to cm, according to 55 $cm = 480\ px$.

```
    def steering_20(radius):
        turn = -50/(radius*(55/480)) #recalculatign pixels to cm .. 100cm Diameter of
    steering 1.0 (30 is distance between 2 points) 55cm = 480 px
        print(turn)
        car.steering = turn

```

Listing 4.14: Driving function; detailed

4.4 Improvements

The core idea behind cropping the image and finding histogram peaks is interesting and reliable. The image is being cropped just from one side and calculated from a horizontal perspective, which brings up information on the $x-$axis. The y value is stored in the cropping variable, but in the future, to be able to analyze the shape of the object, it needs to be cropped from both sides.

Chapter 5

Testing and Evaluation

Ongoing testing took place on the fourth floor in CTU's buildings located in Florenc. For testing purposes a track was printed on the plotter, black square paper was used to simulate potential shadows on the road, and books there were used for obstacles. The test track is shown in the picture 4.2.

The development of the source code was continually tested in the vehicle. For faster development, a few more test tracks were created and tested in the author's home. The tracks were made by adhesive tape, to simulate the track that was primarily important, the lanes were made in the same distance.

In figure 5.1 the principle of object detection is shown. On the right side there is a window of an image in perspective view. The green rectangle is the middle of the lane and also the height of the cropping image, which explores the whole width. On the left side are plotted histograms, which on the $x-$axis have pixels and on y the sum of intensity. Green crosses indicate lane detection, and red circles are possible objects.

Detection of objects is peaks inside the lanes, which achieve the threshold value 5.1. The threshold value is the arithmetic mean of the maximum and minimum intensity of the cropped area.

$$\texttt{objects}[] = \begin{cases} objects[x_i], & \text{if } \texttt{peaks}(\sum y_i) > \texttt{thresh} \\ [\,], & \text{otherwise} \end{cases} \tag{5.1}$$

In the picture 5.2b are displayed the points that were detected between the lanes for each cropping frame, the blue and purple lane. The blue lane is the approximation of those points, and the purple lane is a graphical representation of the radius that must be set for steering.

Along with testing, problems appeared that needed to be dealt with. For this project,

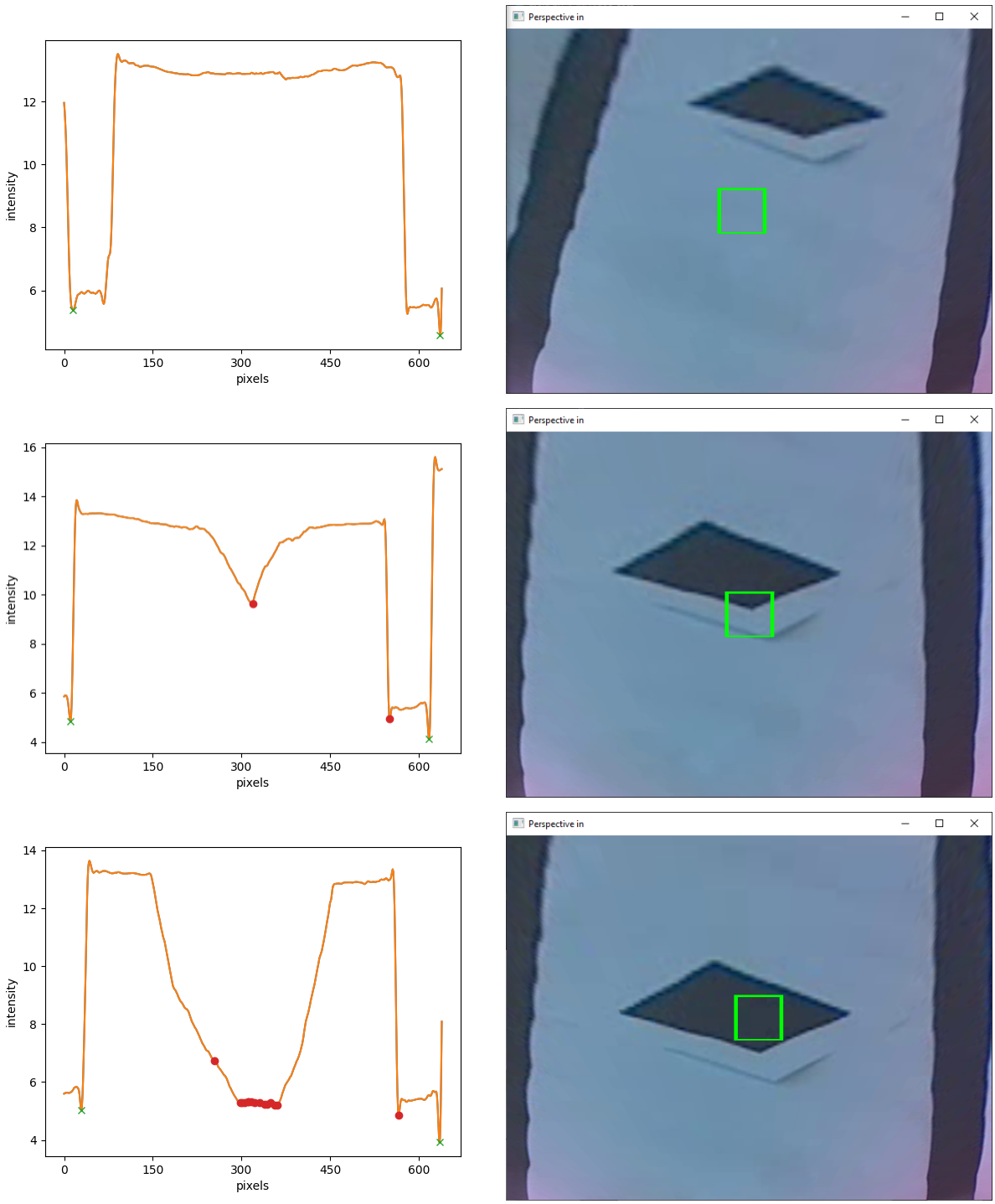


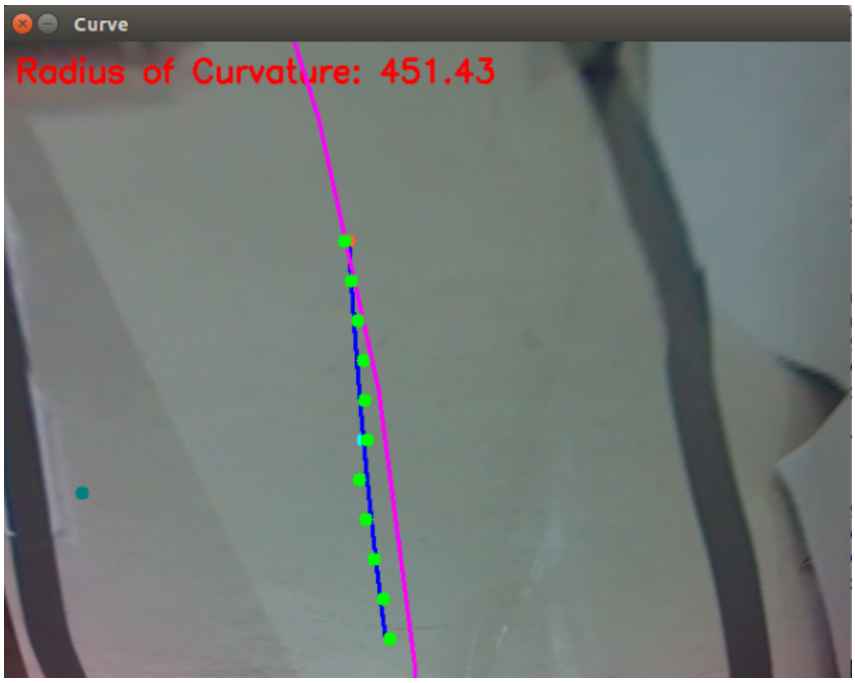
Figure 5.1: Principle of the algorithm

two vehicles were used, one with a camera and lidar and one without the lidar sensor.

Each vehicle has slightly different camera placement, which was almost unrecognizable. The problem arose with reshaping the image into bird's perspective, where the output was far off. The camera calibration had to be done separately on each of the devices.

Other problems were with camera resolution, which could be set higher, but that appeared

(a) Test track windowsill



(b) Curve with circle

to be an unsolvable problem for the second vehicle with Lidar, where the program crashed afterward.

The lidar's power supply is very demanding; therefore, the lidar can not run for the whole task and only can be called if there is an impulse.

Conclusion and Future Work

The task of this bachelor's thesis was to detect an obstacle in front of a moving vehicle. This task was successfully completed. The biggest struggle was with the implementation of steering inside the algorithm, which originally should provide only obstacle verification. The predecessor of the project built the algorithm with sliding widows in parallel, which would help to correct for imperfections that occurred during testing.

For a robust autonomous system, it would be great if the following students combined algorithms and corrected any deficiencies.

Bibliography

[1] The Editors of Encyclopaedia Britannica. *Software*. 2023. URL: https://www.britannica.com/technology/software.

[2] Borko Furht. *Digital Image Processing: Practical Approach*. Springer International Publishing AG, 2018. URL: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=5507921.

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

[4] Yuheng Song and Hao Yan. "Image Segmentation Techniques Overview". In: (2017), pp. 103–107. DOI: 10.1109/AMS.2017.24.

[5] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

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

[7] Dongju Liu and Jian Yu. "Otsu Method and K-means". In: *2009 Ninth International Conference on Hybrid Intelligent Systems*. Vol. 1. 2009, pp. 344–349. DOI: 10.1109/HIS.2009.74.

[8] Novotný. *Základy aplikované počítačové grafiky*. Fakulta dopravní ČVUT, 2023.

[9] Rufin VanRullen and Simon J. Thorpe. "The Time Course of Visual Processing: From Early Perception to Decision-Making". In: *Journal of Cognitive Neuroscience* 13.4 (2001), pp. 454–461. DOI: 10.1162/08989290152001880.

[10] Carl Erick Hagmann Mary C. Potter Brad Wyble and Emily S. McCourt. "Detecting meaning in RSVP at 13 ms per picture". In: *Attention, Perception, & Psychophysics* (2014).

[11] Xiaoyue Jiang. *Deep Learning in Object Detection and Recognition*. Springer Singapore Pte. Limited, 2019. URL: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=5982457.

[12] Gaudenz Boesch. *Object Detection in 2023*. 2023. URL: https://viso.ai/deep-learning/object-detection/#:~:text=Popular.

[13] Michael J. Jones Paul Viola. *Robust Real-Time Face Detection*. 2004. URL: https://www.face-rec.org/algorithms/boosting-ensemble/16981346.pdf.

[14] O. Hmidani and E. M. Ismaili Alaoui. "A comprehensive survey of the R-CNN family for object detection". In: (2022).

[15] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. 2018. URL: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

[16] Kaiming He et al. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.9 (2015), pp. 1904–1916. DOI: 10.1109/TPAMI.2015.2389824.

[17] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: (2016), pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[18] Wei Liu et al. *SSD: Single Shot MultiBox Detector*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 21–37.