Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

An Adaptive Computer-Vision-Based Method to Calculate Vehicle-Motion Parameters

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Markus Chmelar
Matrikelnummer 0826122

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Mitwirkung: Univ.Ass. Dipl.-Ing. Oliver Höftberger

Wien, 7.11.2014

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Adaptive Computer-Vision-Based Method to Calculate Vehicle-Motion Parameters

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Markus Chmelar
Registration Number 0826122

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Assistance: Univ.Ass. Dipl.-Ing. Oliver Höftberger

Vienna, 7.11.2014

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Markus Chmelar
Beethovengasse 36, 3040 Neulengbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Kurzfassung

Bei der Entwicklung von selbstfahrenden, autonomen Fahrzeugen kommen der Zuverlässigkeit und Genauigkeit der Sensoren eines Fahrzeuges immer größere Bedeutung zu. Einer der wichtigsten Parameter ist die Geschwindigkeit des Fahrzeuges. Dieser Parameter wird bisher durch Methoden gemessen, die die Rotation der Räder erfassen. Bei Schlupf der Räder beziehungsweise bei Traktionsverlust versagen diese Methoden.

Thema dieser Diplomarbeit ist die Entwicklung und Implementierung eines neuartigen Sensor-Systems um Bewegungsparameter eines Fahrzeuges wie z.b. Geschwindigkeit und Beschleunigung zu erfassen. Dieser Sensor basiert auf einer Kombination aus einer Kamera und speziellen *Computer Vision* Algorithmen, die die erforderlichen Parameter aus dem Videostream der Kamera berechnen. Durch die weite Verbreitung von hochqualitativen Video Systemen zu immer geringeren Kosten kann ein auf Video basierendes System zum Beispiel in Fahrassistenzsystemen und sogar autonomen Fahrzeugen verwendet werden. In solchen sicherheitskritischen Systemen ist es notwendig, mehrfach redundante Sensoren für die benötigten Messgrößen zu verwenden, um ein hohes Maß an Fehlertollereanz zu erreichen. Um systematische Fehler, die durch Radschlupf beziehungsweise Traktionsverlust entstehen zu vermeiden, muss eine Methode gefunden werden, die auf einem anderen Prinzip als der Rotation der Räder beruhen.

Es ist üblich, mehrere verschiedene Sensoren für die selbe Messgröße gleichzeitig zu verwenden. Die Messwerte der verschiedenen Sensoren werden durch ein *Sensor Fusion System* ausgewertet, um das gewünschte Maß an Zuverlässigkeit und Fehlertoleranz zu erreichen und außerdem die Genauigkeit der Messung zu erhöhen.Um die bestmögliche Funktion eines *Sensor Fusion System* sicherzustellen ist es notwendig, zu jedem Messwert auch einen Konfidenzwert bereitzustellen, der die Zuversicht in die Genauigkeit des Messwertes ausdrückt. Es wurden Methoden erarbeitet, um aus dem Videostream neben den primären Bewegungsparametern auch Konfidenzwerte zu berechnen.

In dieser Arbeit werden unterschiedliche Algorithmen, basierend auf verschiedenen Methoden, implementiert und untereinander verglichen. Neben der Genauigkeit ist dabei auch der benötigte Rechenaufwand ein relevanter Parameter. Weiters wird untersucht, ob und wie es möglich ist, den Sensor dynamisch an variierende Bedingungen anzupassen, wie beispielsweise verschiedene Geschwindigkeitsbereiche oder unterschiedliche Untergründe (Asphalt gegenüber Kopfsteinpflaster). Als Ergebnis dieser Diplomarbeit wurde ein System als wiederverwendbares Software Modul für das ROS (Robot Operating System) implementiert. Der Sensor wurde unter unterschiedlichen Bedingungen in anwendungsnahen Szenarios mit einem autonomen Forschungsfahrzeug erprobt, um die Funktion des Sensors zu demonstrieren.

Abstract

On the road towards autonomous self driving cars, the reliability and accuracy of the vehicles sensors are becoming even more important than ever. An important parameter is the vehicles speed, which is traditionally measured with methods based on the rotation of the vehicle's wheels. When the vehicles wheels are slipping, these methods fail completely.

The topic of this master thesis is the design and implementation of a new sensor system for gathering measurements of a moving vehicle, i.e. speed and acceleration. The sensor uses a video camera in combination with a computer vision algorithm that extracts the required parameters from the provided image stream.

With the widespread availability of high quality video systems at ever decreasing cost, a system like that could be used for instance in driving assistance systems and even autonomously self driving cars. In such a safety-critical system, it is required to provide several redundant sensors to capture the required parameters, in order to achieve fault tolerance. To avoid systematic errors caused by wheel slipping, it is advisable to use a fundamentally different method. Measurements of all sensors are combined in a sensor fusion system to provide the required level of fault tolerance and increase precision at the same time.

In a sensor fusion system, it is also necessary to have a *confidence* value that indicates if the current measurement is likely to be accurate and precise. Methods have been developed derive such a confidence value from the video stream while calculating the primary movement parameters.

This work explores and implements several algorithms based on different methods, and compares them for their performance. Not only the accuracy, but also the computational effort are important parameters. Furthermore, it was investigated if and how it is possible to dynamically adapt the resulting system to various conditions, for example different speed or the difference between a asphalted street, cobblestone and a meadow.

As the final result of this thesis, a sensor was implemented as a Robot Operating System (ROS) module that can easily be reused.

The sensor was tested in different real world conditions with a autonomous research rover to prove the viability of the implemented sensor.

Contents

Kurzfassung **iii**

Abstract **v**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**
1.1 Motivation 1
1.2 Structure of the work 2

2 Concepts **3**
2.1 Safety-Critical Systems 3
2.2 Fault Tolerance 4
2.2.1 Replication vs. Diversity 4
2.3 Sensor Fusion 5
2.4 Computer Vision 5
2.4.1 Related Fields 7

3 State of the Art **9**
3.1 Related Work 9
3.1.1 Computer Vision Methods 9
3.1.2 Egomotion Estimation 11
3.2 Motion Detection 14
3.3 OpenCV 15
3.4 Robot Operating System 15
3.5 Image Processing Concepts 17
3.5.1 Geometric Transformations 18
3.5.2 Image Pyramids 19
3.5.3 Rotation 20
3.5.4 Lense Distortion and Camera Calibration 23
3.6 Image Processing Algorithms 24
3.6.1 Histogram 24

3.6.2 Error Metric Based . . . 26
3.6.3 Phase Correlation . . . 27
3.6.4 FMT . . . 28
3.6.5 Feature based . . . 29
3.6.6 Optical Flow . . . 31

4 Optical Motion Detection Framework 33
4.1 Overview . . . 33
4.2 Homography Estimation . . . 36
4.2.1 Extracting Results and Relation to Physical Parameters . . . 37
4.2.2 Homography Estimation Methods . . . 39
4.2.3 Removing Outliers . . . 40
4.3 Confidence Estimation . . . 42
4.3.1 Normalized Cross-Correlation (NCC) . . . 42
4.3.2 Feature-Based Method . . . 42
4.3.3 Number of Correspondences . . . 43
4.3.4 Prediction and Physical Model Consensus . . . 43
4.3.5 Combining multiple confidence values . . . 45
4.4 Dynamic Adaptation . . . 46
4.4.1 Adaptive non-maximal suppresion . . . 47
4.4.2 Adaptive ROI Size . . . 47
4.4.3 Adaptive ROI Position . . . 48
4.5 Remarks . . . 50
4.5.1 Known Weakness / Difficult Situations . . . 50
4.5.2 Limits / Restrictions . . . 51

5 Implementation 53
5.1 Overview . . . 53
5.2 Environment . . . 53
5.2.1 Robot Operating System . . . 53
5.2.2 OpenCV . . . 53
5.2.3 Xcode . . . 54
5.3 ROS Integration . . . 54
5.3.1 ROS Modules . . . 54
5.3.2 Image Publisher . . . 55
5.3.3 Sensor . . . 55
5.4 Camera Calibration . . . 57
5.5 Software Architecture . . . 57
5.6 Tile Algorithms . . . 58
5.7 Homography Estimation . . . 60
5.8 Confidence Estimation . . . 61
5.9 Frame Algorithms . . . 62

6 Results 67

6.1 Experimental Setup 67
6.1.1 Reference Movement Sequence 69
6.2 Scenarios 71
6.3 Results 77
6.3.1 General Results 77
6.3.2 Performance Results 78
6.3.3 Velocities and Confidence 81
6.3.4 Least Squares vs. Exact Homography 82
6.3.5 Accuracy 83

7 Conclusion **93**
7.1 Major Results 93
7.2 Future Work 94

A Acronyms **97**

Bibliography **99**

List of Tables

3.1 Hierarchy of geometric transformation from [69]. Each transformation preserves all the properties of the lines below, so e.g. the affine transformation preserves parallelism and straight lines, while the similarity transform also preserves angles. . 19

5.1 Interface Summary for Sensor ROS-Module 55
5.2 Computation steps, the classes that implement these steps and their inputs and outputs for Frame Algorithm composition. 58
5.3 Overview of implemented Frame Algorithms 64

6.1 Computational Performance Results. The Relative Time Factor is based on the reference configuration (Histogram + Exact). 79
6.2 Performance Comparison of different Configurations relative to the reference configuration (Historgram + Exact). 80
6.3 Deviations of measured final cumulative values from measured values. 85
6.4 Deviations of measured velocities from reference velocities 86
6.5 Data for Example Plot 87

List of Figures

2.1 Classification of failures [48] 4
2.2 Some classical geometric-optical illusions from [33] 6
2.3 Computer Vision: Abstraction Levels 7
2.4 Some image processing operations: (a) original image, (b) blured, (c) sharpened, (d) smoothed with edge-preserving filter. Taken from [69] 8

3.1 Example of Image Stitching. The top row of images are separate images of a panorama, they are registered to each other, transformed and blended into a single panorama (bottom image). 10
3.2 ROS-Graph example 17
3.3 Declaration of a ROS message (left) and a ROSservice (right) 17
3.4 Illustration of image transforms [69] 18
3.5 Traditional image pyramid where each level has 1/2 the resolution and therefore 1/4 of the size of its prior level. 19
3.6 Illustration of the transformation from cartesian image coordinates to polar coordinates. 21
3.7 Polar-Transform of an image. Y-Axis is radius and X-Axis is angle. Top left is the original image, top right is the polar transformed image with center in the middle of the original inmage. Bottom left has the transformation center at the left border of the original image (vertically centered) and bottom right has the center even further outside of the original image. 22
3.8 Illustration of rotation center with different steering angles. The car in the left image has a steeper steering angle than the car in the middle,. Therefore the turning radius r_1 is smaller than the radius r_2 of the middle car. The car on the right is drifting on a circle with radius r_3. In that case, the rotation-center does not correspond with the position of the steering wheels. 23
3.9 Examples of radial distortion 24
3.10 Intensity Averaging Algorithm 25
3.11 Example of correlation between the top left and bottom left image. The bottom image is a copy of the top image, shifted by an offset of (10px, -5px). 26
3.12 Example of phase correlation between the top left and bottom left image. The bottom image is a copy of the top image, shifted by an offset of (10px, -5px). 28

3.13 An example of feature matching, the right image is a rotated and translated version of the left one. The threshold for the descriptor distance is set very low to only include very good matches, thus reducing the number of features for better visibility 30
3.14 Examples of Optical Flow. The top left and right image are two consecutive frames of a video with moving cars. The bottom left image shows the optical flow with color coded movement directions. The bottom right image shows a similar scene where the optical flow is visualized with vectors. 32

4.1 Basic concept of using multiple image patches / tiles for image registration 34
4.2 Correcting Rotation 38
4.3 Mapping from pixel-distance to physical length 38
4.4 Confidence-Function for Feature-Matching Distance: $c = 1-(e_{\text{matching}}/k)$ with $k = 5$ 43
4.5 Confidence-Function for Number of Tiles: $c = 1 - e^{\frac{x-1}{2}}$ 44
4.6 Confidence-Function for Prediction Error: $c = 1 - e^{(d-k)}$ with $k = 5$ 45
4.7 Confidence-Function for current Speed: $c = 1 - e^{10(\frac{t}{D}-1)}$ 46
4.8 Dynamic Adaption Overview 46
4.9 Adaptive non-maximal suppresion. In (a) and (b), features are selected by a static threshold on their corner strength. In (c) and (d), the ANMS adaptive threshold is used. [21] 48
4.10 Dynamic adaption of frame size: initially, the speed is unknown, so ROI_1 is a square. Later, the speed is already known and ROI_2 has adapted its dimensions to match the movement speed and direction 49
4.11 Size-Adaption-Function for Adaptive Grid Frame Algorithm: $d = d_0 + v_d$ 49
4.12 Simple Edge Detection: The original image on the top left is filtered and produces the grayscale (almost black/white) image at the bottom left. The brighter the pixel, the stronger the edge. The right image visualizes the bottom left image in a mesh-plot. 50

5.1 ROS-Setup with an external image publisher and the sensor as separate ROS-module. 54
5.2 Measurement Message Definition 55
5.3 Overview of the sensor architecture. This illustrates the subtasks that are necessary to calculate a measurement and the calculation flow through the sensor. 56
5.4 Illustration of how *margins* can be used to constraint the effective image frame. 63
5.5 Tile Layouts for Tile Algorithms: 2 Exact (left) and 4 Least Squares (right) 64
5.6 Tile Layouts for Tile Algorithms: Grid (left) and Adaptive Grid (right) 65

6.1 Pioneer 3-AT Research Rover 68
6.2 Experimental Setup 69
6.3 Illustration of vehicle movement parameters 70
6.4 Illustration of cummulative velocities 71
6.5 Floor-Patterns Street 1 72
6.6 Floor-Patterns Street 2 72
6.7 Floor-Patterns Street 3 73
6.8 Floor-Patterns Cobblestone 73
6.9 Floor-Patterns Street 1 74

6.10 Floor-Patterns Lab Dark 74
6.11 Floor-Patterns Institute 74
6.12 Floor-Patterns Kitchen 75
6.13 Floor-Patterns Foyer 76
6.14 Floor-Patterns Staircase 76
6.15 Floor-Patterns Grass 77
6.16 Debug-Visualization of Optical Flow Method. The color for each pixel represents the movement direction at that pixel, the saturation of the color represents the magnitude of the movement. White pixels correspond to zero movement. 77
6.17 Illustration of image smearing at higher speed. The left image was recorded at low speeds and is sharp. The right image was recorded at higher speed and it is smeared in the movement direction. 78
6.18 Comparison of computation time for various Frame Algorithms, relative to the reference configuration (Historgram). 80
6.19 Comparison of computation time for various Tile Algorithms, relative to the reference configuration (Histogram). 80
6.20 Linear- and Angular Velocity with Confidence for Scenario **street2** with Correlation in Grid Configuration 82
6.21 Comparison of Least Squares Homography Estimation vs. Exact Homography Estimation 83
6.22 Example for Mean / Standard Deviation Plot. The Bar indicates the mean value μ, the line extending goes from $(\mu - \sigma^2)$ to $(\mu + \sigma^2)$ 87
6.23 Lateral Deviations of Correlation Algorithms. The value range has been truncated such that the more accurate methods can be compared better. 88
6.24 Linear Deviations of Correlation Algorithms. The mean values are relatively close to zero, but the standard deviation values vary widely. The value range has been truncated such that the more accurate methods can be compared better. 89
6.25 Angular Deviations of Correlation Algorithms. The mean values are relatively almost zero for every entry, but the standard deviation values vary widely. The value range has been truncated such that the more accurate methods can be compared better. 90

7.1 Dynamic adaption of zoom: frame 1 has a zoom-factor that captures a bigger *Field of View* than frame 2, but also the image details are smaller in frame 1 than in frame 2. 95

CHAPTER 1

Introduction

The research on Autonomous Ground Vehicle (AGV)s is currently an important topic in the field of Cyper-physical System (CPS)s. This is pretty evident, when looking at the development roadmaps of distinctive car manufacturers, where autonomously driving cars play an important role [3], [1], [2], [13]. A wide range of engineering domains are involved, one of them being the sensing and measuring of motion parameters.

For obvious reasons, autonomous vehicles are safety-critical systems - the failure in the control of an autonomous vehicle will most likely result in danger to the life of the vehicles passengers and other traffic participants - and as such, the fault tolerance of all systems involved in the control of the vehicle plays a major role. The reliability of sensors is of particular interest. In this work, the focus lies on sensors for movement parameters, in especial **speed**, **acceleration** and **yaw rate**, which are necessary to determine the movement of a vehicle, to estimate its position and evaluate safety conditions (e.g. the force on breaks in ABS systems). This task of *ego-motion estimation* is in turn a prerequisite for autonomous navigation.

1.1 Motivation

Traditionally, vehicle movement is measured by observing the rotation of the vehicles wheels. Tachometer and Odometer both work based on this measurement.

A Problem arises, when the wheel's rotation does not correspond to the vehicle movement anymore. For example in the case of blocking wheels or when wheel slip occurs. Systematic errors like this usually can not be detected without the use of distinct method. Thus, it is desirable to use various fundamentally different methods for measuring the same value at the same time to achieve fault tolerance.

In this work, an optical system is proposed that works with a monocular camera system installed on the vehicle, pointing perpendicular to the ground and filming the ground below the vehicle. Computer Vision algorithms are applied to analyze the video stream by searching for distinctive features in the image or by overlapping consecutive images and find displacements between them. Based on this analysis, the aforementioned movement parameters are calculated. Potential weaks pots of the developed sensor system will be discussed in more detail in the course of this work. However, in the spirit of fault tolerance, the goal is not necessarily to find an alternative method to replace the others, but to utilize different fundamental methods, so that it is possible that systematic errors of one method can be detected and masked by the other methods.

Combining different sensors, a practice that is also known as *sensor fusion*, is not only done to achieve fault tolerance, but also to achieve better accuracy in general. As an example, imagine that there are different speed sensors available that each have a different, possibly intersecting, range within which they are known to work very accurate, and outside these ranges, their accuracy is known to decrease. By *'fusing'* them together in a sensor fusion system, the value of the sensor which, at the current speed, delivers the most accurate value could be weighted stronger than the value of the other sensors. This approach achieves an accurate measurement over the whole desired range.

For the use in a sensor fusion system, a secondary parameter besides the primary movement parameters is necessary to expresses the confidence in the correctness and accuracy of a measurement. In the previous example, this confidence value would be some function of the measurement itself - if the sensor is known to work good in the range of 0 to 45 km/h for example, the confidence value would be high in this range and then decrease when the reading is over 45 km/h. This approach needs prior knowledge about the specific characteristics of the sensors in use.

Analysis of the camera image can not only be used to calculate the primary movement parameters. Additional results, like statistical parameters of the current image, are generated in the process as temporary values, that can be aggregated to reason about the quality of the measurement. A quick example is the number of tracked features in the images if a feature base method is used. This quality measurement can then be used as a confidence value as additional output of the sensor.

1.2 Structure of the work

In the next chapter, important **concepts** like *safety-critical systems*, *fault tolerance*, *sensor fusion* and *computer vision* will briefly be introduced, with which the reader needs to be familiar in order to understand the content of this work. Following that, the **state-of-the-art** will be presented in Chapter 3.

The main part of this work is described in Chapter 4 and Chapter 5, where the architecture of the sensor system is developed and documented. Different image processing and estimation methods are presented, explained and implemented.

Chapter 6 presents the experimental results of the implemented methods with regards to their performance in different circumstances and compares them with each other.

Finally, the results will be concluded and an outlook for future works is given in Chapter 2.

CHAPTER 2

Concepts

2.1 Safety-Critical Systems

When talking about *safety-critical systems*, it is necessary to define whats even meant by *safety*. “Safety must be defined in terms of hazards or states of the system that when combined with certain environmental conditions could lead to a mishap.” ([50]) This definition can then be extended to a technical system: “Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment.” ([45]) Hence, the consequences of a failure are key criterium for a system to be considered safety-critical. There are applications that could be considered as *traditional* safety-critical. For instance aviation, space flight, railway signaling, nuclear power plants or weapons are typical examples where human lives are potentially at risk. But with the ever expanding use of technical systems in every aspect of our daily life, many more of these systems also gain the potential to be considered safety-critical. In [45], some of these new areas are listed. Examples are *telecommunication systems* like the 911 emergency call infrastructure, *transportation control* and *electrical power generation*. “All of these applications are extensively computerized, and computer failure can and does lead to extensive loss of service with consequent disruption of normal activities. In some cases, the disruption can be very serious.” ([45])

It is important not to confuse *safety* with *security*, even though both concepts are closely related. Both deal with threats and risks, but while these risks in the former one affect life or property, the latter risks in the latter one threaten privacy or national security [50].

The automotive industry, where the use of embedded computer systems has seen a rapid adoption that is still ongoing without foreseeable slowdown, is particularly interesting. Whereas at the beginning, “the computer applications on board a car are focused on non critical body eletronics or comfort functions” ([46]), the industry soon adopted an integration of core vehicle functions like engines, brakes, transmission and suspension “with the goal to increase the vehicle stability in critical driving maneuvers” ([46]), all of which do have safety implications. One of the long term goals of this field is to develop self driving cars. A project that is well known to the public is the *Google Autonomous Car* [12] [7] [4].

2.2 Fault Tolerance

In [47], Kopetz starts the chapter about **fault tolerance** with the following sentence: “Fault tolerance is important in safety-critical real-time systems because otherwise a single component failure may lead to a catastrophic system failure.” ([47])

Fault tolerance, which has been defined by Avizienis in 1971 as “the ability to execute specified algorithms correctly regardless of hardware and program errors” ([15]) is therefore one of the fundamental concepts when dealing with safety-critical systems to achieve the required level of safety. The according field of engineering is concerned with the design, specification, implementation and verification of a system that continues to provide its service and continues operation properly even in the event of the failure of one or several of its components.

Failure of a system is defined as a deviation of the actual from the intended behavior. Failures can have diverse manifestations, Figure 2.1 shows a classification of failure.

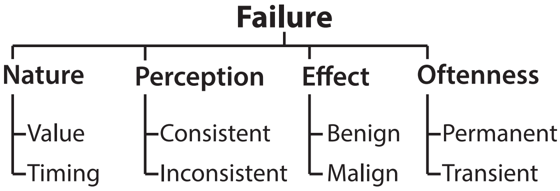


Figure 2.1: Classification of failures [48]

This classification takes into account the **nature** (is a value incorrect or was the value presented at an incorrect time in a real time system), **perception** (does every user perceive the same failure consistent or differently) the **effect** (does the failure cause an effect in the same order of magnitude as the utility of the system or can the failure lead to a catastrophe) and the **oftenness** (does the failure occur only once, or constantly).

Along with the ever increasing complexity of technical systems as a potential cause of errors, the hardware itself becomes more vulnerable to faults: “The trends of shrinking device geometries, lower voltages and higher frequencies in modern processors are expected to increase the rate of intermittent faults” ([73]).

When designing a fault tolerance system, the important parameters are **reliability**(the probability, that the system produces a correct output) and **safety** (the probability, that system either produces a correct output, or a detectable error). For a systematic approach in the design of a fault tolerant system, it is important to specify the types of faults that should be tolerated (using *fault models* and *fault hypothesis*) and how the system should be able to detect and recover from the occurrence of a fault. This specification is later used for the validation and verification of the system.

2.2.1 Replication vs. Diversity

A key concept to achieving fault tolerance in a system is via **redundancy**. There are two kinds of redundancy: **replication** and **diversification**. The first one is the replication of critical components, where for a given level of redundancy (= the number of replicas), a number of identical units are used. A common example is Tripple modular redundancy (TMR), where there are 3 identical replicas of the critical component all work at the same time, and a voter that decides on the final value [47]. Such a system can tolerate the failure of up to 2 units when they fail **silently** or a **byzantine** ([47]) failure of 1 unit.

The other kind of redundancy is **diversification**. This means “a large number of implementation alternatives exist with varying degree of reliability and safety” ([72]). For example when measuring a physical quantity, different methods are used to avoid systematic errors of a specific method. This thesis concentrates on *diversification* by developing a new method to measure movement parameters, to avoid the systematic errors of currently common methods.

2.3 Sensor Fusion

“Sensor Fusion is the combining of sensory data or data derived from sensory data such that the resulting information is in some sense better than would be possible when these sources were used individually.” ([27]) These individual sources are not necessarily required to be identical sensors, a combination of sensor readings from different sensors is also included in this definition just as the combination of multiple successive readings from one single sensor is. In the sense of the above definition, *better* could mean more *accurate* or more *dependable*. In the context of this thesis, the main goal is better dependability.

There are various algorithms to fuse readings from multiple sensors together, from simple **majority voting** and **maximum likelihood estimations** to **Kalman filter** [39] and **Baysian Networks** [18].

2.4 Computer Vision

The overall goal of Computer Vision is to emulate the humans visual perception system and to automatically interpret 3D information from 2D images and videos. We as humans are able to look at a picture of a group of people and count the number of people, identify them and even read into their emotions with appearent ease. In the picture of a scene, we can distinguish between different objects, seperate them from the background and have an intuition about their size and position.

“Perceptual psychologists have spent decades trying to understand how the visual system works and, even though they can devise optical illusions to tease apart some of its principles, a complete solution to this puzzle remains elusive” ([69])

The main problem is, that information is lost when the 3D world is projected into a 2D image, yet it is necessary to infer unknown variables from insufficient information. Many times with ambiguous solutions. The human brain is astoundingly good at finding patterns and reconstructing the missing information, though ambiguous solutions can also trick the human brain in the form of optical illusion (see Figure 2.2 for some examples).

Computer Vision consists of tasks on different abstraction levels. Figure 2.3 tries to illustrate these abstraction levels by means of an example. In the lower level, the image is being processed (see section 2.4.1 for more details). In this example, it is converted from color to grayscale and filtered with a gaussian blur. On the analysis level, the image is segmented. The moving cars

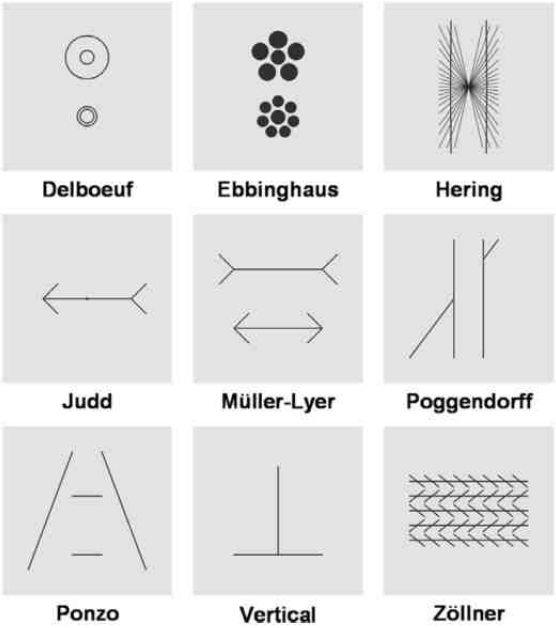


Figure 2.2: Some classical geometric-optical illusions from [33]

are detected and seperated from the background. A green rectangle is drawn around the detected cars and their movement direction and speed is calculated from a sequence of frames like a video stream. On the highest abstraction level, the image is described in natural language just as a human would describe the image to someone else who cant see it: there is a street, two cars are driving into the same direction. Besides these two cars, there seems to be little traffic. Also it seems to be winter, because the side of the street is covered with snow.

The term Computer Vision spans over a lot of different tasks, some examples being:

- **Recognition:** Finding and separating objects in an image, recognizing and categorizing them. For example face detection and recognition.
- **Motion analysis:** Methods and applications to process a sequence of images e.g. from a video, and extracting information from this sequence. Example: Figure 2.3.
- **Scene reconstruction:** Building a 3D model from a set of images of the scene from different perspectives or from an image sequence from a moving camera. E.g. Google Streetview [44], *Building Rome in a Day* [14]
- **Image restoration:** The process of removing disturbances and detriments from an image to recover the original image. [61]

Despite many open problems, Computer Vision is already used in the real worls for real applications. A long list of application can be found in [69], here's just an excerpt, an even more comprehensive list of real world CV applications is given by [51].

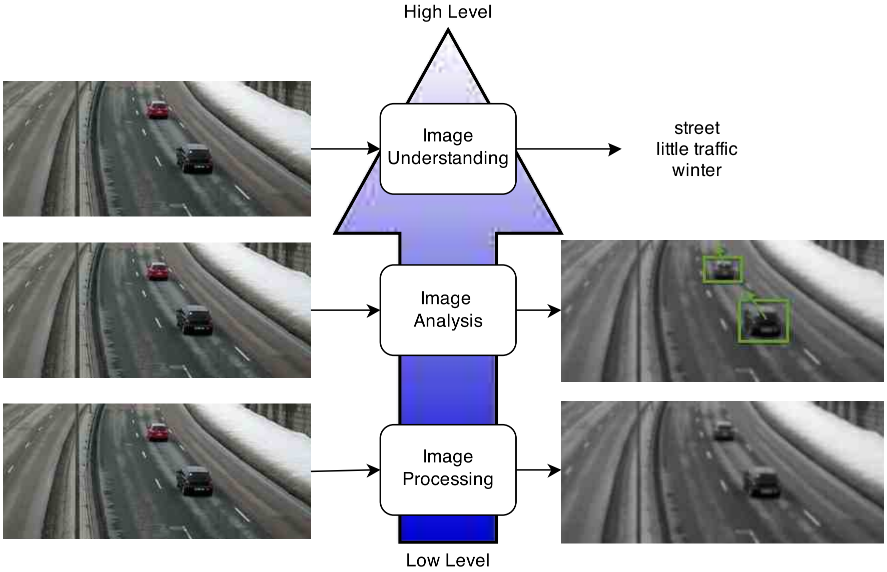


Figure 2.3: Computer Vision: Abstraction Levels

- **Optical character recognition (OCR)** The process of automatically recognizing printed or even handwritten characters from a scan or picture. Already used for example for automatic number plate recognition or for receipts, passports, invoices and other documents.

- **3D model building** The construction of digital 3D models of objects from a moving camera or multiple viewpoints (stereo vision). Already used for construction of 3D maps from aerial photographs, for augmented reality applications or autonomous navigation.

- **Motion Capture** Capturing the natural motion of humans or other things with the use of retro-reflective markers observed from multiple cameras. The cameras are positioned at known positions and can precisely calculate the position of the markers in space. Already used for animations in computer games and movies.

- **Surveillance** Monitoring peoples movement in public places, automatic analysis from surveillance cameras or traffic surveillance, for example in combination with ORC: section control

- **Fingerprint recognition and biometrics** For automatic authentication and access controll, even used in todays smartphones.

2.4.1 Related Fields

The field of computer vision is very closely related to the fields of machine vision and image processing. Other related fields are Mathematics, Statistics, Stochastic, Artificial Intelligence, Machine Learning, Signal Processing, Imaging, and Neurobiology.

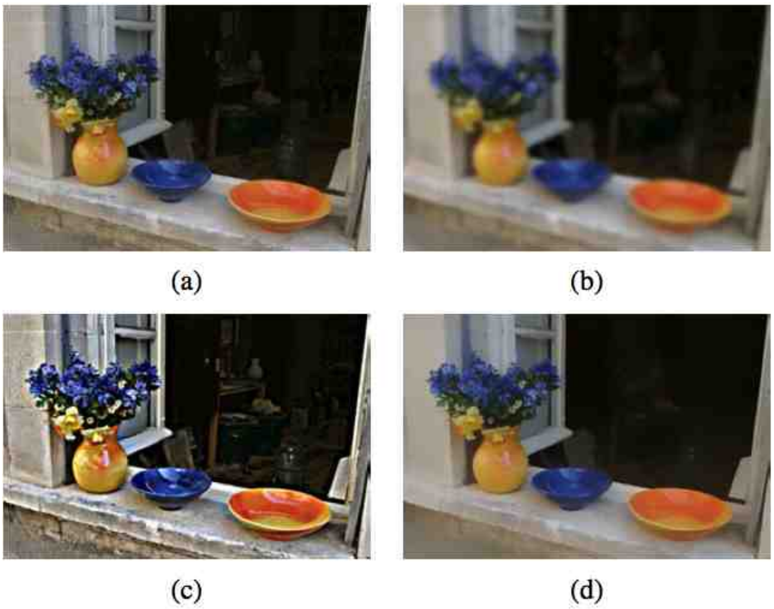


Figure 2.4: Some image processing operations: (a) original image, (b) blured, (c) sharpened, (d) smoothed with edge-preserving filter. Taken from [69]

Machine Vision

The boundaries between Computer Vision and Machine Vision are not exactly defined and there's a big overlap between these two areas. Simply put, while Computer Vision is about *understanding* an image, Machine Vision is 'only' about recognizing certain things. However, while Computer Vision is 'just' about processing an image, starting from the image, the topic of Machine Vision includes the whole process, starting from a physical object through the design and implementation of the mechanical, lightning, optical and camera subsystems [17]. The primary use of Machine Vision methods are for industrial applications where the parameters and the underlying circumstances of the application are very exactly specified and the system is designed exactly to the requirements of the specific application. Examples are barcode/QR-code scanning, quality assurance and sorting on conveyer belts or production lines.

Digital Image Processing

In a wide sense, Image Processing is concerned with every kind of signal processing, where the input is an image or a series of images. It is a term for a range of techniques for manipulating and modifying images. Examples for Image Processing are the removal of noise, the filtering of artifacts, edge detection, image compression or image restoration. See Figure 2.4 for some demonstrations. In practice, almost every Computer Vision application also makes use of Image processing like linear and non-linear filtering, fourier transforms, image pyramids and wavelets. This thesis takes place in the intersection of *Computer Vision* and *Image Processing*

CHAPTER 3

State of the Art

3.1 Related Work

3.1.1 Computer Vision Methods

Computer Vision (CV) is a very broad topic. In the following, some CV areas with notable application for this work will be presented.

Image Stitching

Image Stitching is used to generate seamless panoramic images from single images. Imagine taking a series of images from a wide landscape similar to the top row of images in Figure 3.1. These images are then registered to each other by an **image registration** algorithm. “Image stitching algorithms take the alignment estimates produced by such registration algorithms and blend the images in a seamless manner, taking care to deal with potential problems such as blurring or ghosting caused by parallax and scene movement as well as varying image exposures.” ([68]). In this context, *registration* is the task of transforming different datasets into a common coordinate system. Datasets may be images from different cameras with different viewpoints, images taken at different points in time. Datasets can also be more abstract, like visualizations of extracted features or data from different sensors like depth- or thermal cameras.

Video Stabilization

Video Stabilization is used to correct unsteadiness in a video caused by vibration, bouncing or other small movements of the camera. There are mechanical systems for video stabilization based on accelerometers, gyros or mechanical dampers, but such systems are usually not precise when small, and get really big and bulky with increasing precision [34]. Furthermore, such mechanical systems are generally too large for application in todays smartphones or upcoming gadgets like smart glasses or watches.

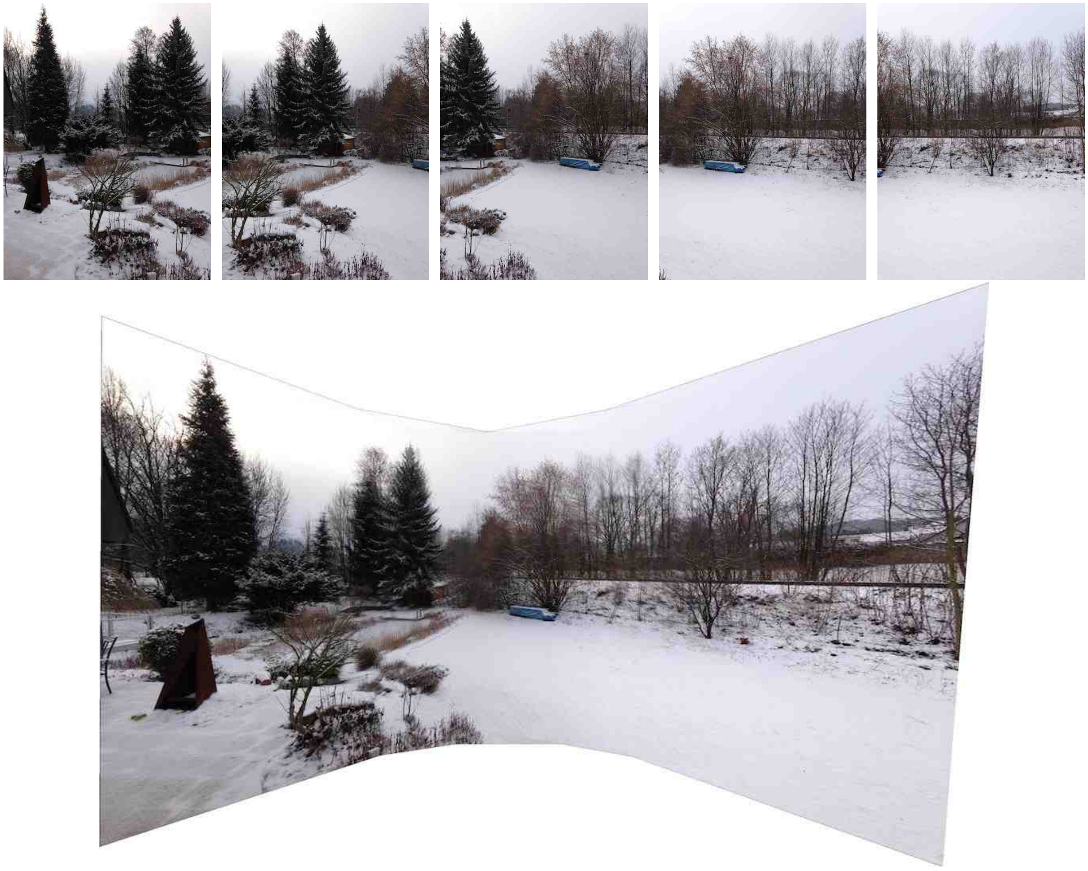


Figure 3.1: Example of Image Stitching. The top row of images are separate images of a panorama, they are registered to each other, transformed and blended into a single panorama (bottom image).

The alternative are pure software solutions that work based on image registration algorithms to correct unwanted movement in images.

Optical Mouse

Optical Mice have to solve the same basic problem in a more constrained setting. “The key to the optical mouse is a sensor chip that reports motion of visible spots relative to the chip coordinate system” ([54]). The image sensor is very close to the surface at a fixed distance, and the surface is constantly illuminated with a self provided light source usually an Light Emitting Diode (LED) or a laser diode. Only translation is detected. The image sensor of a modern optical mouse has a very low resolution e.g. 18x18 pixel, but a very high frame rate of up to 1500 Frames per Second (FPS) [64], [55].

Some related works are based on using optical mouse sensors, e.g. [43] or [49].

3.1.2 Egomotion Estimation

Egomotion Estimation has been an active field of research over the last few years and various methods have been proposed. In this chapter, some related works based in different methods will be presented.

Egomotion Estimation is often used for navigation tasks, for example for autonomous vehicles. By continuously updating the estimated position of the vehicle with the relative movement data from the motion estimation, the position of the vehicle relative to its starting position can be determined. The problem of **passive navigation** - “the ability of an autonomous agent to determine its motion with respect to the environment” ([19]) - is described in [19] as finding two parameters:

- *Heading Direction (HD):* the straight line on which the vehicle is moving.
- *Time to collision (TTC):* the time until the vehicle collides with another object on the straight line projected by the HD

Odometers, gyros or acceleration sensors are often used to determine the vehicles movement, but they are not very precise and with incremental position estimation, the error adds up. In [19] it is claimed, that the task could be performed with visual sensors with lower uncertainty. They introduce a new method based on 3D motion estimation which is computed from a 2D motion field from a camera mounted on the vehicle. This 2D motion field is produced by the motion of the vehicle. This motion is projected to a singular point on the image plane, the so called **Focus of Expansion (FOE)** It is explained, that methods for motion estimation often only treat cases where the rotation is bounded to values where the FOE still is within the Field of View (FOV). Since even small rotations or camera vibrations could violate this condition, these methods efficiency in real contexts strongly depends on the angular velocity values. The proposed method aims to eliminate this restriction by using a “3D motion estimation approach based on approximating the available 2D motion field with a linear combination of 2D bases functions describing the projections on the image plane of 3D elementary motion.” ([19])

[49] works on the same premise as this work. They begin with anaysing the problem of localization that can be further decomposed into *absolute* or *global* localization, and *relative* localization. They concentrate their work on the latter. They note, that usually wheel based method, for example with wheel encoders, are used, and that wheel slip or uneven surfaces could cause errors and poor estimates. Thus, they describe a method that works even in the presence of wheel slip. The proposed method is based on the use of *optical flow* sensors, in fact, they use commercial optical mice as sensors.

If only a single sensor is used, the robots angular movement is lost and a different method has to be used in addition to measure it. Furthermore, a kinematic constraint is given, that assumes the center of the robot to move only perpendicular to the wheel axis. If multiple optical sensors are used, it is either possible to increase accuracy and confidence by feeding the results of all sensors into a data fusion method, or it allows to remove the kinematic constraint when the sensors are placed at suitable locations. Following this result, a lot of attention is paid to the placement of the sensors on the robot, considering different placements in the case of a single sensor as well as when using multiple sensors.

Since they use optical mice, which have a very narrow band of operation with regard to the focal length, they need to place the sensors very close to the ground. In fact, they needed to apply force to press the mice to the ground. This increased the friction, and therefore the chance of wheel slipping. Obviously, this fact also renders this method unusable in environments other than a perfectly flat and smooth surface.

Even though they started out to investigate a very similar problem, they arrived at a solution specialized to the specific application in small to medium robots that operate indoors. It can be assumed that their solution is computationally much less demanding than our method, but this advantage comes at the price of versatility and outdoor applicability. Similar work to [49] was done by [63] as well.

[43] expands the same approach using multiple optical mice as sensors, positioned in a triangular, or general polygon array to enable omnidirectional velocity estimation. They focus heavily on techniques that can be used to estimate velocity from multiple sensors installed in such an arrangement, and most importantly on how to calibrate such an array of sensors. Since the installation of sensors is almost certainly not absolutely perfectly aligned with the triangle or polygon, the resulting accuracy may be degraded significantly. Therefore, calibration with the help of other sensors, for example wheel encoders, is required. Obviously, calibration has to be performed in an environment where systematic errors for wheel based methods, like wheel slipping, do not occur. This method is intended to be simple, efficient and low cost, and the use in personal service robots is specifically mentioned.

[24] also focuses on the position of multiple optical mouse sensors on a mobile robot, providing a method to locate N sensors on the robot that works by solving an optimization problem.

The use of all these methods that are based on optical mice are limited to the use in extremely flat surfaces only found in indoor environments because of limitations of the mouse sensor.

When it comes from indoors applications to cars on real streets, requirements in robustness increase. Computer Vision in general is already applicable for a variety of tasks, beginning from already available parking assistance [67], over lane departure warning systems and lane guidance assistants [25], [76], including meta tasks like detection of free parking spots [38], to all sorts of sensing tasks in the development of fully autonomous cars [28].

[58] gives an very brief overview of the computer vision algorithms and architectures that are involved in an autonomous car.

[16] is using a mobile stereo camera setup computing 3D points from the environment and additionally establishing correspondences between points in successive frames by optical flow calculation. Using multiple cameras, it is possible to calculate the 3D coordinates of points in the environment in a passive way - meaning that no additional information is necessary - by triangulation. The goal is to calculate all six degrees of movement (translation and rotation, each with 3 parameters in the 3-dimensional space) of a vehicle in typical traffic situations. For the egomotion estimation, it is necessary to determine the movement of the camera with respect

to a static scene. In typical traffic situations, there is not only the static scene, but also other self moving participants that must be excluded from the static scene. This is effectively achieved by imposing a so called Smoothness Motion Constraint (SMC) based on the assumption that the frame rate is high enough that the movement between single consecutive frames is very similar. Then, every correspondence is checked with the previous movement and is rejected, if the error is greater than a certain threshold. This threshold needs to be chosen in a way, that any reasonable change in movement between frames is still within the threshold, yet independently moving objects are rejected. They use a fairly sophisticated model to determine the threshold. First, the optical flow is computed for the image of the left camera. If the calculation was successful, the disparities between left and right image are calculated, followed by triangulation. A list of tracked points with their 3D position is generated in order to allow multi frame motion estimation, meaning that the estimation is not only based on the current and the previous frame, but on the sequence of m frames preceding the current one. The aforementioned smoothness constraint is applied to exclude outliers and the remaining points are used to determine the motion parameters with a least squares method. The result is an estimation for the 6 motion parameters, 3 for translation and 3 for rotation in space.

The combination of *stereo vision*, *optical flow*, *smoothness constraint* and *multi-frame estimation* yields very good results in some real life experiments. It is claimed to be better than odometry and to provide better results than GPS for short travelled distances.

While stereo camera systems, when calibrated, are able to recover 3D coordinates of objects in the scene via depth perception, this is not directly possible in monocular systems. Still, it is possible to derive relative motion of the camera by detecting and tracking features and calculating displacements thereof. Many existing methods for egomotion estimation are based on optical flow or local image features to establish a spatial relationship between two images, [31] presents a new method that makes use of the Fourier-Mellin Transform (FMT) for registering images in a video sequence to then calculate the rotation and translation of the camera motion. The FMT is an efficient and accurate method for image pairs where the assumption of rigid transformation between two images holds. That is, that the distortions due to perspective projection are not too large. A camera mounted at the front of the vehicle is used, pointing to the ground at a certain angle. The tilt of the camera with respect to the ground plane needs to be rectified before continuing with the calculation, which is no problem since the mounting parameters (distance to the ground and the angles) are known beforehand. Furthermore, a similar smoothness constraint is applied similar to the method in [16].

In their experiments, they tried different window functions in order “to remove antialiasing effects introduced by the log-polar representation of the FMT” ([31]). They found a Kaiser window to work best.

Comparisons of the FMT method with a correlation based optical flow method and with a SIFT-feature based method are presented, and the biggest differences between the methods seems to be the amount of rotation recovered. Unfortunately, they have not compared the computational performance of the different methods.

In [41], an ego-motion estimation technique is presented that works much simpler than the previously presented publications. They focus on high computational performance to achieve computation at very high framerates and low latency. One of the potential applications mentioned, besides robot movement, is human head tracking, for example for usage in virtual reality headsets. In this application, very low latency is crucial and in fact more important than absolute accuracy. More generally, motion capturing is considered, where there are two principal approaches. In the first one, a specific area is equipped with cameras that are set up and calibrated to track multiple markers in the 3D room that is monitored. The problem with this approach is, that the area is limited and the setup and calibration are long-winded tasks. The alternative is mounting cameras on the subject and estimating the position *inside out* like, e.g., the Wii Remote does. Still, these systems usually require markers placed in the area to be sensed by the camera and this limits the space yet again. Kato proposes an algorithm that works as *cameras on the body* technique that doesn't require markers and needs low computational power to achieve a high update rate and a low latency.

Basically, their method works by calculating intensity averages for each row and each column of the image and then determining the x- and y offset by calculating the correlation with the previous row- and column arrays. Obviously, this method requires very little calculation and can therefore be performed really fast. They claim to have reached 714 FPS with their experimental setup on an Intel Core2 Duo with 2.80GHz. This benchmark only includes the calculation, since they had no camera that could operate on such high frame rates. Also, results varied widely between x- and y-axis, most likely because the images are not square and the length of the row- and column array are therefore different.

Obviously, this method can only calculate movement in x- and y-direction. Rotation can not be detected, but it is very simple and efficient, and seems to be well suited for tasks like head tracking on a Head-mounted Display (HMD) where high update rate and low latency is more important than precision.

3.2 Motion Detection

State-of-the-art are systems that work with odometry data or dead reckoning based on wheel rotation. Odometry data is gathered based on the rotation of the wheels of a vehicle via rotary encoders. When the vehicle moves, the wheels and therefore the rotary encoders turn by an amount which is directly correlated with the distance the vehicle has traveled. Problems arise, when this direct correlation is violated, for example when *wheel slip* occurs. In such a situation, the sensor readings will be wrong. Alternatives are GPS or inertia sensors, all of them with a different set of advantages and disadvantages. GPS does not suffer from wheel slip because it is able to determine the absolute position of the vehicle. However, GPS is only accurate to a range of a few meters (the GPS signal in space is guaranteed to provide a worst case pseudo-range accuracy of 7.8 meter at 95% confidence level, [71]), which is not good enough for most applications. Even worse, its use is limited to outdoors with line-of-sight to at least 4 satellites. In urban environment (*canyon effect*), tunnels or indoors, the service performance degrades or ceases to work at all. Inertial sensors like gyros do not suffer from wheel slip as well, but they are not precise enough, and incremental error sums up over time. Additionaly, they often have to be re-calibrated.

3.3 OpenCV

"OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code." ([8])

The development of OpenCV was started by Intel in 1999 and was later continued by Willow Garage in 2008 ([30]) where it is currently in active development. It is available for a wide range of platforms and operating systems, including Windows, Linux, Mac OS as well as for mobile Platforms like iOS and Android. The Library is written in C++ (and still contains some legacy C), but interfaces are available for a lot of programming languages, including Python, Java or Matlab as well as wrappers for additional ones like C# or Ruby.

"The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc." ([8])

Thanks to its release under a BSD license it is free for the use in academic as well as in commercial use and is in fact used by many major companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda or Toyota [8]

3.4 Robot Operating System

In general, the acronym **ROS** stands for **Robot Operating System** and describes a collection of tools for the specific purpose of controlling the hardware of a robot:

"A robot operating system (ROS) is similar to that of an operating system on a personal computer, in that it comprises a collection of programs which offer control to a user. In the case of a ROS however, these programs allow a user to control the mobile operations of a robot rather than applications on a computer." ([42])

In [42], 16 different ROS's were analyzed with the goal to find a suitable system as a teaching tool in their classroom for students with limited programming experience. While the details of their research is of limited interest for this work, it serves as an example for the wide range of ROS's that exist.

In the context of this work however, the term **ROS** always references one specific system that goes by the that exact name.

The **Robot Operatiing System (ROS)** [11][59] is a flexible software framework as well as a collection of libraries for creating robot applications. In [59], the philosophical goals of ROS are summarized to be

- Peer-to-peer
- Tools-based
- Multi-lingual
- Thin
- Free and Open-Source

The fundamental concepts in ROS are the **nodes**, **messages**, **topics** and **services**.

Nodes

In the context of ROS, the term *node* is synonym to the term *software module*. A modular system can consist of many nodes where each node is a closed process of computation, while nodes can communicate with each other via *messages* and *services*.

"Our use of the term "node" arises from visualizations of ROS- based systems at runtime: when many nodes are running, it is convenient to render the peer-to-peer communications as a graph, with processes as graph nodes and the peer-to-peer links as arcs." ([59])

Figure 3.2 shows an example of a ROS-graph, **gscam_publisher**, **rostopic_5688_1397667160639** and **sensor** are 3 nodes in this graph.

Topics

In order to organize messages in a meaningful way, they are always published for a specific *topic*. A topic is just a name that gives additional semantic to a message, examples of topics could be *camera*, *localization*, *sensor* or *movement*. Nodes can publish to several topics, and several nodes can publish to the same topic. Conversely, a node can subscribe several topics, and several nodes can subscribe to the same topic. Publishers and subscribers do not know of each others existence, let alone about details of their implementation. The utilized *Publish- and Subscribe* therefore helps to decouple nodes and abstract away from their implementation details.

In the example ROS-graph in Figure 3.2 **camera** is a topic with one published message-type *image_raw*. The node **gscam_publisher** publishes this topic, and the two nodes **rostopic_5688_1397667160639** and **sensor** subscribe this topic.

Messages

To communicate with each other, nodes send *messages* to each other. The process of message passing is provided directly by the ROS core, so nodes only need to publish messages and subscribe to topics to receive messages. The structure of a message is declared in an implementation unspecific format and can contain simple datatypes like integers and floats as well as arrays and

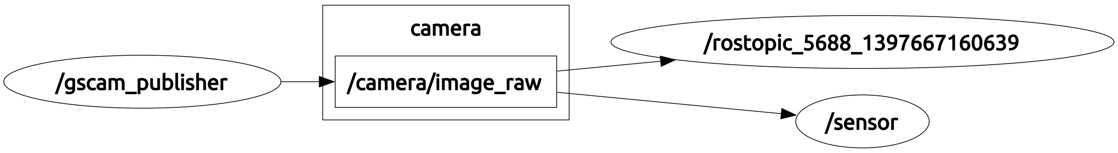


Figure 3.2: ROS-Graph example

```
string first_name              int64 a
string last_name               int64 b
uint8 age                      ---
uint32 score                   int64 sum
```

Figure 3.3: Declaration of a ROS message (left) and a ROSservice (right)

even other messages. An example of such a declaration can be seen in Figure 3.3. The message generation tool from the ROS toolset generates the specific message in whatever language necessary from the declaration.

Services

There are use-cases, where the broadcast scheme of the *Publish- and Subscribe* architecture is not appropriate and a tighter, synchronous communication is needed. For these use-cases, ROS provides the concept of a *service*. A service is declared similar to a message, but in this case, two message structures are declared: one for the *request* and one for the *response*. Just as for a message, the message generation tool generates the specific service-messages from this declaration for which an example is given in Figure 3.3. In contrast to topics, only one node can advertise a specific service at a time.

3.5 Image Processing Concepts

The basic problem is known as the **Image Registration** problem. It has to do with the task of matching two or more images from a single scene taken either from a different viewpoint or at different times and geometrically align them. “The goal of registration is to establish geometric correspondence between the images so that they may be transformed, compared and analyzed in a common reference frame” ([75]). These images could also come from different sensors like a depth-map or a heath image from a thermal camera. In the concrete case of this work, image registration is used to determine the motion of an object (the vehicle) from a set of images that show the floor below the object. Consecutive images are registered to each other and the geometrical transformation, especially translation and rotation, between them is used to reason about the motion of the object.

In [32], a large survey of different image registration techniques can be found. This section will describe general considerations about the necessary image processing on a higher level. Concrete methods will be presented in section 3.6.

3.5.1 Geometric Transformations

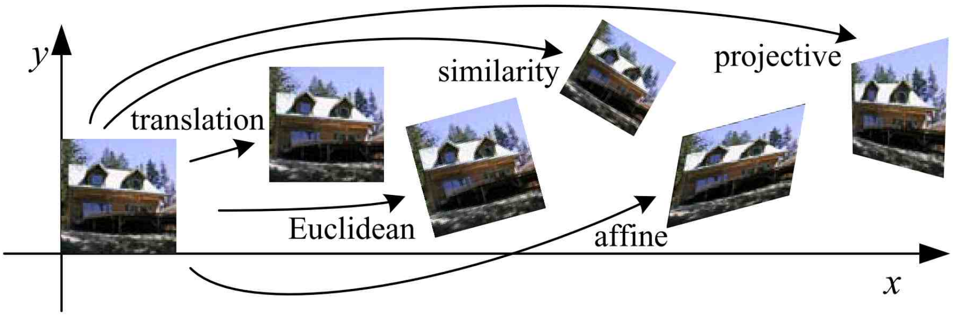


Figure 3.4: Illustration of image transforms [69]

Considered are only transformations on the 2D plane. Figure 3.4 displays these transformations with an example image. There is an inherent order to these transformations with increasing number of Degrees of Freedom (DOF), that means with increasing number of parameters. Starting from the most simple one, translation(which is just a shift of the image in x- and y direction), each transformation contains the image changes of all previous transformation plus additional changes. Therefore, an rigid (or Euclidean) transform consists of translation plus rotation of the original image. The similarity transform adds a change in scale, the affine transformation adds shearing and the projective transformation finally adds a perspective distortion. Similarly, there is a number of properties of the original image preserved by the transformation. For these, the highest order transform preserves the least number of properties (i.e. the projective transform only preserves straight lines) and in decreasing order, each next transformation preserves all the previous properties plus an additional one. Table 3.1 shows a summary of the transformations with their preserved properties.

This table also indicates the necessary size of the transformation matrix to represent the according transformation.

When working with transformations on 2D-images, image points are usually converted to *homogeneous* coordinates. Every coordinate (x, y) can be converted to a homogeneous coordinate x_1, x_2, x_3 where $x = \frac{x_1}{x_3}$ and $y = \frac{x_2}{x_3}$.

In this representation, the effect of a transformation can be calculated by multiplying the source point with the transformation matrix:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \cdot [\,\mathbf{H}\,]_{3x3} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

For this, the transformation matrix needs to be of size $3x3$. Each $2x3$ matrix in Table 3.1 can be extended to a full homogeneous transformation matrix by appending a third row with $[\,0\;0\;1\,]$.

Chapter 3.6 in [69] goes more into detail about geometric transforms of images.

Transformation	Matrix	#DOF	Preserves
translation	$[\,\mathbf{I} \mid \mathbf{t}\,]_{2x3}$	2	orientation
rigid (Euclidean)	$[\,\mathbf{R} \mid \mathbf{t}\,]_{2x3}$	3	lengths
similarity	$[\,s\mathbf{R} \mid \mathbf{t}\,]_{2x3}$	4	angles
affine	$[\,\mathbf{A}\,]_{2x3}$	6	parallelism
projective	$[\,\tilde{\mathbf{H}}\,]_{3x3}$	8	straight lines

Table 3.1: Hierarchy of geometric transformation from [69]. Each transformation preserves all the properties of the lines below, so e.g. the affine transformation preserves parallelism and straight lines, while the similarity transform also preserves angles.

3.5.2 Image Pyramids

An Image Pyramid is a multi-scale representation of an image constructed from that image and several downscaled copies thereof, arranged in levels with each level being of lower resolution than the next higher level. Figure 3.5 displays the concept of image pyramids with an example.

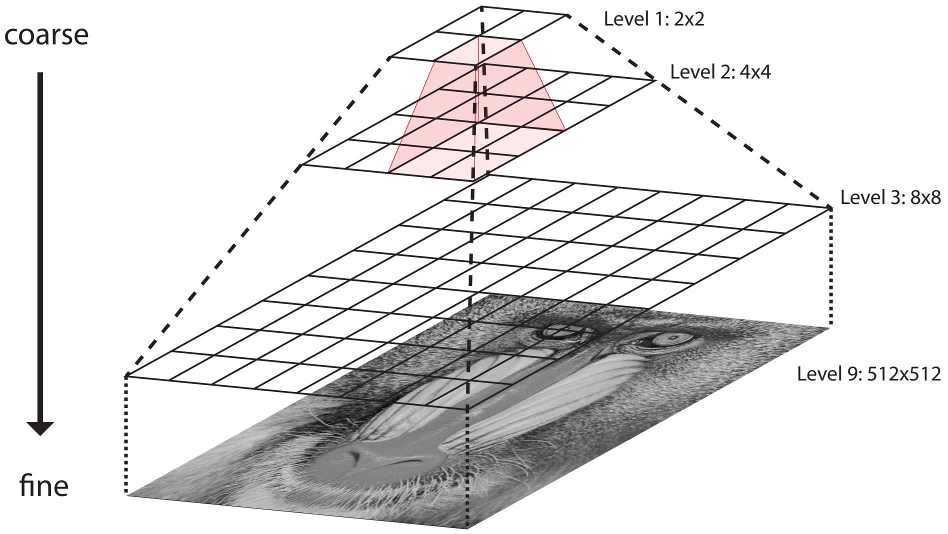


Figure 3.5: Traditional image pyramid where each level has 1/2 the resolution and therefore 1/4 of the size of its prior level.

There are different ways to construct specific pyramids, like **gaussian pyramides** [65], **laplacian pyramids** [22] or more complex forms like the scale space consisting of sub-octave gaussian and Difference of Gaussian (DoG) levels used in Scale Invariant Feature Transform (SIFT) [52]. These methods differ mostly in the way the downscaling between image levels is performed.

Pyramids are commonly used for *scale invariant* algorithms like the aforementioned SIFT-algorithm, where, basically, the scale of a feature is given by the level of the pyramid on which it was detected. Another common application of pyramids is to "accelerate coarse-to-fine search algorithms" ([69]).

The latter one is the application that is useful for improving the performance of an algorithm based on an error metric (i.e. any kind of function that determines, how different two images are, taking into account their relative orientation to each other. Common error metrics are Normalized Cross-Correlation (NCC) or Sum of Square Differences (SSD), performed on grayscale versions of the images. The goal is to find an relative alignment of the images, where the error is minimal). The algorithm starts on the coarsest scale and aligns the image as good as possible. Because the resolution is low, the number of possible alignments that have to be tried is low. When the best alignment is found, the procedure is repeated on the next finer scale of the pyramid, where the previously found alignment is used as initial position, and the search range is limited by the scale factor between the levels. Not only the number of positions that have to be tried is much lower compared to the version without image pyramid, but also the total **number of pixels** that have to be taken into account for the calculation of the error is reduced by mainly on downscaled versions of the original image. On the other hand, the construction of the pyramid, especially downscaling between levels introduces additional computational overhead, though this is usually negligible when compared to the savings.

3.5.3 Rotation

It was mentioned before that the possibility of rotation already increases the complexity of the image registration problem significantly. As long as only translation is allowed, it is still feasible to simply '*guess and check*' by overlaying both frames and calculating the error, repeat this for every possible offset by shifting the images relatively to each other on the x- and y axis and then take the relative orientation with the best match or lowest error. This process can be optimized by the use of Image Pyramids as described in subsection 3.5.2.

However, as soon as rotation is allowed, the number of possible configurations that have to be checked simply becomes to large to be feasibly solved by such a naive approach. Therefore, a more efficient method is needed.

Polar Transform

Usually, an image is presented in a two dimensional spacial domain over **x** and **y** which represent a coordinate on a scale corresponding to the width and the height of an image in cartesian coordinates. The same spacial image can also be represented in polar coordinates. This representation brings some special properties. One of them being the fact that rotation of the image in cartesian domain is transformed to a pure translation on the φ-axis in polar coordinates. This property makes it possible to apply methods that can only detect **translation** on the transformed image in order to find the rotation.

To calculate the transformation, for every point (x, y) the polar coordinate (r, φ) can be calculated with the following formulas:

$$r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$$

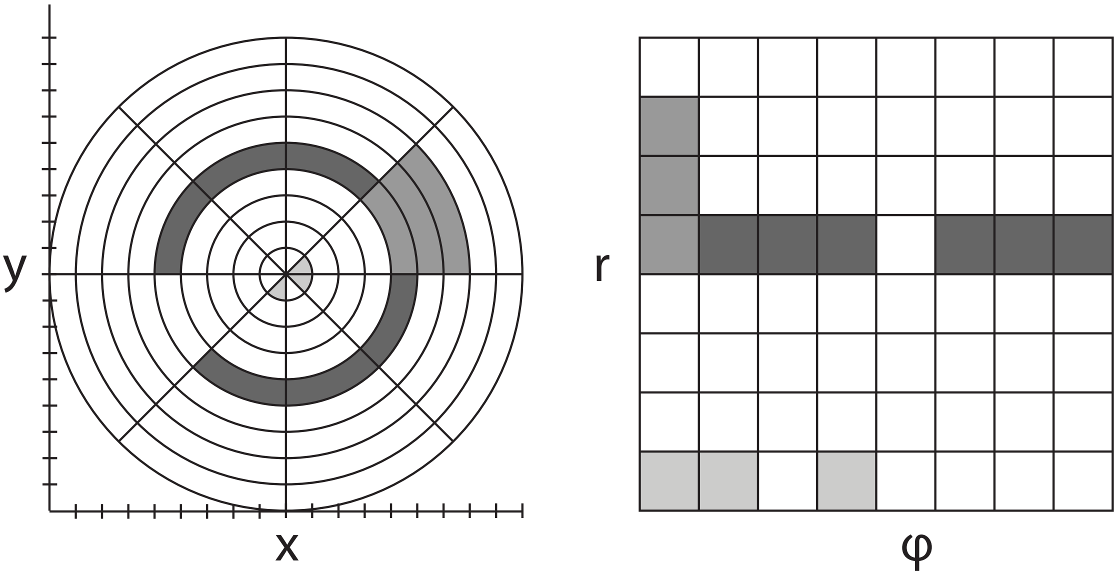


Figure 3.6: Illustration of the transformation from cartesian image coordinates to polar coordinates.

$$\varphi = \tan^{-1}\left(\frac{y - y_c}{x - x_c}\right)$$

where (x_c, y_c) is the center point of the transformation which can be chosen arbitrarily. Figure 3.7 shows the effect of choosing different center points.

Note that both, r and φ are unlikely to result in integers which would be required to represent the result again in an image. To correct this, interpolation has to be used. Figure 3.6 visualizes the process of polar-transformation and Figure 3.7 illustrates the transformation with a real example image.

An extension of the polar transform is the Log-Polar Transform (LTP) which is described and used in [62] and [75]. In the LTP, the *logarithm* of the radius is used which results in a change of the coordinates $(r, \varphi) = (\log(r), \varphi)$ and brings the advantage, that a change in scale of the cartesian image is mapped to a translation along the r-axis in log-polar representation and thus can be detected just as easily as rotation and translation. However, in the scope of this work, change in scale is not of further interest.

There's one caveat to the method of using polar-transformed images though: the center of the rotation has to be known beforehand. In principle, it would be possible to configure that based on the position of the sensor on the vehicle, but theory rotational center of a vehicle is not constant and depends on the steering angle. Figure 3.8 illustrates this with the example of a car that turns with two different steering angles. Both cars turn on concentric circles around the same center which is far outside of the images. The radius r_1 of the left car is smaller than the radius r_2 of the middle car. Therefore the rotation center, relative to the car, is located at a different distance depending on the steering angle.

Theoretically, it would be possible to read out the position of the steering wheel and calculate

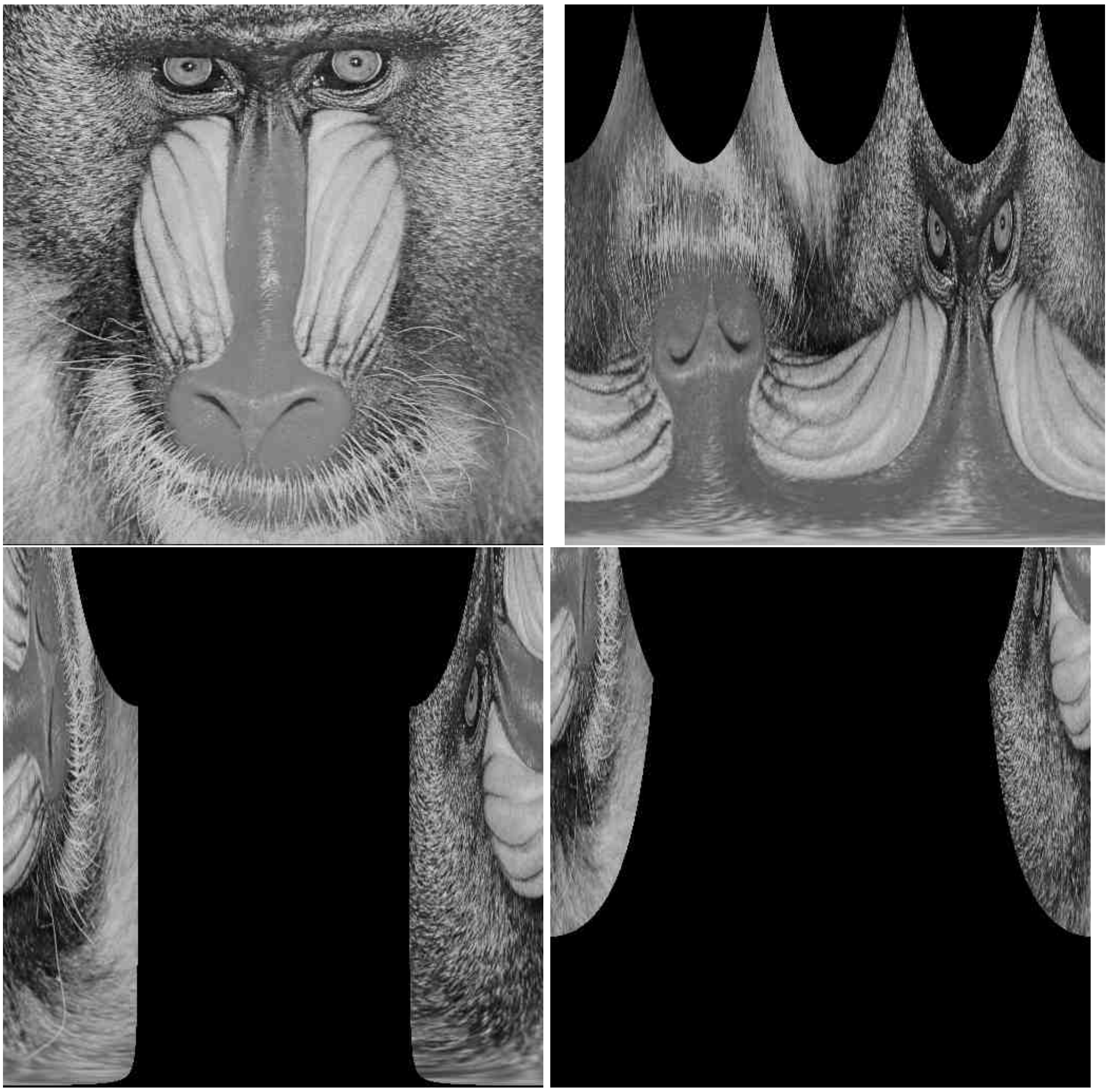


Figure 3.7: Polar-Transform of an image. Y-Axis is radius and X-Axis is angle. Top left is the original image, top right is the polar transformed image with center in the middle of the original inmage. Bottom left has the transformation center at the left border of the original image (vertically centered) and bottom right has the center even further outside of the original image.

the rotation center based on that position. This could work as long as the vehicle is moving *normally*. The car on the right is *drifting* on a circle with radius r_3. In this scenario, the radius does not correspond with the steering angle anymore. Even worse, the line from the car to the center of the rotation is not perpendicular anymore, which is hinted by the short-dashed line. Albeit, it could be defined that the measurement is not required to be perfectly accurate in such an unusual case. Wheel based methods would work even worse in such a case because of

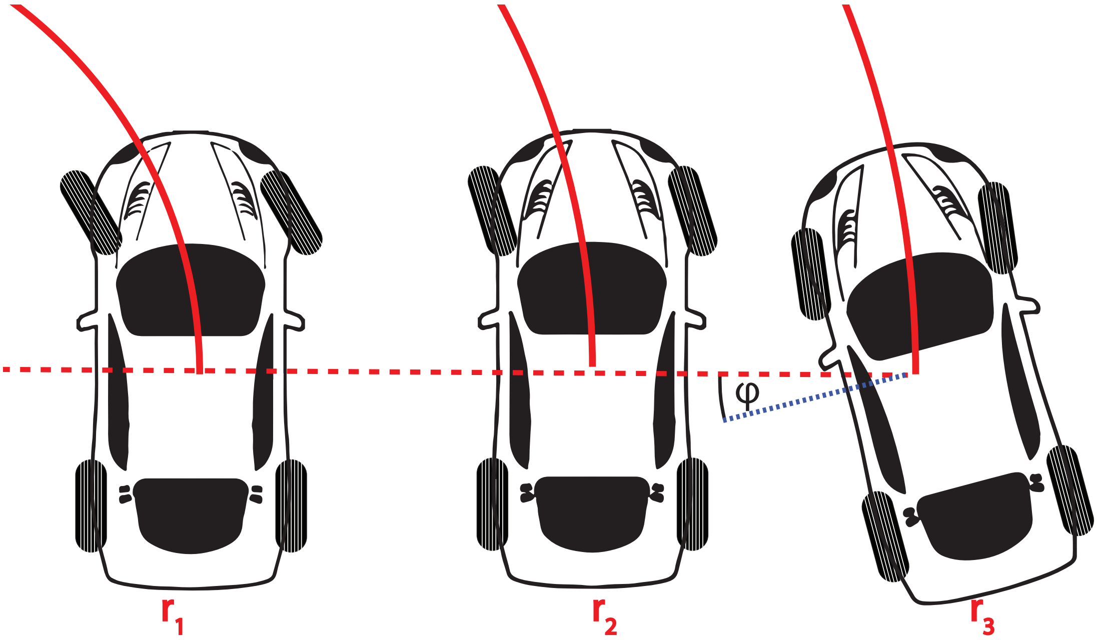


Figure 3.8: Illustration of rotation center with different steering angles. The car in the left image has a steeper steering angle than the car in the middle,. Therefore the turning radius r_1 is smaller than the radius r_2 of the middle car. The car on the right is drifting on a circle with radius r_3. In that case, the rotation-center does not correspond with the position of the steering wheels.

spinning wheels. Another problem is also already hinted in Figure 3.8. The radius from the car to the rotational center can become very large, especially for small steering angles. But still, for the maximum rate at which a car can turn, the rotation center lies well outside of the image. The bottom examples in Figure 3.7 show the effect of a center at the edge of the image or outside: a large portion of the resulting image is simply black while the original image is compressed into a small portion of the transformed image. This could be solved by not transforming the whole range from $0 \leq \varphi \leq 2\Pi$ and $0 \leq r \leq r_{\text{max}}$, but rather only the range where the original image is located $0 \leq \varphi_{\text{min}} \leq \varphi \leq \varphi_{\text{max}} \leq 2\Pi$ and $0 \leq r_{\text{min}} \leq r \leq r_{\text{max}}$.

3.5.4 Lense Distortion and Camera Calibration

Due to imperfect lenses, most cameras do not exactly produce a *linear* projection. This unwanted *nonlinearity* results in a distortion in the produced image that either pushes image points further away from the image center (*barrel distortion*, Figure 3.9b) or pulls them towards the image center (*pincushion distortion*, Figure 3.9c) by an amount proportional to their distance from the image center [69].

This radial distortion results in straight lines not appearing as straight lines in the produced image. This reduces the quality of measurements.

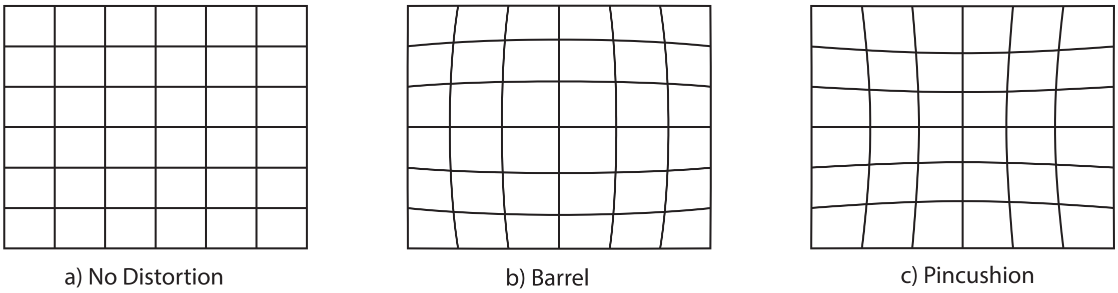


Figure 3.9: Examples of radial distortion

It is possible to correct this distortion separately. This process is described in [69] in chapter 2.1.6. In order to perform the correction process, the distortion parameters need to be determined first, which can be done by a camera calibration process. There has been a lot of research on the topic of *camera calibration* ([23], [77], [66]). The approach that has been used for the implementation is described in Section 5.4.

3.6 Image Processing Algorithms

The *image registration problem* presented in the previous section can be found in a broad variety of practical problems from *panoramic image stitching* to *video stabilization* and thus, a variety of algorithms to solve the problems exists. According to [62], they can be classified into two categories. The first class consists of **intensity based** methods where the whole image is used to calculate the transformation based on intensity values of each pixel. The other class are **feature based** methods. Here, images are first analyzed and points of interests (also called *features*) are calculated. In the next step, only these features are used to calculate the mapping.

This section presents some algorithms of both classes that have been implemented. These implementations have been tested in experiments to compare them and to find, which ones produce the best results for certain scenarios, these results can be found in chapter 6.

3.6.1 Histogram

This is one of the simplest possible methods which is also computationally very cheap. It is presented in [41] as a method for ego motion estimation. Specifically mentioned applications are human head tracking and robot movement.

Given the n-th image in a sequence with X rows and Y columns, the method works by first converting the image into grayscale. This conversion results in exactly one intensity value for each pixel that can then be normalized to the range $0...1$. These intensity values are then summed up for each row and each column as is demonstrated in Figure 3.10.

$$\mathbf{R_n}[y] = \sum_{x=0}^{X} I_n(x, y)$$

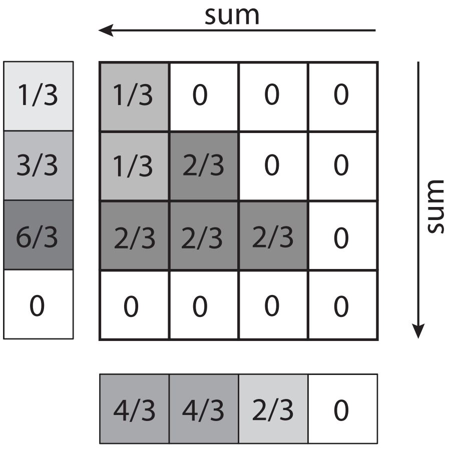


Figure 3.10: Intensity Averaging Algorithm

$$\mathbf{C_n}[x] = \sum_{y=0}^{Y} I_n(x,y)$$

I_n... n-th image, $\mathbf{R_n}(y)$... Sum of Row y, $\mathbf{C_n}(x)$... Sum of Column x

$\mathbf{R_n}$ and $\mathbf{C_n}$ are vectors with length equal to the width and height of the image $\mathbf{I}$ respectively, containing the column- and row intensity averages for the n-th image. These vectors can be compared with their corresponding counterparts from the previous frame by calculating the cross-correlations between the n-th and the $(n-1)$-th frame:

$$r_{row}(j) = \frac{\sum_{i=0}^{Y}(\mathbf{R_n}(i) - \overline{\mathbf{R_n}}) \cdot (\mathbf{R_{n-1}}(i-j) - \overline{\mathbf{R_{n-1}}})}{s_{\mathbf{R_n}} \cdot s_{\mathbf{R_{n-1}}}}$$

$$r_{col}(j) = \frac{\sum_{i=0}^{X}(\mathbf{C_n}(i) - \overline{\mathbf{C_n}}) \cdot (\mathbf{C_{n-1}}(i-j) - \overline{\mathbf{C_{n-1}}})}{s_{\mathbf{C_n}} \cdot s_{\mathbf{C_{n-1}}}}$$

with $\overline{\mathbf{R_n}}$ and $\overline{\mathbf{C_n}}$ being the mean of $\mathbf{R_n}$ and $\mathbf{C_n}$, and $s_{\mathbf{R_n}}$ and $s_{\mathbf{C_n}}$ being the standard deviations of $\mathbf{R_n}$ and $\mathbf{C_n}$ respectively.

The offset between frame $n-1$ and n can then be found by searching for the index of the maximum $\max(r_{row})$ and $\max(r_{col})$. The parameter j can range from $-X$ to X and $-Y$ to Y respectively. Since the index $(i-j)$ could potentially be outside of the array, it is important to define how to deal with these cases. One possibility is to simply ignore these cases and only include terms in the sum where the index is within the bounds. Another way is to take the last valid value.

$$C(x) = \begin{cases} C(x) & 0 \le x \le X \\ C(0) & x \le 0 \\ C(X) & x \ge X \end{cases}$$

The biggest advantage of this method is, that it is computationally very cheap compared to the following methods. Its results are expected to be reasonably good for images with irregular

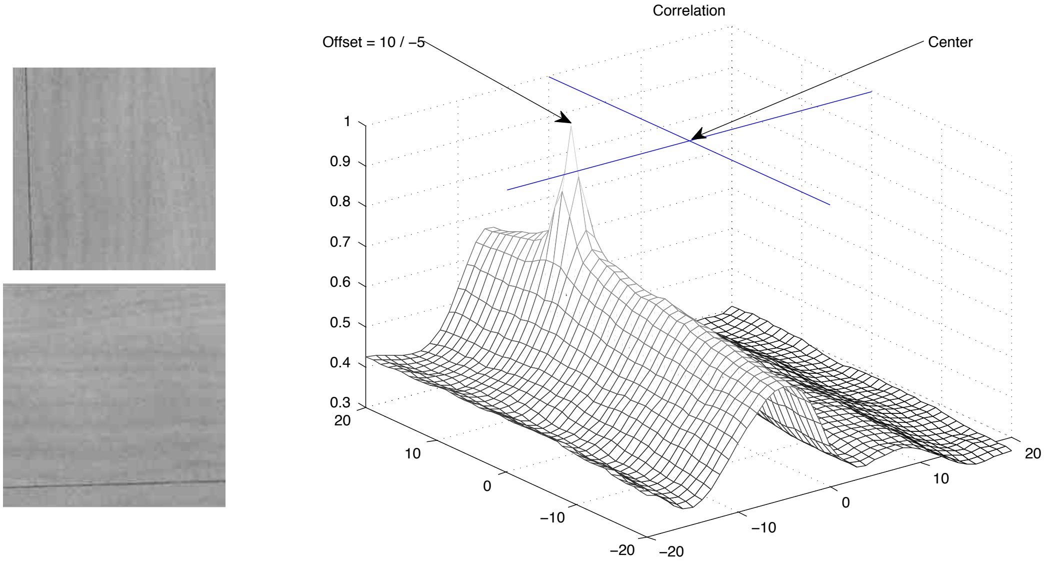


Figure 3.11: Example of correlation between the top left and bottom left image. The bottom image is a copy of the top image, shifted by an offset of (10px, -5px).

distinctive structure. However, when the texture is smooth and homogeneous, the row- and column histogram may become uniform and thus, it is not possible to calculate reliable results in such cases.

3.6.2 Error Metric Based

This method takes one image $I(x, y)$ and a template $t(x, y)$. In the context of this thesis, the template will be the current (n-th) frame and the image will be the previous ($(n-1)$-th) frame. The template is laid over the image and for every possible alignment and the error is calculated. Finally, the alignment with the minimal error is selected. In general, any error metric could be used, for example **SSD**, but **cross-correlation** or more specific **NCC** are most commonly used.

In [20], the formula for calculating the NCC is given for an image I of size $M_x \times M_y$ and template t of size $N_x \times N_y$ with

$$\gamma(u, y) = \frac{\sum_{x,y}(I(x, y) - \overline{I_{u,v}}) \cdot (t(x-u, y-v) - \bar{t})}{\sqrt{\sum_{x,y}(I(x, y) - \overline{I_{u,v}})^2 \sum_{x,y}(t(x-u, y-v) - \bar{t})^2}}$$

where $I(x, y)$ and $t(x, y)$ are the intensity values of the image I and template t at position (x, y) and $\gamma(u, v)$ is the value of the NCC at position (u, v). “Furthermore, $\overline{I_{u,v}}$ denotes the mean value of $I(x, y)$ in the area of the template t shifted to (u, v) which is caulcuated by" ([20])

$$\overline{I_{u,v}} = \frac{1}{N_x \cdot N_y} \sum_{x=u}^{u+N_x-1} \sum_{y=v}^{v+N_y-1} I(x, y)$$

$\bar{t}$ is the mean of the template t and can be calculated in a similar way.

In the context of this work, image I and template t are two consecutive image frames, and therefore $M_x = N_x$ and $M_y = N_y$.

To find the translation between these two images, it is again necessary to find the indices (u_{max}, v_{max}) where $\gamma(u, v)$ is maximal. Figure 3.11 shows an example. The function value $\gamma(u, v)$ is plotted for the NCC between the two images on the left, where the bottom image is just a shifted version of the top one. The peak denotes the value where the NCC is maximal and its offset from the center is equivalent to the translation between these images.

Obviously, this method is computationally more expensive than the histogram approach from the previous section. **Image Pyramids** (subsection 3.5.2) are often used to reduce the number of necessary calculations.

The precision of this method is expected to be much higher than the previously presented histogram method. It should even be possible to detect translations in fine textured images. However, [68] mentions, that the NCC is undefined for two images with zero variance and in general, "its performance degrades for noisy low-contrast regions" ([68]).

3.6.3 Phase Correlation

In its essence, an image is a set of intensity samples on a two dimensional plane. If these samples are interpreted as samples of a function $f(x, y)$, this function can also be transformed from the spatial domain into spectral domain. Even though the signal in the spatial domain is not a function of time, the spectrum is usually called *frequency domain*.

"Translation, rotation, and scale all have their counterpart in the Fourier Domain. Fourier methods differ from other registration strategies because they search for the optimal match according to information in the frequency domain" ([60]).

The advantage of using the frequency domain is, that it is possible to achieve much better resistance against noise than in the spatial domain [62].

The phase correlation method relies on the *shift property* of the Fourier transform. If an image f_2 is just a shifted version of image f_1

$$f_2(x, y) = f_1(x - x_0, y - y_0)$$

with displacement (x_0, y_0), then their Fourier transforms F_1 and F_2 are related by

$$F_2(\omega_x, \omega_y) = e^{-j2\pi(\omega_x x_0 + \omega_y y_0)} \cdot F_1(\omega_x, \omega_y)$$

This means, that the spectrum F_2 is just a phase-shifted version of the spectrum F_1 with the same magnitude. According to the *shift theorem* the phase difference can be expressed as

$$e^{-j2\pi(\omega_x dx + \omega_y dy)} = \frac{F_1(\omega_x, \omega_y) \cdot F_2^*(\omega_x, \omega_y)}{|F_1(\omega_x, \omega_y) \cdot F_2^*(\omega_x, \omega_y)|}$$

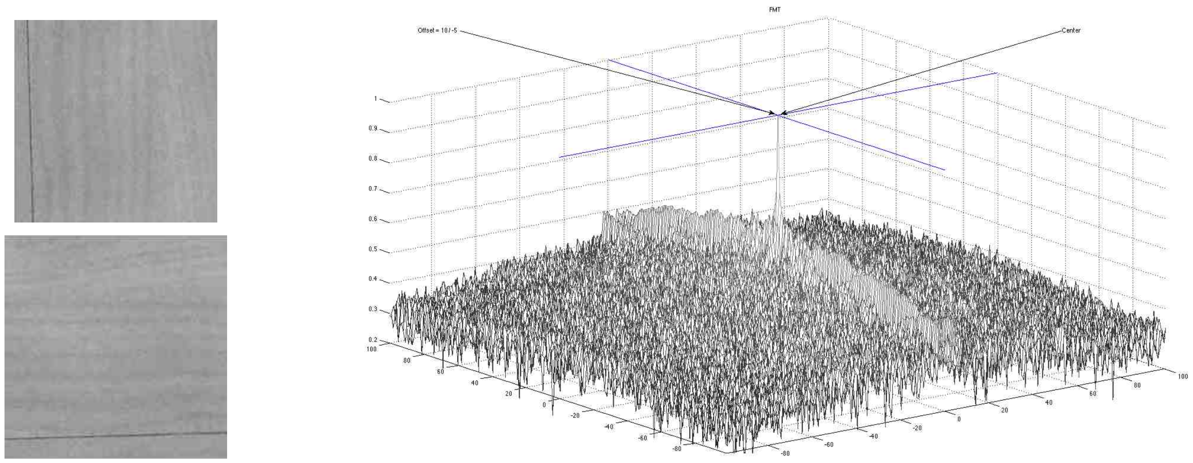


Figure 3.12: Example of phase correlation between the top left and bottom left image. The bottom image is a copy of the top image, shifted by an offset of (10px, -5px).

where * is the complex conjugate [60], [62]. This phase difference can be transformed back into the spatial domain with the inverse Fourier transform which will result in an image which isapproximately zero and only has one impulse located at a position that corresponds to the displace x_0, y_0 of the images. Therefore, this displacement can be found by determining the position of the peak in the resulting image.

Figure 3.12 shows an example of the phase correlation applied to the two images on the left, where once again, the bottom left image is a shifted version of the top left image. The right image shows the result of the process that was explained before where a peak at the position of the offset can be seen. In comparison to the **correlation** of the same images in Figure 3.11, the peak in the phase correlation is much clearer with a much sharper peak and a larger distance from the peak to the surrounding values.

Later in subsection 4.5.2 it will been claimed that the accuracy of algorithms is limited by the resolution of the camera or more specific by the physical size of pixels in the image. However, this is not always true as [29] presents an extension to the phase correlation algorithm that achieves even sub-pixel accuracy.

3.6.4 FMT

Another method operating in the frequency domain is the FMT. In contrast to the **phase correlation**, the FMT can also calculate the rotation and even scale.

Additionally to the Fourier *shift property* that was used for the **phase correlation**, the *rotation property* needs to be addressed:

$$F_2(\omega_x, \omega_y) = e^{-j2\pi(\omega_x x_0 + \omega_y y_0)} \cdot F_1(\omega_x \cos\theta_0 + \omega_y \sin\theta_0, -\omega_x \sin\theta_0 + \omega_y \cos\theta_0)$$

so now, the magnitudes of F_1 and F_2 are not equal anymore, but

$$|F_2(\omega_x, \omega_y)| = |F_1(\omega_x \cos\theta_0 + \omega_y \sin\theta_0, -\omega_x \sin\theta_0 + \omega_y \cos\theta_0)|$$

This relation shows, that the magnitude of F_2 is a rotated version of the magnitude of F_1 [60].

A difference in scale is related by the Fourier *scale property*. But since scale is not considered in the scope of this work, it is not further mentioned here. Details about scale calculation can be found in [60].

Because the phase spectrum is not affected by the rotation, the translation can still be calculated from the phase spectrum via **phase correlation**.

In [60], the rotation of the magnitude spectrum is calculated in a few additional steps, all of them are already familiar at this point:

1. Transform the magnitude spectra into polar (or log-polar, if scale is also considered) representation. As explained in section 3.5.3, the rotation in the magnitude spectra is now transformed to a translation in the polar magnitude spectra
2. Calculate the polar translation via **phase correlation** on the polar magnitude spectra
3. Calculate the rotation of the original images from the polar translation.

The problem with this method is the same that was already mentioned in section 3.5.3: the center of rotation needs to be known for the polar transform. Nonetheless, this method has been used for *visual vehicle egomotion estimation* in [31]. More details about the similarities and differences of their work to this one can be found in subsection 3.1.2.

3.6.5 Feature based

All previously mentioned methods belong to the class of **intensity based** methods where all pixels in the images are used for calculation. In contrast, **feature based** methods first extract features from the images and then work with them.

The basic steps are usually the same

- **Feature Detection:** search for stable features that can be found and identified in other images as well with high probability.
- **Feature Description:** in preparation for the next step, every detected feature has to be described in a way that allows it to compare features and to recognize the same feature in different images.
- **Feature Matching:** for every feature in an image, try to find the same feature in the other image and establish a relation.
- **Transformation Estimation:** from the correspondences, a transformation that maps the points in one image to their corresponding points in the second image is estimated. (see section 4.2)

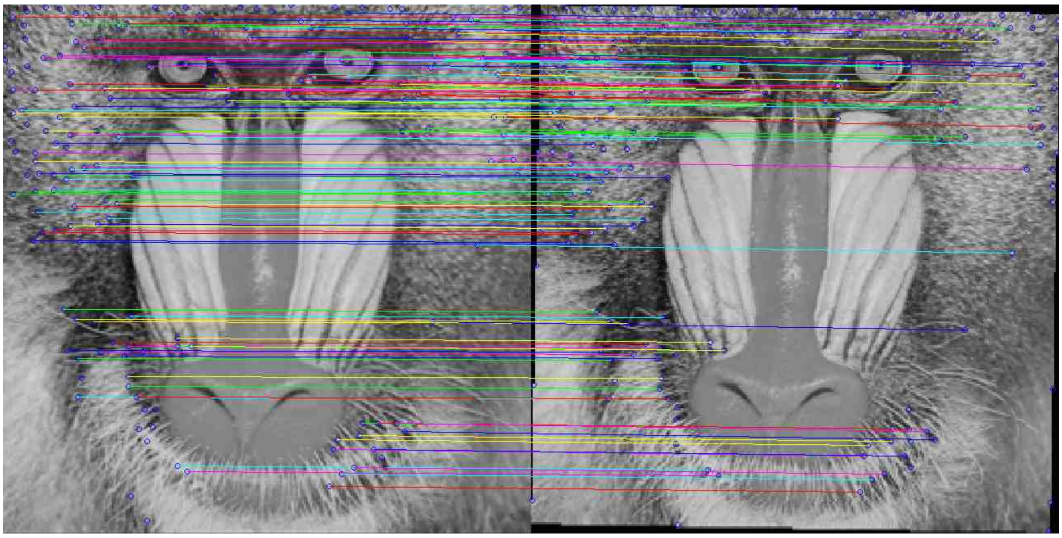


Figure 3.13: An example of feature matching, the right image is a rotated and translated version of the left one. The threshold for the descriptor distance is set very low to only include very good matches, thus reducing the number of features for better visibility

Features (also called **keypoints** or **points of interest**) in a general computer vision context can be “significant regions (forests, lakes, fields), lines (region boundaries, coastlines, roads, rivers) or points (region corners, line intersections, points on curves with high curvature)” ([78]). In the context of this work, mostly points are relevant.

As of today, there are numerous feature detection algorithms available, starting from the early *Moravec corner detection* [57], the classic *Harris Corner Detector* [35] to mordern methods like *SIFT* [52]. An extensive overview can be found in [70].

One important characteristic of a feature detection algorithm is the *repeatability* and it is *invariance* to geometric transforms. For example, **rotation invariance** means, that the same feature can be found even in a rotated version of the image, which is an important characteristic in the context of this work, whereas **scale invariance** - the ability to detect the same feature at a different scale is not so important.

The detected features in the reference image have to be matched against the detected features in the other image. In order to be able to identify a feature and match it against another one, features have to be described in some way. This is done via a so called **descriptor**. For images that are only related by translation, simple error metrics like *SSD* or *NCC* of the pixel-intensity values can be used on small image patches around the feature [69], [56]. However, in most cases there will be additional transformation, like rotation or even perspective distortions. To still be able to match features, more robust descriptors need to be used, for example *MOPS* [21] or *SIFT* [52], or one of the many more that can be found in [69] and [70].

Next, the features need to be matched to find correspondences and once again, there is a wide range of possible solutions. Szeliski further divides this step into two separate components. “The first is to select a *matching strategy*, which determines which correspondences are passed on to the next stage for further processing. The second is to devise efficient *data structures* and *algorithms* to perform this matching as quickly as possible." ([69]). If the feature descriptor is designed appropriately and is represented as vector, the euclidean distance between two features can directly be used to rate the matching between features. Due to the transformation, it is possible that not every feature from the reference image is visible in the other image as well. Thus it is not guaranteed that every feature has a match, and due to noise and imperfections in the descriptor, features might be matched wrongly. The most easy way to deal with this is to take the closest match for every feature, but define a threshold and discard matches where the distance between descriptors exceeds that threshold. In Figure 3.13, a matching of features from one image to a rotated and translated version of the same image is demonstrated. For more elaborate methods, have a look at [69], [70] or [78].

The final step, called **transformation estimation**, is not different from the other algorithms presented in this section and is described in section 4.2.

3.6.6 Optical Flow

According to [69], optical flow is the most general version of motion estimation where motion estimates are computed for each pixel to create a dense motion field. There are *local* methods based on image patches that usually involve Taylor series expansion of the *image displacement function*

$$E_{LK-SSD}(\mathbf{u}+\Delta\mathbf{u}) = \sum_i [\mathbf{J_1}(\mathbf{x_i}+\mathbf{u})\Delta\mathbf{u} + e_i]^2$$

where

$$\mathbf{J_1}(\mathbf{x_i}+\mathbf{u}) = \nabla I_1(\mathbf{x_i}+\mathbf{u}) = (\frac{\delta I_1}{\delta x}, \frac{\delta I_1}{\delta y})(\mathbf{x_i}+\mathbf{u})$$

is the *image gradient* or *Jacobian* at $(\mathbf{x_i}+\mathbf{u})$ and

$$e_i = I_1(\mathbf{x_i}+\mathbf{u}) - I_0(\mathbf{x_i})$$

is the current intensity error [69]. An example of this method is the Lucas-Kanade method presented in [53]. Alternatively, there are also methods that work globally by trying to minimize a global energy function, a classic example being the method of Horn and Schunck [37]. In [74] both, local and global methods are combined to calculate dense motion fields that are more robust against noise.

An example of an application of the optical flow method is displayed in Figure 3.14, where two consecutive frames of a traffic surveillance camera are displayed. The cars have continued their motion between the frames, the bottom left shows the movement of pixels in a color coded way, where the color represents the direction and the saturation the velocity. The bottom right image explicitly displays arrows for movement vectors.

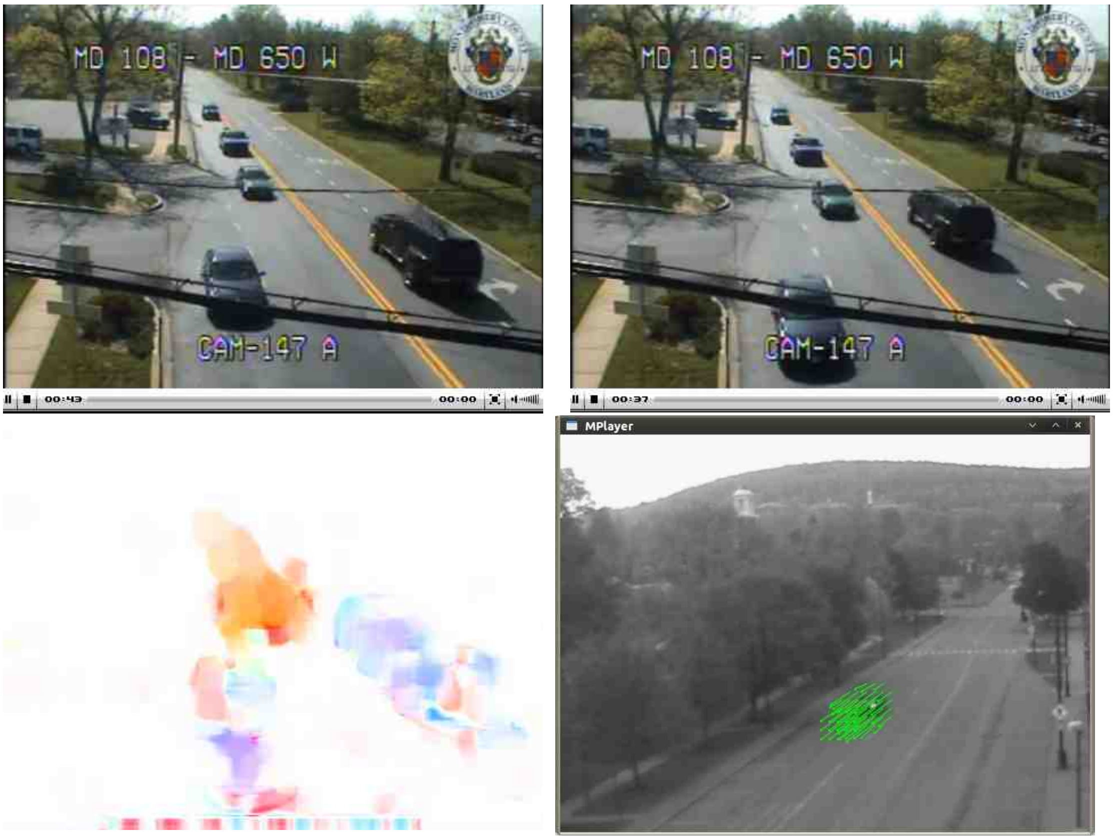


Figure 3.14: Examples of Optical Flow. The top left and right image are two consecutive frames of a video with moving cars. The bottom left image shows the optical flow with color coded movement directions. The bottom right image shows a similar scene where the optical flow is visualized with vectors.

CHAPTER 4

Optical Motion Detection Framework

4.1 Overview

The goal of this work was to construct a motion sensor based on visual methods as an alternative approach to traditional speed sensors that are based on wheel rotation. The use of different methods is motivated by the desire to avoid systematic errors of a particular method by using several diverse methods as explained in subsection 2.2.1. In the case of wheel based sensors, the systematic error that should be avoided is *wheel slippage*, which results in situations where the wheel rotation does not correspond correctly with the actual speed of the vehicle. A flat tire would be another scenario in which wheel base methods fail by delivering wrong sensor readings.

The new system consists of a video camera mounted on the vehicle for which the motion should be measured, pointing perpendicular to the ground, and a processing unit, executing a suitable algorithm which is part of this work as well. While the vehicle is moving, the camera produces a video stream of the ground over which the vehicle is moving, consisting of a sequence of single image frames. Every pair of consecutive images is related with its predecessor and successor via a transformation determined by the change in position or viewpoint from where the image is taken. Thus by the position of the vehicle in space. As long as these image frames have sufficient overlap, it is possible to calculate this transformation between consecutive images, and consequently to conclude the difference in position from where they have been taken. This problem is usually referred to as **image registration** [75]. From this difference in position, the movement speed and direction of the vehicle can be concluded, as well as the derived values such as acceleration and turning rates.

An image based sensor can furthermore produce additional measurements exceeding the possibilities of a wheel based method, like confidence values from analysis of the surface in the image. These analysis results could be used to adapt the method to the current circumstances as well. These aspects will be explained in more detail later in this chapter.

Rotation from multiple correspondences

If the camera is allowed to move freely around, the transformation between frames can become of high order (see Table 3.1) and the solution of the image registration problem becomes quite hard. By mounting the camera to the vehicle in a fixed known distance from the ground pointing perpendicular at the ground, the transformation order is restricted. To be precise, only **translation** in x- and y direction and **rotation** are allowed which corresponds to an **euclidean** transform. This limitation simplifies the problem of finding the transformation, though the addition of possible rotation already poses a significant increase in complexity compared to other similar problems like video stabilization (see Section 3.1.1) or optical mouse (See section 3.1.1).

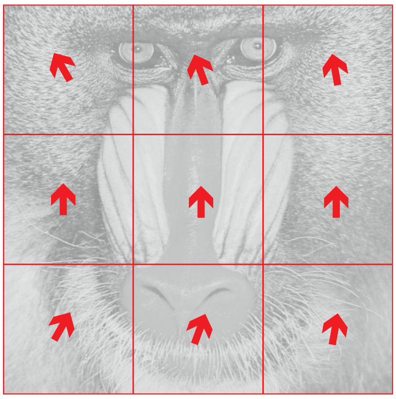


Figure 4.1: Basic concept of using multiple image patches / tiles for image registration

The proposed system solves the rotation problem with an alternative approach. Many correspondences of the form $((x, y), (x', y'))$, where both points (x, y) and (x', y') denote the same point in different images, are calculated. A transformation can be derived from these correspondences. Figure 4.1 illustrates the principal idea of dividing the image into patches or tiles and calculating the translation for each patch separately. The region of the patch in the original image will be called **Region of Interest (ROI)** in this work. The arrows indicate the movement of each local patch. Looking at the image as a whole, the arrows already suggest a global transformation including rotation around a center that lies outside of the image to the left. What the human brain does intuitively from such a set of arrows can also be calculated formally. The necessary mathematical models are presented in Section 4.2.

If the tiles are relatively small compared to the amount of change caused by the rotation, it is possible to use methods that only detect translation without their result being distorted by the rotation. This opens up the possibility to use methods that are simpler and can be performed with less computational effort. Moreover, since it is not necessary that the tiles cover the whole image, the number of pixels for the calculation could further be reduced.

Alternatively, some methods inherently produce point correspondences as output like for example feature based methods or optical flow calculation.

```
input  : An image frame
output: Movement parameters
1  for ever do
2      frame ←getFrame();
3      rois ←splitIntoRois(frame);
4      foreach (roi,previousRoi)in (rois, previousRois) do
5          // Actually, not every tuple (roi, previousRoi) is enumerated, only every pair of
           corresponding ROIs;
6          roiTranslations.append(calculateTranslation(roi, previousRoi));
7      end
8      previousRois ←rois;
9      movementParameters ←calculateMovementFromROIS(roiTranslations);
10     confidence ←calculateConfidenceFromROIS(roiTranslations);
11     publishResult(movementParameters, confidence);
12 end
```

Algorithm 4.1: High-Level description of main runloop

Sensor Architecture

The architecture of the sensor is based around the method of using multiple correspondences. Image frames that are sent from the camera to the sensor are segmented into several ROIs. For each ROI correspondence is calculated with one of the implemented image processing algorithms from Section 3.6. From these correspondences, the homography is calculated as well as a confidence value.

Algorithm 4.1 shows a high level description of the main calculation loop. For each frame, one iteration of the loop has to be performed, producing one measurement for each frame. The single steps can be separated relatively easily as long as their interfaces are well defined and stable. Especially, the calculation for a single ROI in *line 6*, the movement calculation in *line 9* and the confidence calculation in *line 10* are eligible for separation which allows to exchange and combine different methods for these processing steps.

It has already been shown in section 3.6 that there are various image processing methods that can be used. It will further be shown that for the *Homography Estimation* and the *Confidence Estimation*, several different algorithms can be used as well (see section 4.2 and section 4.3).

Naturally, a few questions arise:

- Which method works best for the different tasks of calculating local correspondences, a global transformation and confidence values?
- Does a method always work well?
- If not, under which circumstance does a method work well and can this knowledge be used to improve the results?
- How should the methods for these different tasks be combined? Do they interfere with each other or can they be combined arbitrarily?

These questions are answered by experiments in chapter 6.

To be able to experiment with different configurations, the software architecture needs to allow easy reconfiguration of single parts of the calculation. This is not only advantageous for experiments, the exact same structure can later be utilized to dynamically adapt the system to changing conditions, see section 4.4. It has already been mentioned, that the computation steps in algorithm 4.1 are clearly separated from each other. In section 5.5 it is explained, how the implemented software architecture meets these requirements.

4.2 Homography Estimation

A homography is defined as "an invertible mapping h from (the *projective space*) $\mathbb{P}^2$ to itself such that three points x_n, x_2 and x_3 lie on the same line if and only if $h(x_n), h(x_2)$ and $h(x_3)$ do." ([36]). The names *projectivity* and *projective transformation* are used synonymous. The following theorem from [36] shows, how the homography can be represented in a homography matrix, and how this homography matrix relates the source and target points to each other:

Theorem. *A mapping from* $\mathbb{P}^2 \rightarrow \mathbb{P}^2$ *is a projectivity if and only if there exists a non-singular* 3×3 *matrix* $\mathbf{H}$ *s.t. for any point in* $\mathbb{P}^2$ *represented by a vector* $\mathbf{x}$ *it is true that its mapped point equals* $\mathbf{Hx}$

When dealing with 2D-images, the image-points (x, y) need to be converted into so called *homogeneous* coordinates x_n, x_2, x_3 where $x = \frac{x_n}{x_3}$ and $y = \frac{x_2}{x_3}$.

One property of the homography matrix $\mathbf{H}$ is, that it can be multiplied by any constant factor without change to the actual transformation. For this reason, $\mathbf{H}$ only has 8 degrees of freedom and 8 unknown values that need to be solved for, even though it has 9 elements [26]. Table 3.1 in subsection 3.5.1 already presented the transformation matrices for different geometric transformations with the projective transformation being the represented by the homography matrix and the other transformations being constraint variants thereof.

The calculation produces a set of point correspondences of the form $(x_n, y_n) \rightarrow (x'_n, y'_n)$. They are converted to homogeneous coordinates $(x_n, y_n, 1)^T \rightarrow (x'_n, y'_n, 1)^T$. The problem is to find the homography matrix $\mathbf{H}$ s.t.

$$\forall n : \begin{pmatrix} x'_n \\ y'_n \\ 1 \end{pmatrix} = \mathbf{H} \begin{pmatrix} x_n \\ y_n \\ 1 \end{pmatrix}$$

There are two basic approaches to solves this problem: calculating an exact solution and calculating an approximate solution. The first approach only takes the minimum required number of correspondences and calculates the solution from those. The problem with that approach is, that it's error prone when dealing with real world data where the accuracy of the correspondences is affected by noise and measurement inaccuracies. Approximation methods take a bigger number of correspondences as input and produce a solution that best fits all the points at once, which is better suited for real world applications. However, there are also ways to use exact solutions, for example when combined with Random Sample Consensus (RANSAC).

One example of each approach will be represented, additional methods can be found in [40] and [26].

4.2.1 Extracting Results and Relation to Physical Parameters

Recall the form of the transformation matrix from subsection 3.5.1 for rigid transformations:

$$[\,\mathbf{R} \mid \mathbf{t}\,]_{2x3}$$

As explained in subsection 3.5.1, this matrix can be extended into a full homography matrix by appending a third row of $0, 0, 1$ which leads to the following matrix where all elements are explicitly named:

$$\mathbf{H} = \begin{pmatrix} \cos\varphi & -\sin\varphi & t_x \\ \sin\varphi & \cos\varphi & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

It can be seen, that the necessary measurement results t_x, t_y (translation in x- and y direction) and φ (rotation angle) can be directly taken from the calculated homography:

$$t_x = \mathbf{H}_{3,1}$$

$$t_y = \mathbf{H}_{3,2}$$

$$\varphi = \cos^{-1}\mathbf{H}_{1,1}$$

If the sensor is not mounted directly in the center of the vehicle, the rotation causes deviations of the measured translation values to the real translation values. The situation is depicted in Figure 4.2. This figure shows a pure translation by the value of φ. If the sensor is positioned with an offset of R_y from the center, this rotation causes additional translations Δx and Δy at the position of the sensor:

$$\Delta x = R_y \cdot \sin(\varphi)$$

$$\Delta y = R_y \cdot \cos(\varphi)$$

The measurements for t_x and t_y have to be corrected. Assuming the sensor has an offset only in the movement direction and is in the lateral center of the vehicle:

$$t_{x_{\text{corrected}}} = t_x + R_y \cdot \sin\varphi$$

$$t_{y_{\text{corrected}}} = t_y + R_y \cdot (1 - \cos\varphi)$$

However, obviously the translation values t_x and t_y have been calculated in the unit of **pixels**. Figure 4.3 displays the relation of physical lengths to pixel distance for one dimension in a two dimensional example. Basically, a length with a physical dimension is mapped onto an image with pixel dimension. The relation between these is given by a scale factor that depends on the cameras angle of view φ and the distance d of the lens to the image plane (the ground).

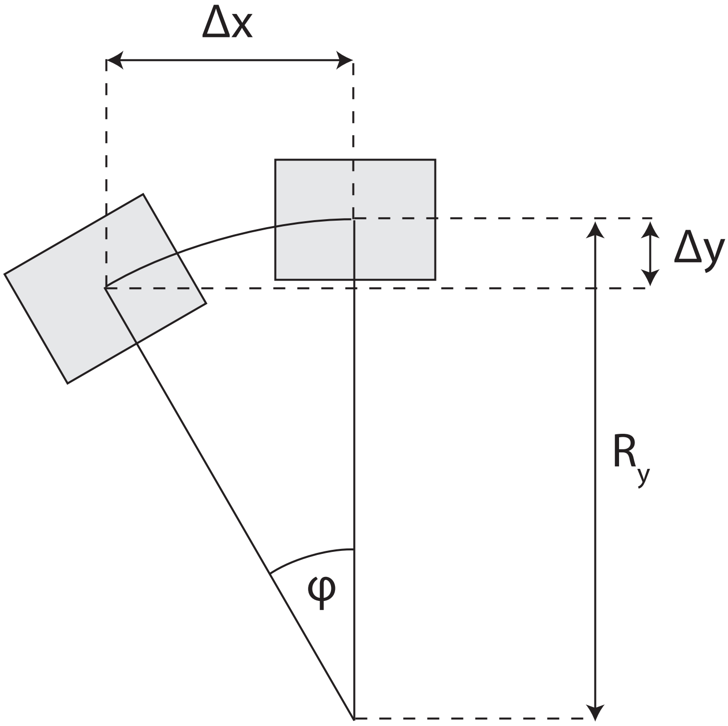


Figure 4.2: Correcting Rotation

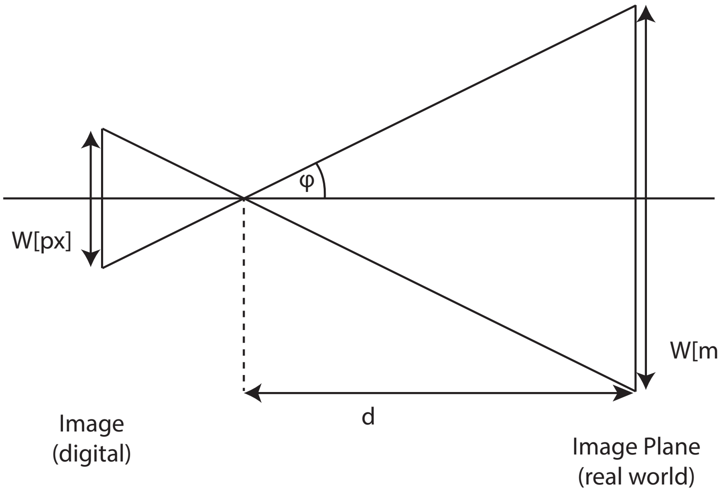


Figure 4.3: Mapping from pixel-distance to physical length

$$W[m] = 2 \cdot d \cdot \tan\varphi \ \widehat{=} \ W[px]$$

The width in pixel units is constant and known from the cameras resolution, therefore the scale factor s_x can be calculated from

$$s_x = \frac{W[m]}{W[px]} = \frac{2 \cdot d \cdot \tan\varphi}{W[px]}$$

The scale factor s_y for the height can be calculated analogous. Unfortunately, the cameras angle of view φ is often not known and needs to be measured. But then it is easier to directly determine the scale factor by either measuring the physical dimensions of the field of view or by taking a picture of a calibration pattern with structures of known physical size. It is important for both options, that the distance is equal to the distance from the ground at which the camera will be mounted. This can be done during camera calibration (see subsection 3.5.4). Furthermore, for correctly calibrated cameras the scale factors s_x and s_y should be identical, so that the scale factor s can be used.

Furthermore, the time between two measurements needs to be known. This is given by the framerate of the camera:

$$\Delta t = \frac{1}{f_{\text{camera}}}$$

The speed in m/s can then be calculated with the following formulas:

$$v_x[m/s] = t_x \cdot s_x \cdot f_{\text{camera}} = t_x \cdot s \cdot f_{\text{camera}}$$

$$v_y[m/s] = t_y \cdot s_y \cdot f_{\text{camera}} = t_y \cdot s \cdot f_{\text{camera}}$$

Further measurements like the acceleration can then be derived as usual:

$$a[m/s^2] = \frac{\Delta v[m/s]}{\Delta t} = \Delta v[m/s] \cdot f_{\text{camera}}$$

4.2.2 Homography Estimation Methods

Exact Homography

It has already been established that for this work, only *isometry* transformations are considered. Therefore exactly **two** correspondences are needed. From these two correspondences, the homography is calculated with the following formula from [40]. One of the two correspondences is selected as a pivot element (x_n, y_n) and (x'_n, y'_n):

$$H_e = A_b \cdot R_{\hat{\Theta}} \cdot A_0$$

with

$$\mathbf{A_b} = \begin{pmatrix} 1 & 0 & x'_n \\ 0 & 1 & y'_n \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{A_0} = \begin{pmatrix} 1 & 0 & -x_n \\ 0 & 1 & -y_n \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{R}_{\hat{\Theta}} = \begin{pmatrix} R(\hat{\Theta}) & \mathbf{0} \\ \mathbf{0}^T & 1 \end{pmatrix}, \quad \mathbf{R}(\hat{\Theta}) = \begin{pmatrix} \cos(\hat{\Theta}) & -\sin(\hat{\Theta}) \\ \sin(\hat{\Theta}) & \cos(\hat{\Theta}) \end{pmatrix}$$

and

$$\hat{\Theta} = \Theta' - \Theta$$

Obviously, by restricting the transformation to *isometry*, the formula is simplified significantly. For a general solution and different methods have a look at [40] and [26].

Least Squares DLT (Direct Linear Transform)

This approximation method calculates the least squares solution from all correspondences as described in [40]. Again, because the homography is limited to isometry, a simplified form can be used. For more general solutions and alternative methods, consult [40] and [26].

$$H = \begin{pmatrix} a & -b & c \\ b & a & d \\ 0 & 0 & 1 \end{pmatrix}$$

The parameters a, b, c, d can be calculated by solving the following system:

$$\begin{pmatrix} -w'_i \cdot y_i & -w'_i \cdot x_i & 0 & -w'_i \cdot w_i \\ w'_i \cdot x_i & -w'_i \cdot y_i & w'_i \cdot w_i & 0 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} -y'_i \cdot w_i \\ x'_i \cdot w_i \end{pmatrix}$$

where the point correspondences need to be in homogeneous coordinates (x_i, y_i, w_i).

4.2.3 Removing Outliers

It's the nature of real world applications, that measurements are not always perfectly precise and that distortions and noise can cause erroneous measurements. These erroneous measurements lead to erroneous correspondences or so called **outliers**. Outliers affect the precision of the resulting Homography Estimation, how server the impact is depends on the specific method used for the Homography Estimation.

There are methods to detect outliers that can be applied to remove them from the dataset of correspondences before the Homography Estimation calculation, so that resulting homography is not affected by these outliers.

Correspondence Consensus

This method is based on the assumption that all single correspondences *agree* with the result i.e. if the resulting homography is applied to a correspondence, the error to the actual result should be minimal.

First, the homography $\mathbf{H_n}$ for the frame n is calculated from all correspondences.

$$\mathbf{H_n} = F(\mathbf{c_n})$$

where F is any Homography Estimation method and $\mathbf{c_n}$ is the set of all correspondences for frame n.

Then, the set of outliers $\mathbf{c_{n_o}}$ is calculated with

$$\mathbf{c_{n_o}} = \{c_{n_i} | (| \begin{pmatrix} x_{n'_i} \\ y_{n'_i} \\ 1 \end{pmatrix} - \mathbf{H_n} \begin{pmatrix} x_{n_i} \\ y_{n_i} \\ 1 \end{pmatrix} | > T)\}$$

and contains all correspondences, where the distance of the correspondence target (x'_i, y'_i) to the correspondence source (x_i, y_i) that has been transformed by the calculated Homography $\mathbf{H_n}$ is greater than a threshold T.

Finally, the homography $\mathbf{H}$ is calculated again from the correspondences **without** outliers:

$$\mathbf{H} = F(\mathbf{c_n} \setminus \mathbf{c_{n_o}})$$

Unfortunately, the underlying assumption that the relative motion is the same for every correspondence is only true for **euclidean transforms** (only translation in x- and y direction). As soon as rotation is allowed, this assumption does not hold anymore and this method can not be applied.

SMC

In [16], a *Smoothness Motion Constraint* is developed with which outliers can be detected and removed very effectively.

This method is based on the assumption that the observer (e.g. the vehicle) underlies physical constraints and has some inertia that prevents it from instantly changing its movement. Much rather, the movement direction and speed can only change at a certain rate which at a sufficient frame rate yields a *smooth motion* between consecutive frames.

Based on this assumption, every correspondence can be checked and rejected, if the motion differs too much from the previous motion.

The process is similar to the **Correspondence Consensus** method can be used, but instead of calculating $\mathbf{H_n}$ from all correspondences, the homography $\mathbf{H_{n-1}}$ from the previous frame is used instead.

$$\mathbf{c_{n_o}} = \{\mathbf{c_{n_i}} | (| \begin{pmatrix} x'_{n_i} \\ y'_{n_i} \\ 1 \end{pmatrix} - \mathbf{H_{n-1}} \begin{pmatrix} x_{n_i} \\ y_{n_i} \\ 1 \end{pmatrix} | > T)\}$$

and

$$\mathbf{H_n} = F(\mathbf{c_n} \setminus \mathbf{c_{n_o}})$$

where the subscript $_n$ denotes values from the current frame and the subscript $_{n-1}$ denotes the previous frame.

As will be explained in the next section, the threshold T can be choosen to correspond to the physical limits of the vehicle to discard only those correspondences as outliers, that are physically not possible to be correct.

Not only does this method reduce computational cost compared to the **Correspondence Consensus** method but more importantly, since it is based on the previous movement instead of a constant assumption, it is also applicable when rotation and even higher order transforms are allowed.

4.3 Confidence Estimation

The confidence is a value that reflects, how accurate a measurement is estimated to be. Especially for the use in a sensor fusion system (section 2.3), a metric is required to reason about the accuracy and validity of incoming measurements. Conventional methods involve knowledge available at design time, like ranges of speed where a certain sensor is known to perform well or ranges where it is known to not work good. Using different sensors that work well at different ranges and weighting their results with the this knowledge based on the current speed measurement can result in accurate measurements over wider ranges than with single sensors.

One major advantage and novelty of the system proposed in this thesis is, that it inherently produces a variety of values like statistical data of the current image that can be used to reason about the quality of the measurement and calculate a confidence value.

First and foremost, the image processing methods often produce various parameters like number of detected features or quality of matches that can be used to estimate their current performance. Such calculations are specific to the concrete image processing algorithm. Some examples are:

4.3.1 Normalized Cross-Correlation (NCC)

The method based on NCC as presented in subsection 3.6.2 calculates a value $I(x, y)$ of the image with an overlaid and transposed version of the image for each possible offset (x, y) . The resulting movement is the offset (x_i, y_i) where $I(x_i, y_i)$ is maximal.

Due to the normalization, $I(x, y) \ni [0, 1]$ is always a value between 0 and 1 where 1 is equivalent to a perfect match and 0 is equivalent to no match at all. This means, that the value $I(x, y)$ can directly be used as a confidence estimation. Other error metrics than NCC can also be used, but would need to be normalized to $[0, 1]$.

4.3.2 Feature-Based Method

In the feature matching step of feature based methods (see subsection 3.6.5), it is necessary to calculate a score in order to match the features. One simple method is to calculate the euclidean distance of the descriptors. This distance can be used to calculate the confidence. If the distance is 0, the matching is perfect and the confidence is high. The greater the distance, the higher the matching error e_{matching}, and therefore, the confidence becomes lower. Usually, there is a threshold k on the error. Matchings with an error greater than this threshold k are discarded as **outliers**. The confidence function is a linear function:

$$c = \begin{cases} 1 - \frac{e_{\text{matching}}}{k} & e < k \\ 0 & e \geq k \end{cases}$$

This function is depicted in Figure 4.4.

Furthermore, there are generic measurements that do not depend on a specific implementation. These are calculated on frame level from information from the tiles like the number of utilized tiles, consistency of a measurement with a prediction model (based on previous measurements) or knowledge about the limits of the system itself.

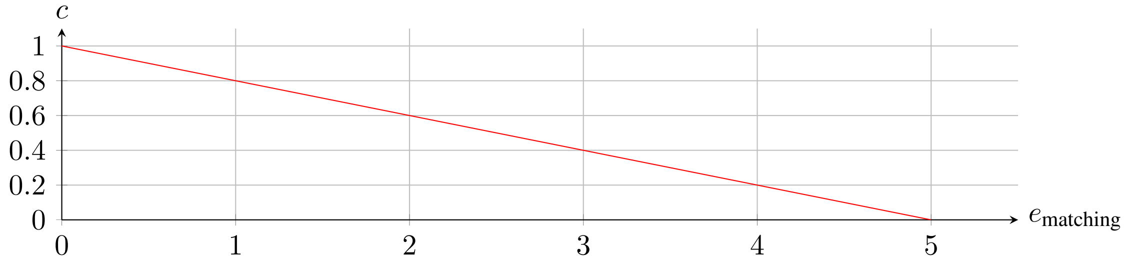


Figure 4.4: Confidence-Function for Feature-Matching Distance: $c = 1 - (e_{\text{matching}}/k)$ with $k = 5$

4.3.3 Number of Correspondences

For feature based methods, the more features are detected, the more stable the solution is and thus, the higher the confidence is. Even for other methods with a fixed number of tiles, the number of correspondences can be less than the number of tiles when some correspondences have been discarded as outliers.

There is a minimum number of correspondences required for the Homography Estimation task. As explained in subsection 4.2.2, this number is 2 for *isometry* transformations.

If the **Exact Homography** method from subsection 4.2.2 is used which requires exactly 2 correspondences, a confidence can be calculated as a simple binary value of wether the necessary 2 correspondences are available or not:

$$c = \begin{cases} 0 & N_{\text{correspondences}} < 2 \\ 1 & \text{else} \end{cases}$$

If the **Direct linear transform (DLT)** method from subsection 4.2.2 is used, a higher number of correspondences means a more robust solution and, thus, a higher confidence. However, the higher the number of correspondences gets, the smaller the improvement becomes.

A function has been chosen to weight the number of correspondences into the confidence value with

$$c = 1 - e^{\frac{x-1}{2}}$$

which is also depicted in Figure 4.5. The parameters have been chosen such that the confidence quickly approaches 1 as soon as the number of correspondences becomes greater than 6 which is three times the minimum required number of correspondences.

The parameters have been chosen because in experiments a further increase in correspondences did not increase the accuracy significantly.

4.3.4 Prediction and Physical Model Consensus

In subsection 4.2.3, SMC has been introduced as a method to remove outliers. It works by determining the distance from the actual measurements to a prediction based on the previous

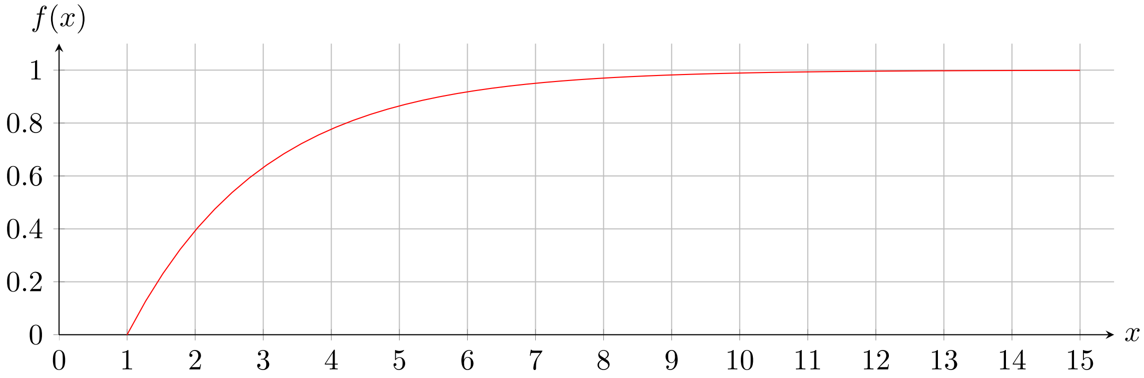


Figure 4.5: Confidence-Function for Number of Tiles: $c = 1 - e^{\frac{x-1}{2}}$

measurement because it is assumed that the speed and direction of the vehicle can not change instantly. It is expected, that there is a small error from the prediction to the actual measurements. However, the magnitude of the error is limited by the physical characteristics of the vehicle i.e. its maximal acceleration.

To recap, a correspondence is a tuple $(\mathbf{p_i}, \mathbf{p_i'})$ with $\mathbf{p_i}$ as the source point and $\mathbf{p_i'}$ as the target point. Their relation is defined by the current homography H_n. Based on the previous homography H_{n-1}, a second target point is calculated from the source:

$$\mathbf{p_i''} = H_{n-1}\,\mathbf{p_i}$$

which is the **predicted** target point. Now, the prediction error can be calculated as the distance from the predicted to the actual target point:

$$d_i = |\mathbf{p_i'} - \mathbf{p_i''}|$$

This is the same calculation as required for SMC and therefore, requires no additional computation. It is only necessary to save the error distance for all correspondences when calculating SMC.

This process calculates one prediction error distance d_i for every correspondence c_i which then have to be reduced into one single confidence value. In order to do so, first the average prediction error distance is calculated:

$$d = \sum_{i=1}^{N} \frac{d_i}{N}$$

From this average error distance, the confidence is calculated with the following formula:

$$c = \begin{cases} 1 - e^{(d-k)} & d < k \\ 0 & d \geq k \end{cases}$$

which models the fact that no change (= constant speed) or small changes (= small acceleration) are expected. As the acceleration approaches the threshold k, the confidence approaches 0. This function is depicted in Figure 4.6. The value of the threshold k should correspond to the maximal acceleration of the vehicle, see subsection 4.2.1 on how the pixel units can be related to physical units.

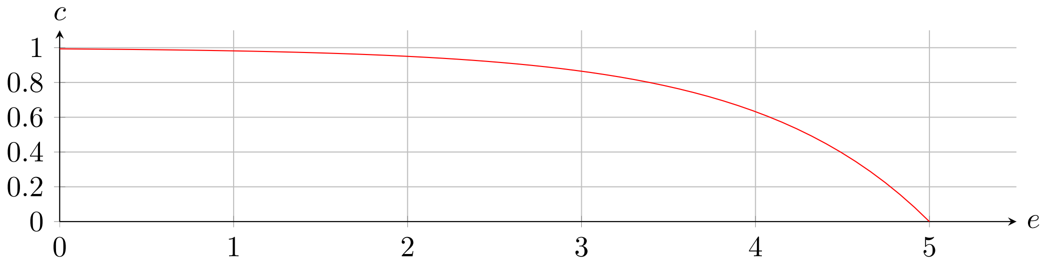


Figure 4.6: Confidence-Function for Prediction Error: $c = 1 - e^{(d-k)}$ with $k = 5$

The second limiting factor determined by physical properties of the system is the maximal speed that can be measured. This comes from the fact that there needs to be at least a certain amount of overlap between two frames. With respect to the confidence calculation, the movement distance is used in pixel units. In subsection 4.5.2 this will be further related to a physical unit as well. The confidence is calculated with the formula:

$$c = \begin{cases} 1 - e^{10(\frac{t}{D}-1)} & \frac{t}{D} < 1 \\ 0 & \frac{t}{D} \geq 1 \end{cases}$$

where t is the translation in x or y direction in pixels, and D is the corresponding dimension (either the width or the height of the ROI) in pixels. This function operates on the quotient $\frac{t}{D}$, which is the translation in x or y direction in relation to the width or height of the ROI. The function (plotted in Figure 4.7) has been chosen such that the confidence decreases quickly for overlaps smaller than 20%. This threshold value has proved to be appropriate as a common baseline for all implemented algorithms. Some algorithms may even need less overlay to work properly, therefore the parameters of the function could be adjusted to specific algorithms as well.

4.3.5 Combining multiple confidence values

For every frame, multiple individual confidence values are generated. Every single tile produces a confidence value and furthermore, there are additional global confidence values that have been just introduced in this section. Ultimately, all these individual confidences have to be combined into one single value. The simplest way to do this is by averaging all individual confidences:

$$c = \sum_{n=1}^{N} \frac{c_n}{N}$$

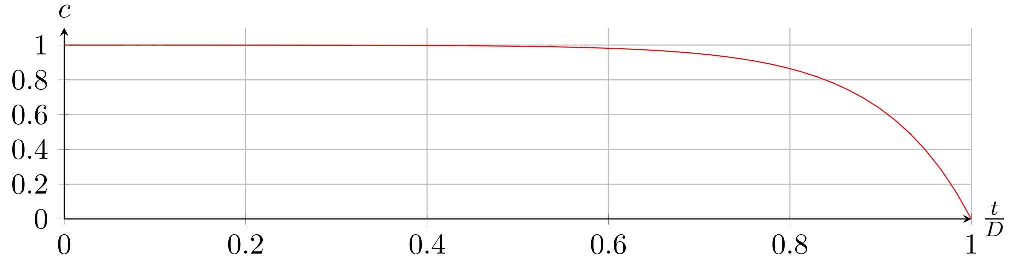


Figure 4.7: Confidence-Function for current Speed: $c = 1 - e^{10(\frac{t}{D}-1)}$

where N is the number of different confidences. More complex methods like weighted averages or complex filter methods are also possible.

4.4 Dynamic Adaptation

The calculation of the confidence value is based on a variety of information produced during normal operation of the sensor. The question aries, if this information can also be used to increase the confidence and precision by adapting the sensor to the current situation (e.g. current speed).

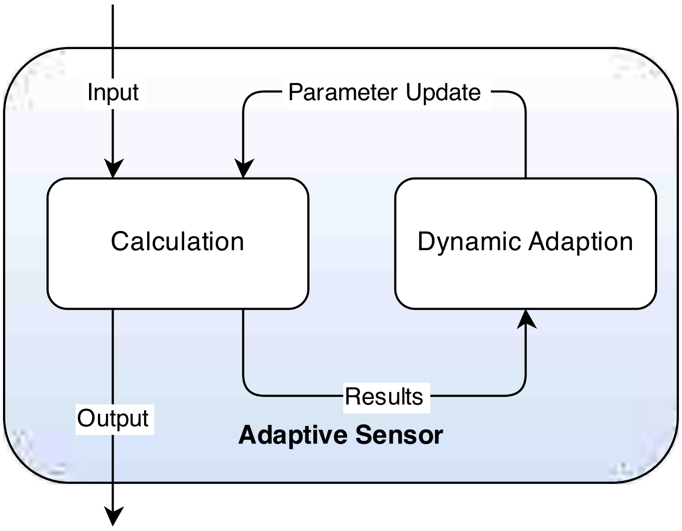


Figure 4.8: Dynamic Adaption Overview

In Figure 4.8, the general structure of an adaptive system is depicted. The first part is the calculation of an output (a measurement) from an input (the current image frame). Additionally, results of this calculation, not necessarily just the output, but also possibly intermediate results or other values derived from the input, are used to adapt the parameters of the calculation. This adaption is then fed back into the calculation to take effect. This adaption can be specific to the image processing methods like **ANMS** and it can also change global parameters of the calculation like **Adaptive Frame Size** or **Adaptive Frame Position**.

4.4.1 Adaptive non-maximal suppresion

Adapting a threshold is something commonly done. An example is the following method used for feature based methods. In a typically image, features are generally not evenly distributed, but rather clustered. For many operations an even distribution is necessary, or at least preferable. To solve this problem, the threshold that decides if a feature is significant enough to be included for further processing or rejected is not statically fixed. **Adaptive non-maximal suppresion (ANMS)** [21] is such a method, where the threshold is dynamically adapted to filter features that are not global maxima, but rather local maxima in a pixel radius. Thus distributing features more evenly over the image. Figure 4.9 shows a comparison between static thresholds at the top and ANMS-threshold at the bottom row.

4.4.2 Adaptive ROI Size

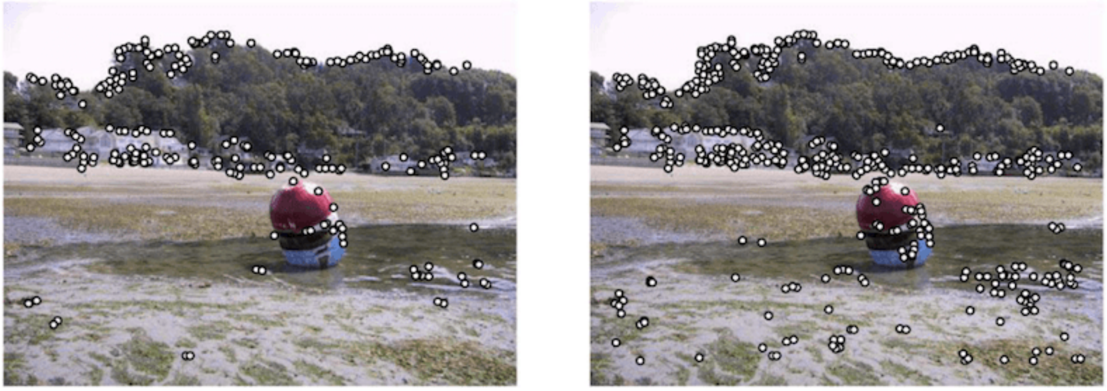
As will be explained in subsection 4.5.2, the maximal speed measurable by the sensor is determined by the size of the ROI which is used for calculation. Therefore a ROI should preferably be of large size. On the other hand, the computational complexity is in direct relation with the size of a ROI, thus, a ROI should not be excessively large. For most methods, a ROI does not need to be square though. To find a sensible configuration for the size, previous knowledge of the movement speed and direction can be used, to size and skew a ROI to yield maximal precision while maintaining roughly the same area and therefore keeping the computation time approximately constant while varying the width and height. Figure 4.10 visualizes the principle. Lets assume that the system has just been initialized before taking the first image for frame 1, where the ROI is square because initially, there is no speed information available. In frame 2, there is already a previous measurement available with movement speed and direction, therefore the ROI size can be adapted. The vehicle is moving to the right, therefore the ROI is strained on the y axis and clinched on the x axis.

A simple approach is to keep the *headroom* (bigger size than theoretically necessary) of the ROI constant as long as possible. This can be achieved by adding the current translation value to a base size:

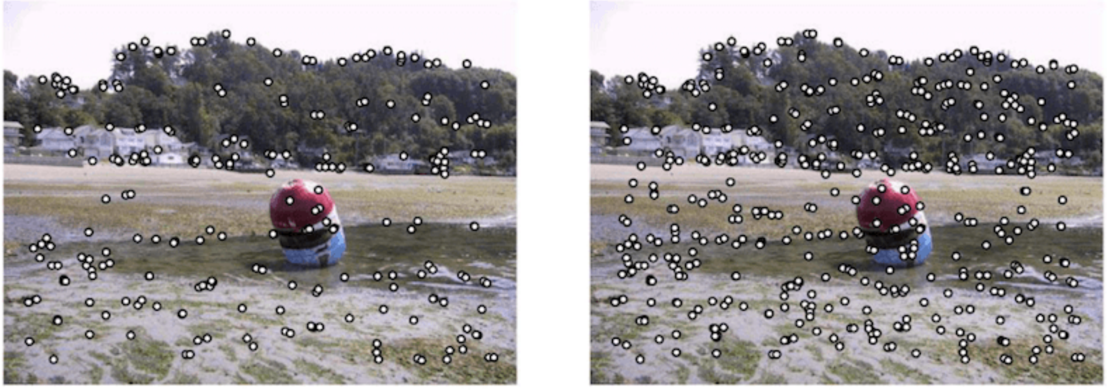
$$w = w_0 + |v_x|$$

$$h = h_0 + |v_y|$$

Figure 4.11 shows a plot of this adaption function for a base size of $d_0 = 20$. d stands for *dimension*, either w or h. Obviously, the size of the ROI is ultimately limited by the size of the whole image frame. This needs to be considered when implementing this strategy.



(a) Strongest 250 (b) Strongest 500



(c) ANMS 250, $r = 24$ (d) ANMS 500, $r = 16$

Figure 4.9: Adaptive non-maximal suppresion. In (a) and (b), features are selected by a static threshold on their corner strength. In (c) and (d), the ANMS adaptive threshold is used. [21]

4.4.3 Adaptive ROI Position

Similarly, the position of a ROI can be moved around within the frame. ROIs should preferably be positioned in regions of the frame, where Tile Algorithms can produce accurate results. Edges are a good indication of *'good'* ROI-positions in the frame. The following is a method which is very cheap with respect to computational complexity:

$$E = (I \otimes [-1, 1]) \circ (I \otimes [-1, 1]^T)$$

where I is the original image, filtered with the *kernels* $[-1, 1]$ and $[-1, 1]^T$ to find horizontal and vertical edges. These filter results are then multiplied element wise to produce the result E which represents the strength of edges in the original image. In Figure 4.12 the resulting E is visualized for an example image.

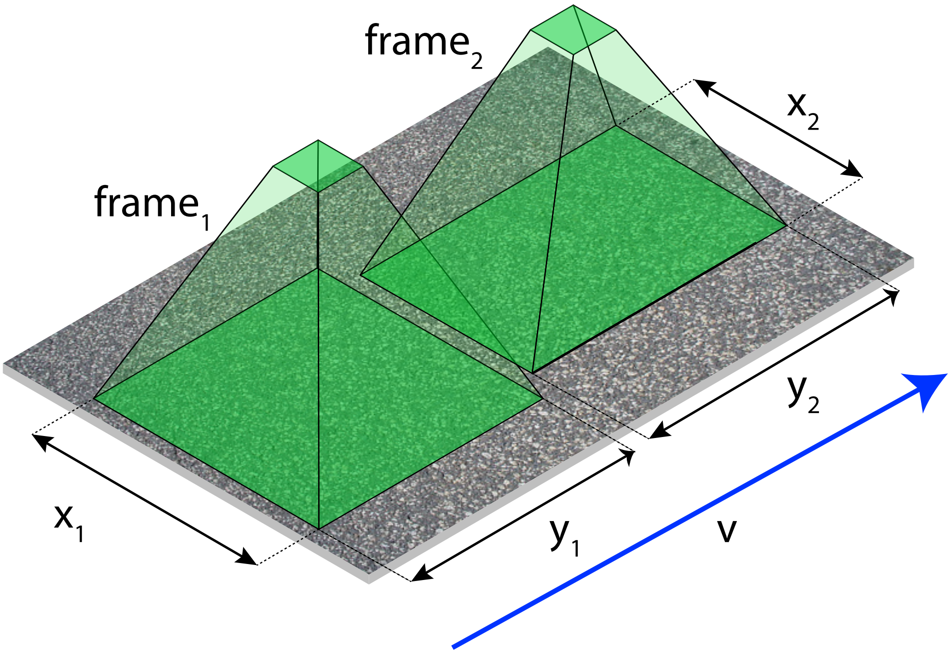


Figure 4.10: Dynamic adaption of frame size: initially, the speed is unknown, so ROI$_1$ is a square. Later, the speed is already known and ROI$_2$ has adapted its dimensions to match the movement speed and direction

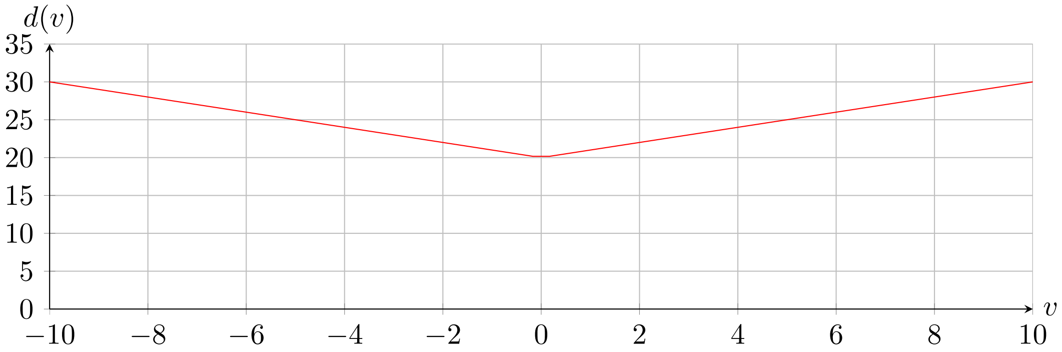


Figure 4.11: Size-Adaption-Function for Adaptive Grid Frame Algorithm: $d = d_0 + v_d$

The bright pixels / spikes in the resulting image E represent strong edges. Areas with very strong edges or with a high edge density are prefered positions for ROIs.

Since Tile Algorithms work sequentially and need a current, as well as a previous image, it is not very practical to continuously move ROIs around. The position adaption should rather be triggered for a ROI individually, when its confidence decreases and better positions for that ROI can be found.

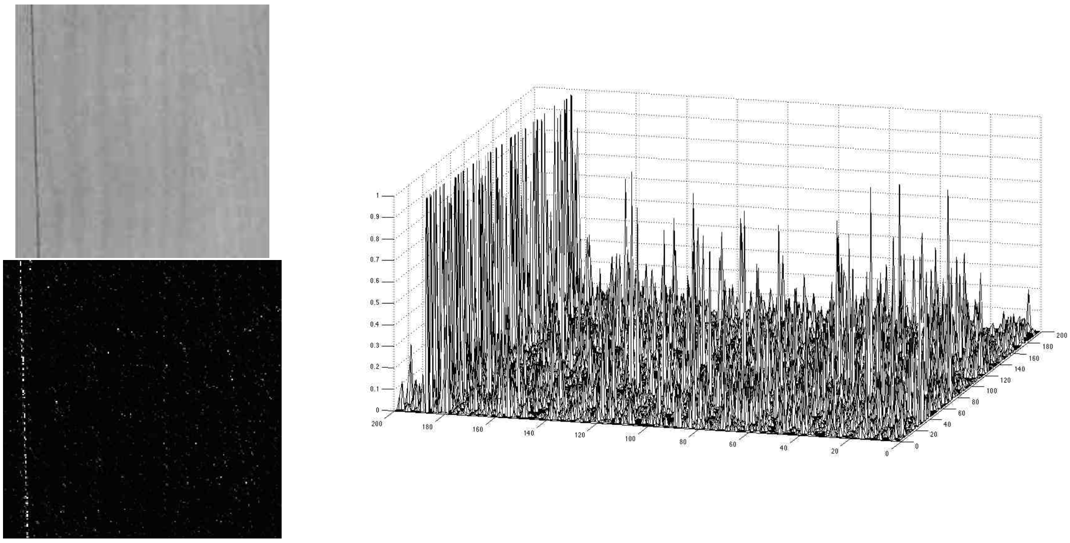


Figure 4.12: Simple Edge Detection: The original image on the top left is filtered and produces the grayscale (almost black/white) image at the bottom left. The brighter the pixel, the stronger the edge. The right image visualizes the bottom left image in a mesh-plot.

4.5 Remarks

4.5.1 Known Weakness / Difficult Situations

Naturally, there are systematic weaknesses that cause hard or even insolvable problems to system like the proposed one.

The most obvious problem with using a camera is, that it is sensitive to light conditions. When it is getting dark, the output image starts to get noisy and the number of erroneous pixels increases rapidly. Below a certain brightness threshold there is not enough light to produce an usable image anymore. An obvious solution is to illuminate the ground below the camera, but that increases cost and energy consumption and may not be desirable anyway. However, it is not mandatory to record images within the visible spectrum of light. In some situations it might be more feasible to capture infrared light for example. But that still leaves the problem of additional hardware cost and energy consumption. Another manifestation of the same problem is the occlusion of the view. For example, by stirred sand and dust when driving over a gravel road or on a sandy road. Furthermore, the sensor could become dirty from mud or dust and needs to be kept clean.

Another kind of problem is caused by environmental influences where the ground, or at least the part perceived by the camera, is not stable. One could imagine driving on a street in fall when there is dry leaves and foliage on the ground that go flying when the car drives along. Their movement could confuse the sensor because it is random and not correlated with the movement of the car at all. Similarly, when it is raining and there is already water puddles or even flowing water on the street. The water perceived from the camera is not solid. Its flow and also the reflections on the water could also confuse the camera.

In General, it is not strictly necessary to use a video camera as input. Any other method that produces a texture with sufficient resolution and update rate as output could be used instead and some other imaging method could be immune to some or all of the aforementioned problems.

4.5.2 Limits / Restrictions

There are two fundamental restrictions that limit the **maximal** and **minimal** transformation distance between two frames that still can be detected. The former is very obvious: the translation has to be smaller than the size of the image because there needs to be at least a small overlap between two consecutive frames.

$$d_{x_{\text{max}}} = W \cdot s$$
$$d_{y_{\text{max}}} = H \cdot s$$

where W and H are the *width* and *height* of the image frame in pixels, and s is the scale factor introduced in subsection 4.2.1, that relates pixel units to physical distance units. From these distances, the maximal measurable speed can be calculated by taking the frame rate of the camera into account:

$$v_{x_{\text{max}}} = d_{x_{\text{max}}} \cdot f_{\text{camera}}$$
$$v_{y_{\text{max}}} = d_{y_{\text{max}}} \cdot f_{\text{camera}}$$

To show the general order of magnitude, take for example a cam that takes 30 FPS and has a FOV of 0.5 meters which is typical for a camera mounted at a height of ~ 0.5 meters. For sake of simplicity it is assumed that the camera has a square FOV (which implies $v_{x_{\text{max}}} = v_{y_{\text{max}}} = v_{\text{max}}$). The maximal measureable speed with this setup then is

$$v_{\text{max}} = 0.5 \cdot 30 = 15m/s = 54km/h$$

However, keep in mind that in order to be able to calculate results with sufficiently high confidence, the overlapping area should be sufficiently large. This fact is modelled by a confidence estimation based on the meassured speed, see section 4.3.

For an autonomous rover, $54km/h$ is certainly fast enough. For application in a car however, the limit can easily be increased by using high speed cameras. If a camera with 120 FPS is used instead, the limit is already increased to $216km/h$ for example.

The second limit is a little bit more subtle and not absolutely defined. For most methods the accuracy is bound to whole pixels, meaning that the minimal measurable physical speed corresponds to the dimensions of the area visible in a single pixel.

This value is directly given by the value of the scaling factor s from subsection 4.2.1.

However, there are some methods that achieve even sub-pixel accuracy. One of them is the extension to the **FMT** method thats mentioned in subsection 3.6.3.

CHAPTER 5

Implementation

5.1 Overview

In this chapter, the conceptual framework from chapter 4 will be transferred from theory to a practical implementation. The first paragraphs of this chapter will describe the environment that was used for development. Following that, the software architecture of the implemented system is described and it is highlighted how the implementation works within the ROS framework. Then, the actual processing procedures are explained along with the details of their general implementation as well as specific methods that have been implemented.

5.2 Environment

5.2.1 Robot Operating System

As explained in section 3.4, ROS is framework- and toolset for the implementation of complex robot systems. ROS is open source, widely used in research and portable to a lot of (robotic-) platforms. Therefore, the choice to implement this work's practical part as a ROS module ensures that the implementation is usable in practical applications and also for other researchers with no extra work in an environment they are already familiar with.

5.2.2 OpenCV

A major part of the implementation consists of the implementation and the use of Computer Vision (section 2.4) methods. OpenCV has been chosen as the computer vision framework to be used, because it is Open Source and cross-platform with support for many operating systems and runtime frameworks. Most importantly, OpenCV is also available for use within the ROS framework. Also, it is very widely used and currently maintained by a Non-Profit organization.

5.2.3 Xcode

In order to be able to develop without the need of having actual access to the robot, the project was started in Xcode[1] with the ROS module being isolated into a submodule of the project. Only a small wrapper is necessary to run the sensor either as standalone OSX application or as a ROS module. This approach has been chosen because the author is familiar with the toolset for debugging, analyzing and profiling provided by Xcode.

5.3 ROS Integration

For an application to be usable within the ROS framework, it needs to be implemented as a ROS module. The running instance of an application within ROS is called a node. The ROS runtime allows the execution of multiple independent nodes at the same time and provides a way for nodes to communicate with each other via a **Publish/Subscribe**-architecture.

For a program to be usable as a ROS module, it is only necessary to implement a few specific lines of code that register the application as a node with a chosen name to the ROS runtime and some additional lines to register as publisher and/or subscriber for communication.

ROS modules are distributed as packages. A package is just a fixed scheme for organizing source files and some configuration. This configuration enables the ROS build tools to automatically manage dependencies and compilation of the module.

Executing programs in the ROS runtime are refered to as ROS nodes. When the executable programm in a module is running, it is also refered to as node.

5.3.1 ROS Modules

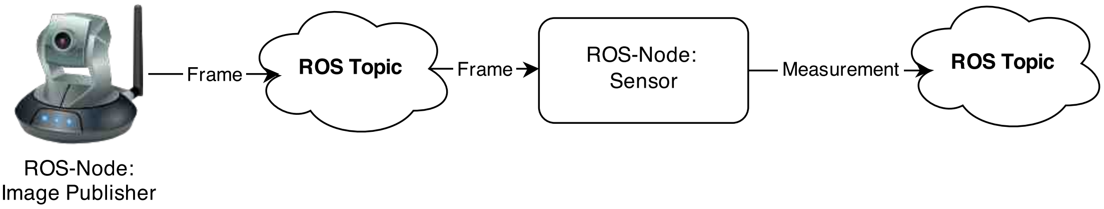


Figure 5.1: ROS-Setup with an external image publisher and the sensor as separate ROS-module.

Figure 5.1 illustrates the resulting ROS setup with an image publisher and the sensor as separated ROS modules. The modules communicate over a ROS topic, the **Image Publisher** node publishes a video stream to which the sensor node subscribes. The sensor node then publishes its result to another ROS topic to make these results available to other nodes.

---

[1] Xcode is an IDE for developing software in **Objective C**, **C++** and **Swift**, developed by Apple for OS X

```
int16 delta_x_px
int16 delta_y_px
float32 phi
float32 confidence
int64 number_of_frames
float32 frameProcessingTime
float32 meanProcessingTime
```

Figure 5.2: Measurement Message Definition

	Topic	Message Type
Input	`camera/image_raw`	`sensor_msgs/Image`
Output	`sensor/movement_params`	`measurement`

Table 5.1: Interface Summary for Sensor ROS-Module

5.3.2 Image Publisher

The sensor works on an image stream as input. The stream has to be published to the ROS topic **camera/image_raw** with the message type **sensor_msgs/Image**.

It was discussed to include the image acquisition directly into the sensor module which would have reduced the communication overhead of sending the image stream over the ROS system via message publishing. However, this approach would have limited the number of useable cameras to only those that are directly implemented in the sensor.

By using a ROS-topic as image input to the sensor, the image acquisition process is completely decoupled from the sensor. The big advantage of this approach is, that the used camera can be easily replaced without the necessity to change the sensor. With respect to portability, this is an important requirement that ultimately outweighs the communication overhead.

Currently, the gscam [5] [6] ROS-module is used as webcam driver which supports a wide range of webcams. A ROS launch file is provided to start gscam for the used webcam.

5.3.3 Sensor

The sensor module gets sequential images as input in form of a ROS message and calculates *measurements* as output. A *measurement* is a custom ROS message that encapsulates the results like speed in x- and y direction, angular velocity, confidence, and some additional performance indicators. Its definition can be seen in Figure 5.2. Table 5.1 summarizes the in- and output of the sensor with their exact topics and message types.

The calculation flow for a single frame is depicted in Figure 5.3. It has been divided into two basic tasks that consist of a few steps each. The first one is the processing for whole frames. This task is carried out by the so called Frame Algorithm. The Frame Algorithm however does not do the actual image processing, it rather coordinates the image processing in a number so called tiles. This is necessary, because as mentioned in subsection 3.5.3, calculation of rotation from one single image is not possible in a single step. The solution that was presented in section 4.1

is to calculate a number of correspondendes at multiple smaller patches of the image that only account for translation and then to calculate the global rotation from these correspondences.

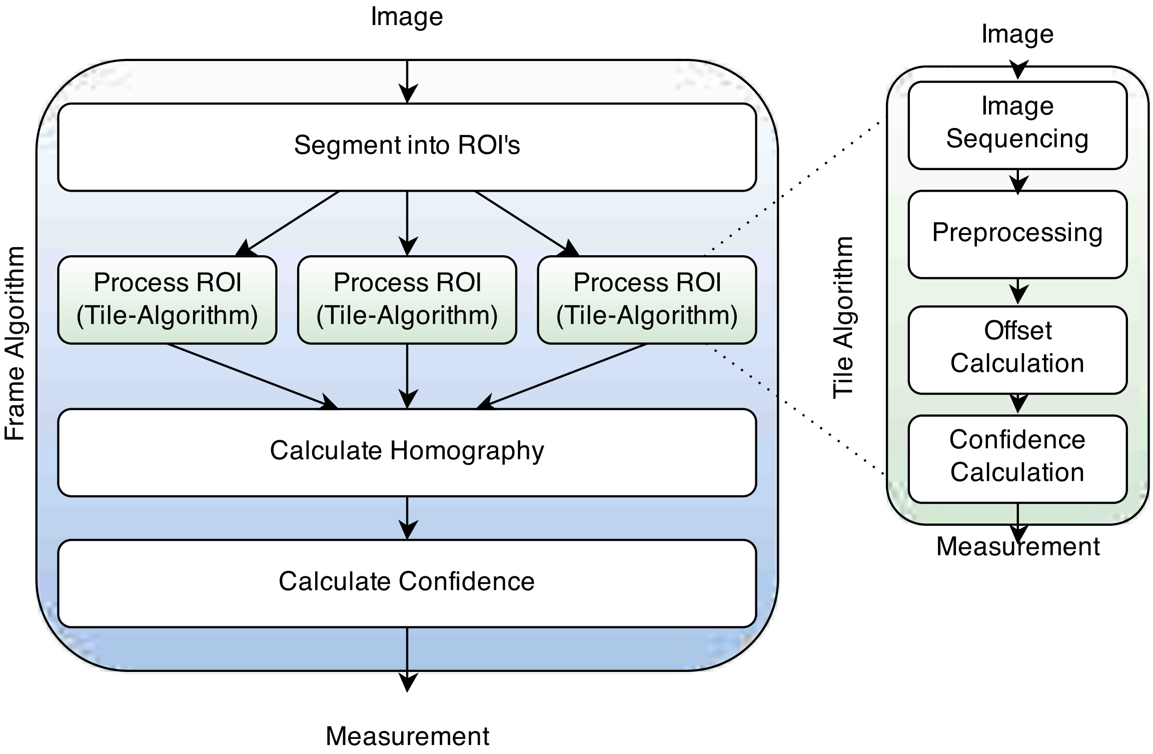


Figure 5.3: Overview of the sensor architecture. This illustrates the subtasks that are necessary to calculate a measurement and the calculation flow through the sensor.

The Frame Algorithm distributes several ROIs in the whole frame, where a ROI is just a rectangular image section. These image sections are then extracted into separated **tiles**. The difference between a ROI and a tile is, that the ROI is a reference to a region within a bigger image, in this case the whole frame, wheras the tile is that image region extracted into smaller image that contains only that region. Every tile is processed by a Tile Algorithm. There are various specific methods and algorithms that can be used for this task, some of them have been presented in section 3.6. Regardless of the specific algorithm, the processing steps for each tile are always the same and can be seen in Figure 5.3. The Tile Algorithm is agnostic about the fact that this is usually just a part of a bigger frame. Since the difference between the previous and the current tile is calculated, the sequence of images needs to be handled. Then, the tile frame is preprocessed, the offset between the previous and the current tile is calculated as well as a confidence estimation. Then, the Tile Algorithm outputs a single measurement that contains the offset distance $\vec{d} = \begin{pmatrix} dx \\ dy \end{pmatrix}$ and the confidence estimation. The Frame Algorithm collects the offsets of all tiles and constructs correspondences $(x_i, y_i) \leftrightarrow (x'_i, y'_i)$ from the position of the tile $\begin{pmatrix} x_i \\ y_i \end{pmatrix}$ and the offset $\vec{d_i}$ with the formula $\begin{pmatrix} x'_i \\ y'_i \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix}$ for all tiles i.

These correspondences are used to calculate a global homography that also includes rotation. Furthermore, a global confidence value is calculated that contains information about how accurate the calculated measurement is. There are two kind of confidences involved: **method dependent** values are based on specific knowledge of a certain Tile Algorithm method and the input data, so specific values have to be determined by the Tile Algorithm. On the other hand, as soon as there are several tiles in a frame, their specific confidences need to be combined into a global solution. This process has to be **generic**. Details about specific confidence calculations are mentioned in the description of the corresponding Tile Algorithm, the generic ones are explained in section 5.8.

5.4 Camera Calibration

In order to get accurate measurements, the camera that is used needs to be calibrated. This is necessary to rectify images from the camera and remove distortions that come from the camera's imperfection (see subsection 3.5.4 for more detail).

The **MonocularCalibration** [10] **ROS**-module has been used for this task. The calibration works by using a known image. In this case a checkerboard pattern with known dimensions. This image is then moved around in the cameras field of view, covering different positions, scales and orientations relative to the camera. The calibration node then calculates and displays the calibration matrix. The calibration matrix can also be exported into a calibration file which can directly be imported into the **gscam** image publisher node. This means, that the images published by **gscam** have already been corrected and the sensor does not have to deal with camera calibration.

5.5 Software Architecture

The sensor module has been implemented in C++ in an Object Oriented Programming approach.

The aforementioned parts of the calculation have been partitioned into a structure that very closely resembles the structure from Figure 5.3. These parts have been implemented in separated classes that interact with each other. To be precise, there are 4 different classes that are important.

The main entry point is the Frame Algorithm that basically encapsulates the whole computation. Its main responsibility is the segmentation of the image into tiles and the coordination of these tiles and the rest of the data flow between the other components in the course of the computation. It is implemented in the class `FrameAlgorithm` (From now on, the tyewriter font denotes the name of a class). The general behavior of handling tiles is implemented in this class. Specific behavior regarding tile placement and dynamic relocation of tiles is intended to be implemented in a subclass of `FrameAlgorithm`. Each frame is processed by an instance of the class `FrameAlgorithm` which implements the basic *housekeeping* of the calculation necessary for every specific algorithm. This *housekeeping* consists of **initialization**, **managing the frame sequence**, **managing size change**, **performance measurement** and **creating**

Calculation-Step	Class	Input	Output
Offset of a tile	`TileAlgorithm`	Frame(ROI)	Correspondence
Homography estimation	`HomographyEstimation`	Correspondences + last Result	Homography
Confidence estimation	`ConfidenceEstimation`	Correspondences + last Result	Confidence

Table 5.2: Computation steps, the classes that implement these steps and their inputs and outputs for Frame Algorithm composition.

debug output. The implementation of a specific image processing algorithm like one of those presented in section 3.6 is intended to done in a subclas of `TileAlgorithm`. The following homography estimation and the final confidence estimation are both implemented as subclasses of `HomographyEstimation` and `ConfidenceEstimation` respectively. For these two, the superclass merely provides an the interface that all specific subclasses have to implement.

Other than for an organized codebase, this class structure serves a second purpose. It is clear that with the variety of methods for Tile Algorithms, Homography Estimation and Confidence Estimation as well as various ways to layout and position tiles, there are ultimately countless possible configurations for the sensor by combining these different methods.

To be able to find configurations that work well, it was necessary to create a structure in which different methods can be exchanged, combined and tested with ease which was achieved by a composition structure.

The `FrameAlgorithm` has *slots* for tiles in the form of `TileAlgorithms` and for a `HomographyEstimation` and a `ConfidenceEstimation`. At runtime, these slots can be filled by instances of the corresponding classes or specific subclasses. The behavior of the sensor is therefore **composed** from several classes. By exchanging these instances at runtime, even drastic adaptions are possible dynamically.

Table 5.2 summarizes the classes that implement the separated computation steps and their (informal) interfaces in form of their inputs and outputs that are used for the composition. They all follow the same pattern and have a minimalistic interface, it is easy to implement new strategies in a new subclass.

5.6 Tile Algorithms

Interface

The main interface for a Tile Algorithm is very simple and contains the following methods:

- `processFrame(cv::Mat)` processes a new frame
- `getMeasurement()` returns the last measurement
- `prepareRoiUpdate()` prepares the tile for a change of the roi

All these methods are already implemented in the superclass and are responsible for all the *housekeeping* necessary for any algorithm. A subclass can specialize its behavior by implementing these methods:

- `processFrameInternal()` this is where the actual algorithm has to be implemented.
- `preprocessFrameInternal(cv::Mat &,cv::Mat &)` pre-processing of an image, usually consists of colorspace conversions or filtering.
- `type()` just returns the type of the algorithm

These methods will be called at the appropriate time during the frame processing by the superclass.

Additionally, there are two methods for debugging purpose, in most cases they can be handled entirely by the superclass and do not actually need to be implemented in a subclass:

- `setDebug(bool)` enables or disables the debug mode
- `getDebugFrame(cv::Mat &)` returns the current frame with additional debug information drawn into it (like e.g. the visualized offset)

Common Behavior

The common behavior which is implemented by the superclass, also called *housekeeping* earlier, consists of **initialization**, **managing the frame sequence**, **managing size change**, **performance measurement** and **creating debug output**.

```
input  : An image tile
output: Movement parameters
1  currentTileImage ←preProcessFrameInternal(input);
2  if ¬initialized then
3      previousTileImage ←currentTileImage;
4      initialized ←true;
5  end
6  startClock();
7  movementParameters ←processFrameInternal();
8  stopClock();
9  if debugMode then
10     prepareDebugFrame();
11 end
```

Algorithm 5.1: High-Level description of tile computation

The computation sequence for a tile can be seen in algorithm 5.1. In the first line, the input image is preprocessed. This step usually consists of some image filtering to remove noise or color conversions since most algorithms operate on grayscale images. Lines 2-4 deal with the initialization of the image sequence. If the current frame is the first frame, the same frame is simply used as previous frame as well since it is always necessary to have a previous frame initialized. This means that for the first frame, the calculated offset will obviously be 0. Lines 6 and 8 have to do with measuring the performance of the calculation, while line 7 is the actual processing. Finally, line 10 prepares a debug frame when the debug mode is activated.

Implemented Tile Algorithms

The following algorithms have been implemented in order to be able to compare these different methods with each other.

- **Histogram (subsection 3.6.1):** is a custom implementation of the theoretical algorithm
- **Correlation (subsection 3.6.2):** is a custom implementation that makes use of some *OpenCV* functions for the correlation.
- **Phase Correlation (subsection 3.6.3):** is a custom implementation of the theoretical algorithm.
- **Features (subsection 3.6.5):** makes use of the *OpenCV* function `goodFeaturesToTrack` which already implements ANMS.
- **Optical Flow (subsection 3.6.6):** uses the *OpenCV* optical flow algorithm.

5.7 Homography Estimation

After the first processing steps, a set of local correspondences has been calculated, be it from a set of tiles with an offset or from a set of features that have been matched. These correspondences have to be combined to find a global solution for the transformation from one frame to the other. For this specific application as motion sensor, the homography transform is constraint to a *isometry* transform, as long as the camera is correctly mounted pointing perpendicular to the floor which is assumed to be the case. In section 4.2, two methods have been introduced and pointers to additional methods have been provided.

Subclasses of `HomographyEstimation` need to implement only one method,

```
virtual HomographyParams_t
    estimateHomographyFromCorrespondences(vector<PointCorrespondence_t>
    &correspondences, measurement_t previousResult);
```

which takes a vector of correspondences from the current frame as well as the results from the previous frame and returns the parameters of the homography.

In the following paragraphs, the implemented algorithms are described.

Exact Match

An exact solution is unique and maps the source points exactly to their target points. To calculate an exact solution, the number of required correspondences is exactly defined by the type of transformation that should be calculated and is equal to the DOF of the transformation.

This `HomographyEstimation` is an implementation of the exact method that has been introduced in section 4.2.2. The exact method needs exactly 2 correspondences. If more correspondences are passed in, still only the first two are taken into account.

Least Squares DLT (Direct Linear Transform)

With real-world data which is impaired by noise and inaccuracies, an exact solution is often not appropriate. If one of the correspondences is subject to measurement errors, the homography is directly affected. Moreover, since only isometry is considered, certain cases of measurement errors can cause extreme errors in the calculated homography, since the exact method forces the solution to match the data. Such a situation could arise for example if the measurement suggests that the points have changed their position due to a measurement error which could be explained by a change in scale or perspective, but since these transformations are not considered, the results might be off by a large factor.

To remedy the situation, usually more samples than the minimum required number are taken and are averaged.

One typical method to find an optimal solution for a lot of data is a **least squares** approach, that is in short, the calculation of a solution where the sum of squared distances from the real datapoints to the datapoints that are projected by the solution is minimal.

This `HomographyEstimation` is an implementation of the DLT method that has been introduced in section 4.2.2.

Smoothness Motion Constraint

Even though non-exact estimations are more robust with regard to measurement errors, outliers still impair the accuracy of the resulting homography. To improve the accuracy, these *outliers* can be determined and removed from the data set before the calculation. In section 4.2.3, a method to remove outliers based on a smoothness motion constraint has been presented which has been implemented in this `HomographyEstimation`. The `Least Squares DLT`-Homography Estimation method is then used to calculate the homography from the remaining correspondences.

5.8 Confidence Estimation

The final processing step in algorithm 4.1 is the estimation of the confidence of the solution. A wide variety of parameters can be used for the estimation. More details about the theory of confidence estimation can be read in section 4.3.

It is important to understand that this Confidence Estimation is about the confidence for a whole frame. Additionally, every Tile Algorithm calculates a confidence based on the specific algorithm that it implements. This is only possible with deeper knowledge about the specific Tile Algorithm. Therefore, on Frame Algorithm level, an abstract estimation based on generally known parameters is needed. Obviously, the tile's confidences can still be factored into the global confidence.

A subclass of `ConfidenceEstimation` needs to implement only one method,

```
virtual float
    estimateConfidenceForFrame(IV::FrameAlgorithmMeasurement_t const
    &measurement, IV::measurement_t previousResult);
```

where `FrameAlgorithmMeasurement_t` is a special datatype that captures all the current correspondences and the confidences for each tile and measurement_t is the measurement from the previous frame that can be used to determine the confidence based on the change of measurements from one frame to the next.

The methods described in section 4.3 have been implemented in the aforementioned way.

5.9 Frame Algorithms

Interface

The main interface is very simple and consists of just two methods that are necessary for a caller to utilize a Frame Algorithm:

- `processFrame(cv::Mat)` processes a new frame
- `getMeasurement()` returns the last measurement

There are also additional methods for special setup and parameterization, e.g. for setting of margins, adding and updating ROIs in arbitrary positions as well as in a column/row pattern. If one of the implemented subclasses is used, these methods can be ignored by the caller. The most important two are the methods that add tiles to the frame:

- `addROI(cv::Rect, tileAlgorithm_t)` adds a ROI with a Tile Algorithm of the given type to the Frame
- `setRowsAndColums(int, int, tileAlgorithm_t)` spans Tile Algorithms of the given type in a grid with ranging over the whole frame (respecting the margins) with the given number of rows and tiles

Furthermore, there are two additional methods for debugging purposes

- `setDebug(bool)` enables or disables the debug mode
- `getDebugFrame(cv::Mat &)` returns the current frame with additional debug information drawn into it (like e.g. the visualized offset)

Common Behavior

The `FrameAlgorithm` superclass implements the bulk of behavior needed for frame processing. In Fact, it can be used as it is when the caller just adds a tile to it. All implemented subclasses are merely templates for different prepared configurations and are explained in detail later in this section.

The basic processing procedure depicted in Figure 5.3 and shown in algorithm 4.1 is implemented in the `FrameAlgorithm` superclass. By dynamically invoking the actual `FrameAlgorithm` instance at runtime and calling a post-processing hook, the basis for customization, and even dynamic reconfiguration, is laid out. Additionally, adaption to changes of the frame size and the necessary reinitialization for the new sizes are implemented, as well as some basic performance benchmarking like counting the number of processed frames and measuring the time that the algorithm takes for processing a single frame and calculating an average processing time. These processing times are used to reason about the computational performance and efficiency of certain methods.

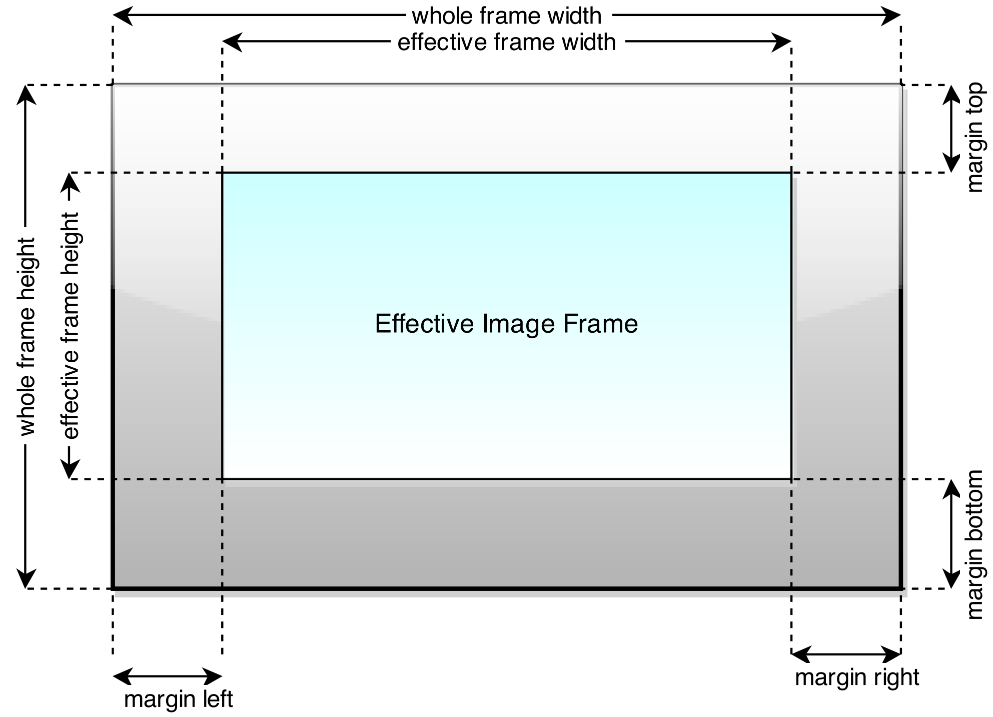


Figure 5.4: Illustration of how *margins* can be used to constraint the effective image frame.

Furthermore, most of the necessary configuration and parameterization methods are already implemented in the `FrameAlgorithm` superclass, like support for **margins** around the frame. Margins are necessary when it is not possible to mount the camera in an ideal way. For example, in tests, there often was an area in the bottom of the frame, where a shadow of the robot was visible and it was not possible to mount the camera further away to avoid the shadow. Obvi-

Algorithm	# ROIs	Homography Strategy	Confidence Strateg	Dynamic Adatpion
Feature Based	1	Least Squares SMC	Projection Difference	✗
Exact	2	Exact Match	Tiles Mean	✗
Least Squres	4	Least Squares SMC	Tiles Mean	✗
Grid	15	Least Squares SMC	Projection Difference	✗
Adaptive Grid	9	Least Squares SMC	Projection Difference	✓

Table 5.3: Overview of implemented Frame Algorithms

ously, this shadow would have had a negative effect on the calculation. So the bottom margin was adjusted to ignore the area of the image in which the shadow was visible. Figure 5.4 explains how the margins effect the resulting *effective image frame*. Even though these margins are implemented in the superclass, every subclass is responsible to respect these margins when placing tiles.

In order to prepare a Frame Algorithm , tiles need to be added to a frame. There are two method for this purpose. One for adding a tile with an arbitrary ROI, and one that pans out a grid of tiles over the whole frame. It is perfectly fine for a caller to add ROIs at will. Alternatively, one of the preconfigured subclasses can be used. They already initialize tiles and ROIs.

It was already mentioned, that the subclasses of Tile Algorithm can be thought of as configuration templates. They feature specific tile/ROI layouts, Homography Estimation and Confidence Estimation strategies. If one of these subclasses is used, no further configuration is necessary, and only `processFrame(cv::Mat)` and `getMeasurement()` need to be called. The following paragraphs describe the implemented configurations and their specific properties. Additionally, Table 5.3 summarizes the key facts about them.

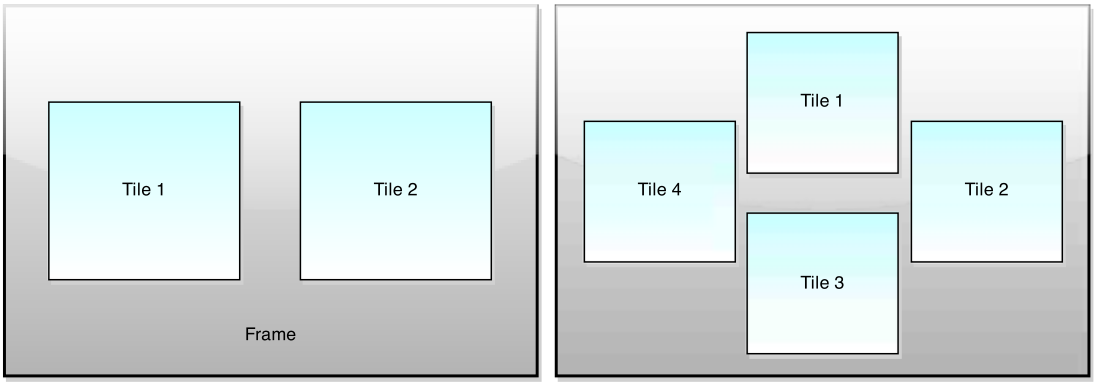


Figure 5.5: Tile Layouts for Tile Algorithms: 2 Exact (left) and 4 Least Squares (right)

Feature Based

Feature based Frame Algorithms are special because, they inherently produce a large number of correspondences, one for each detected feature. This means, in contrast to other methods it is not necessary to use several tiles to calculate a rotation.

This Frame Algorithm is designed to hold a single feature based Tile Algorithm with the tile ranging over the whole effective image frame (respecting the *margins*). So the tile layout looks exactly like in Figure 5.4.

The homography strategy is *Least Squares SMC* and the default confidence algorithm is *Projection Difference*.

Exact

This Frame Algorithm was implemented to test the *Exact Match* homography algorithm. Since an exact match needs exactly two correspondences, two tiles are arranged in the pattern that can be seen in Figure 5.5. The correspondence algorithm in use is *Tiles Mean* because both correspondences are exactly in the homography, so there are no outliers.

Least Squares

This algorithm arranges 4 tiles around the center of the frame without overlapping. The layout is visualized in Figure 5.5. The matching strategy of choice is *Least Squares SMC* and it is intended to learn about the performance of the least squares method with a low number of tiles. Due to its small number, the confidence is calculated by *Tiles Mean*.

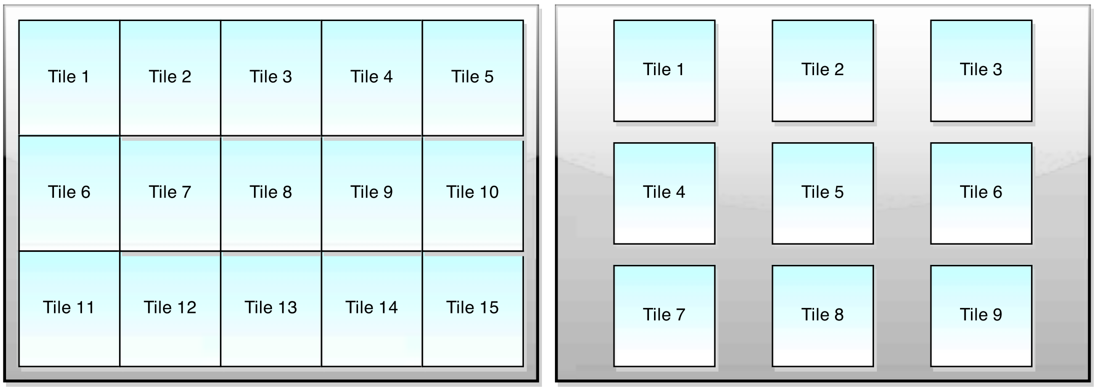


Figure 5.6: Tile Layouts for Tile Algorithms: Grid (left) and Adaptive Grid (right)

Grid

It is intuitive to just divide the whole frame into equally sized ROIs and use all of them. This algorithm implements this strategy with tiles of a fixed size as can be seen in Figure 5.6. The default parameters are 5 columns and 3 rows, but these numbers could be changed. The default homography strategy is *Least Squares SMC* and because the number of tiles is relatively high, the *Projection Difference* confidence algorithm is used.

Adaptive grid

This is an adaptiv version of the gird strategy. The layout is similar grid, but the tiles are not placed to take up the whole frame, but rather are placed at fixed positions that are defined by the number of columns and rows, which are 3x3 by default.

The dynamic adaption then changes the size of the tiles in reaction to the speed that was calculated according to the **Adaptive Frame Size** strategy from section 4.4. The tile will be stretched in the direction of the movement vector and compressed in the direction normal to the movement vector. This causes the area of the tile, and therefore the also calculation time, to stay almost the same, while the stretching enables the measurement of higher speeds since the maximal measurable speed is determined by the size of the tile.

CHAPTER 6

Results

6.1 Experimental Setup

The sensor has been tested in an experimental setup with a vehicle driving in different environments.

The vehicle used in the experiments was a **Pioneer 3-AT**[1] research rover with a **Playstation 3 EYE**[2] USB camera mounted on the rover to observe the ground under the rover.

The rover was programmed to move on a certain path including several speed levels in forward and backward direction, left- and right turns while also driving forward and pure rotations while not moving forward or backward.

This sequence of movement was then performed by the rover in different environments with different floor patterns. The video from the camera has been recorded for every run. These videos have been used as input for the sensor in different configurations in order to determine the performance of varying sensor configurations in different scenarios.

Pioneer 3-AT Research Rover

“Pioneer 3-AT is a small four-wheel, four-motor skid-steer robot ideal for all-terrain operation or laboratory experimentation.” ([9]). Figure 6.1 shows an image of this rover. It has **ROS (GROOVY Distribution)** installed and can be controlled with the **RosARIA** [3] module by sending *cmd_vel* commands with *linear*- and *angular* velocities.

To perform the scripted motion path, a separate ROS-module has been implemented that sends a defined sequence of *cmd_vel* commands. Because of varying message delays (ROS does not garuantee real time deadlines) and a possible influence of the surface, the path is not expected to be exactly the same for each run. In practice, however, the resulting paths had very little deviation.

---

[1] http://www.mobilerobots.com/ResearchRobots/P3AT.aspx

[2] http://us.playstation.com/ps3/accessories/playstation-eye-camera-ps3.html

[3] http://wiki.ros.org/ROSARIA

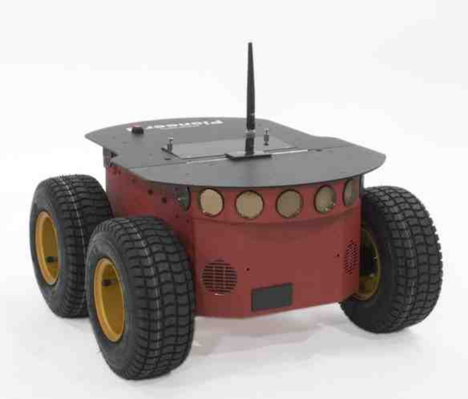


Figure 6.1: Pioneer 3-AT Research Rover

Playstation 3 EYE Camera

A Playstation 3 EYE Camera has been used as the camera of choice because it can achieve frame rates of up to 120 FPS. Yet, because it has been produced in huge quantities for the Playstation 3, its price is even below the price of other USB webcams.

However, for the same reason, there are no official drivers available. For live usage in ROS, the *gscam* module already works with the camera as described in subsection 5.3.2. To be able to record videos from the webcam, it was necessary to develop a separate capture program specifically for the PS3 Eye. This specifically created capture program saves the video frame by frame as single images as they are received from the camera to avoid errors introduced by lossy video compression. Additionally, it offers control over basic hardware settings of the camera, like **gain** and **shutter speed**, which was necessary to adjust for different lighting conditions, as described in the next section.

Calibration Values

The camera was mounted on the rover at a height of 25cm, with an offset of -43cm from the robot's center (mounted at the backside of the robot), with no lateral offset. The camera was recording with 30FPS in standard VGA resolution (640x480). The field of view was measured to be 21,2cm in the width dimension. These parameters result in the following calibration values:

- **camera_offset:** -1298px
- **pixel_size:** 0,00033125m
- **fps:** $30s^{-1}$
- **maximal speed:** $0{,}8m/s$

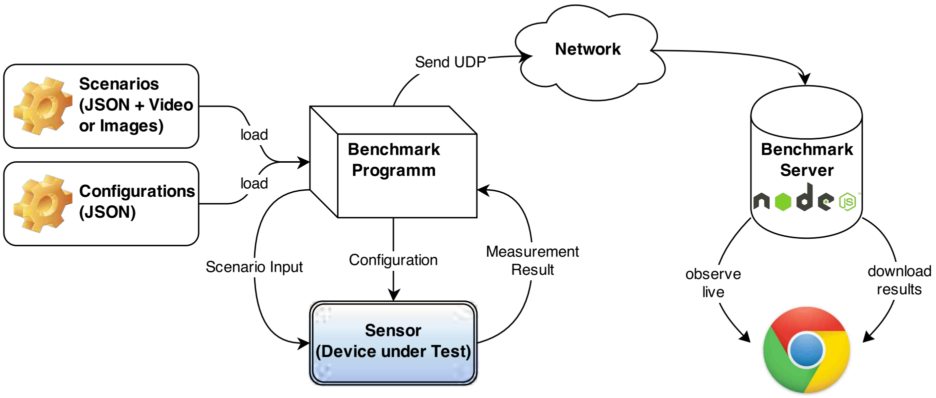


Figure 6.2: Experimental Setup

Value Recording

In total, 17 different sensor configurations have been tested in 11 different scenarios. In order to do so, a benchmark program has been written that loads a list of sensor configurations and a list of scenarios as JSON files. With this approach, it is easy to add new configurations and scenarios in the future.

The experimental setup is depicted in Figure 6.2. For each scenario, the input is read either as compressed video file or as sequence of single images. For every configuration, a new instance of the sensor is allocated and configured accordingly. The scenario's input images are sent to the sensor and the sensors output is then sent over UDP to the benchmark server.

The benchmark server is implemented in Javascript with the *node.js*[4] Framework. It hosts a UDP server to receive the measurements from the sensor as well as a local HTTP server, that visualizes the received values in a live plot. Furthermore, the HTTP server allows the download of the results in the form of .csv files at the end of a benchmark run.

6.1.1 Reference Movement Sequence

The movement sequence is illustrated in Figure 6.3 as the linear and angular velocity for the vehicle. The solid line represents the commands that are issued to the vehicle. The dotted line represents a simulation of the actual movement of the vehicle taking inertia into account.

Another way to visualize the movement is to plot the accumulated linear and angular velocity as depicted in Figure 6.4. These values represent the position and orientation from the starting point at each time instance. This visualization gives a better sense of movement and absolute

---

[4] http://nodejs.org/

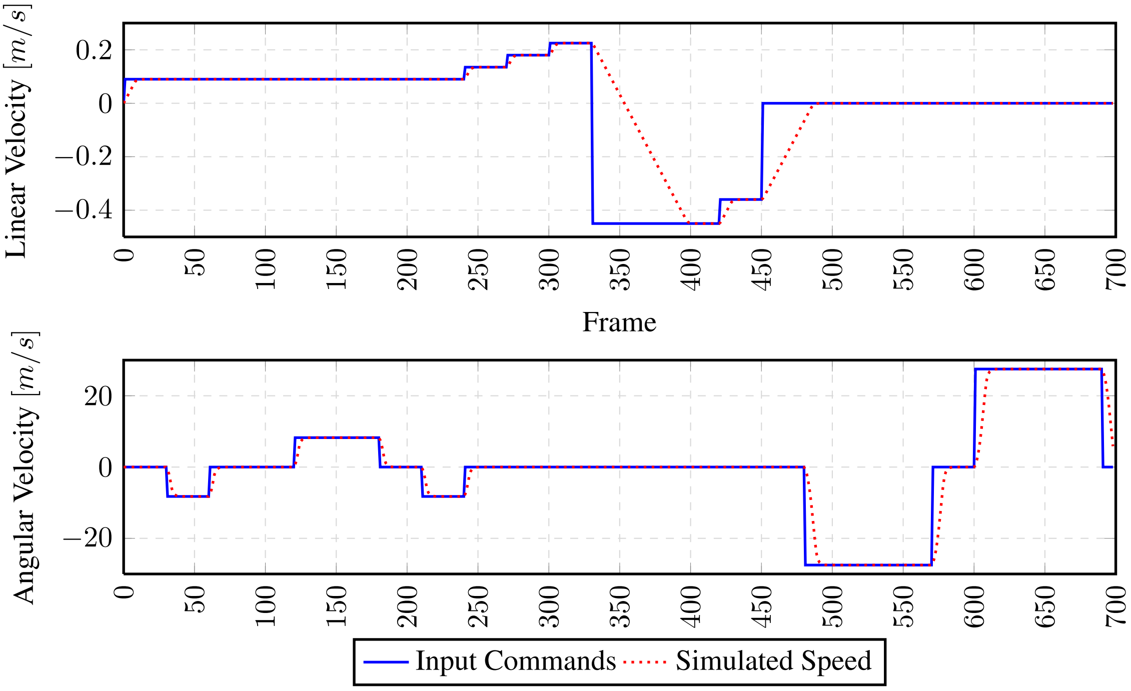


Figure 6.3: Illustration of vehicle movement parameters

deviations can be spotted easier. Again, the solid line is the ideal movement based on the input commands, while the dotted line is the simulated movement taking the vehicles inertia into account.

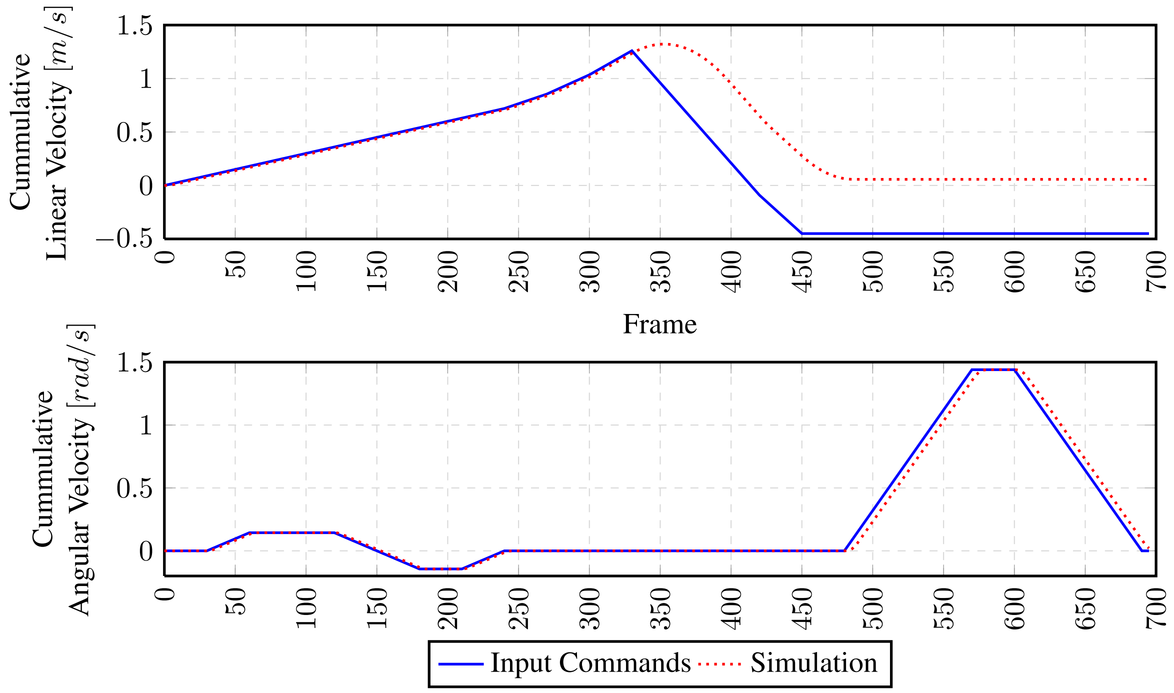


Figure 6.4: Illustration of cummulative velocities

6.2 Scenarios

In order to assess the performance of the sensor in different situations, various places with varying floor patterns in and around the institute building have been examined. These different set-ups will be called *scenarios*.

For every scenario, the accumulated linear velocity for every Tile Algorithm is plotted to give a first impression on over how good the general performance of the different algorithms is. This plot is mainly meant to privode an outline. Some algorithms can already be discarded at this point. The others will then be analyzed in more detail.

Street

The images in Figures 6.5, 6.6, 6.7 and 6.8 depict different street scenarios. The corresponding videos have been recorded on a day with fair weather, shortly after lunch when the sun was very bright. The settings of the camera had to be adjusted to the minimal possible gain and fastest shutter speed. Still, the images are very bright.

The first one, **street1** (Figure 6.5), is an example of a tarred street. The picture is dark, but with high contrast and a rich structured texture. The surface is evenly flat.

The second (**street2**, Figure 6.6) and third (**street3**, Figure 6.7) scenarios are both examples of a common street. The surface is flat and even, and similarly to street1, the textures have a rich structure and good contrast.

The fourth scenario is from a **cobblestone** path (Figure 6.8). The surface is pretty rough and bumpy. Because of this, the robot suffered from sever vibrations while driving on this surface. Furthermore, the color of the surface was very bright and reflected a lot of light. Therefore the image is very bright as well, even with the camera settings adjusted as much as possible.

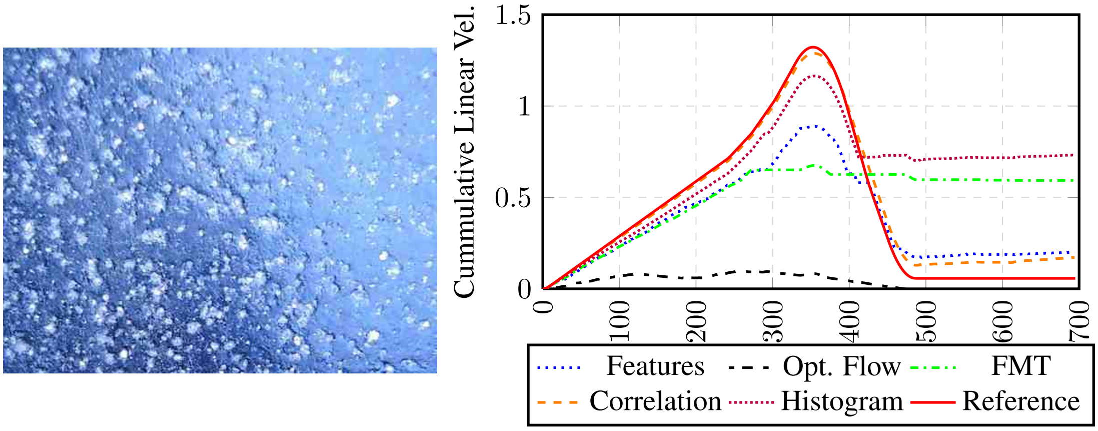


Figure 6.5: Floor-Patterns Street 1

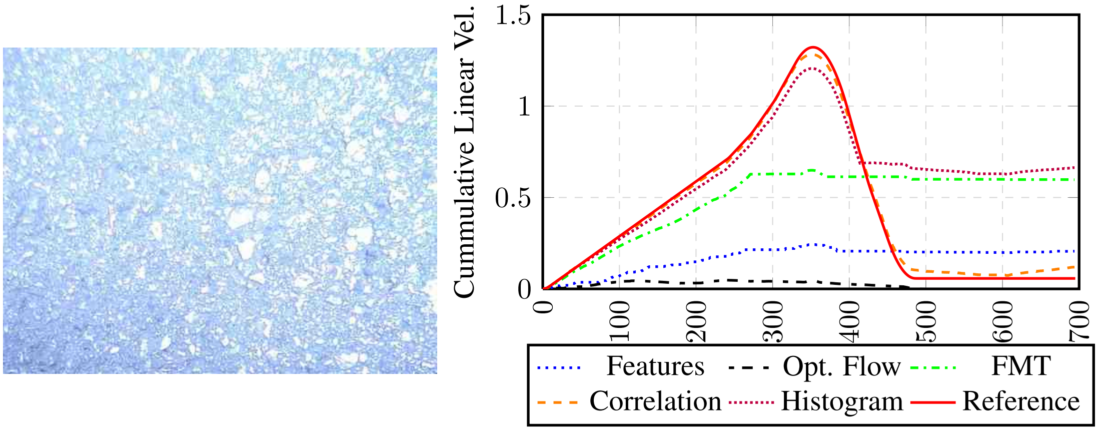


Figure 6.6: Floor-Patterns Street 2

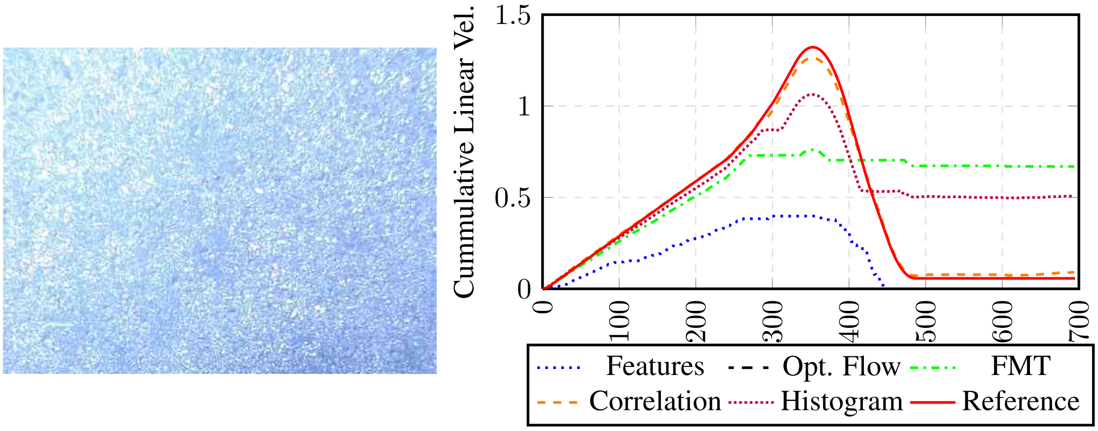


Figure 6.7: Floor-Patterns Street 3

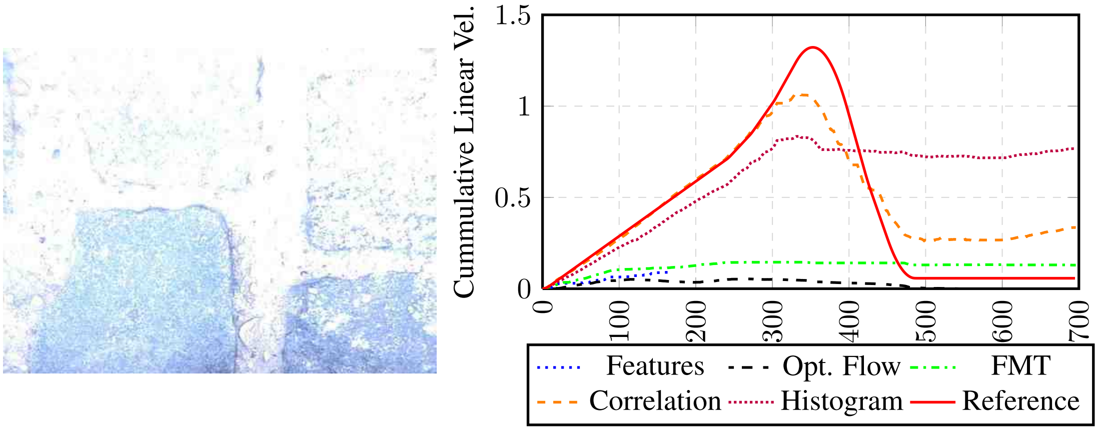


Figure 6.8: Floor-Patterns Cobblestone

Indoor laminated

These scenarios (see Figures 6.9, 6.10, 6.11 and 6.12) have been recorded indoors in the institute building. The surface in all these cases is a laminated floor that is very evenly flat. The first two scenarios were recorded in a lab room with different lighting conditions. In the dark one, the window blinds of the room have been lowered to dim the incoming sunlight. The room was located at the shadow side of the building. In these scenarios, both, shutter speed and gain, are in the default center position. The third one was recorded inside of the institute where there was a big window through which the sun was shining. Because of this, the image is very well lit. The last one was recorded in a different room where there is no window, and the room is only lit with artificial light. Here, the gain had to be increased by about 50%, which already produced visible noise in the images.

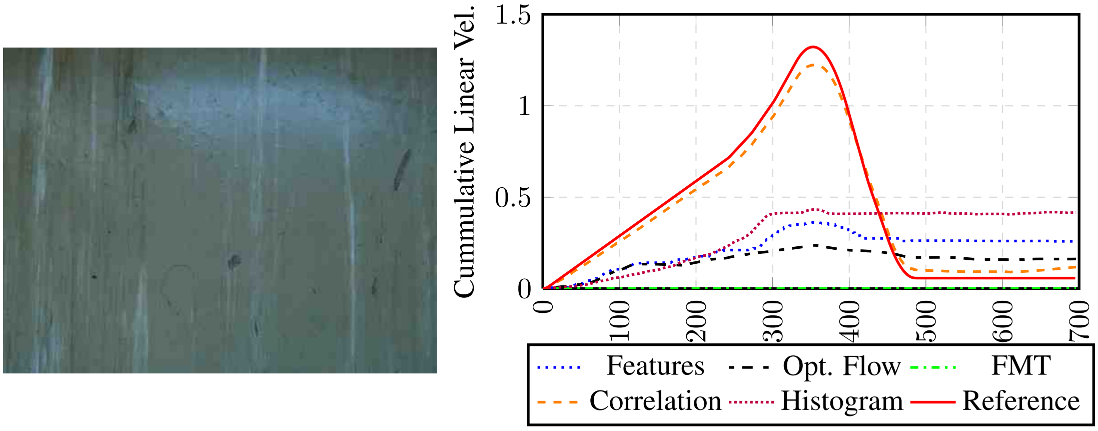


Figure 6.9: Floor-Patterns Street 1

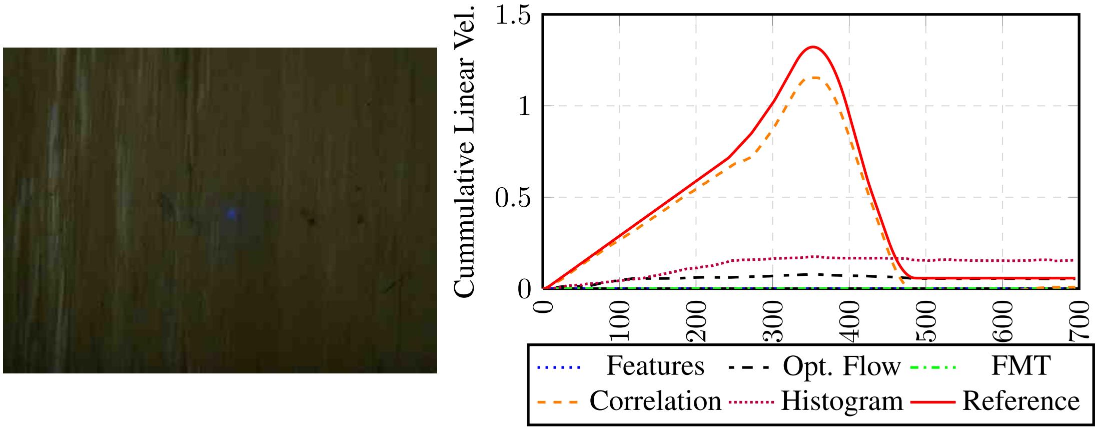


Figure 6.10: Floor-Patterns Lab Dark

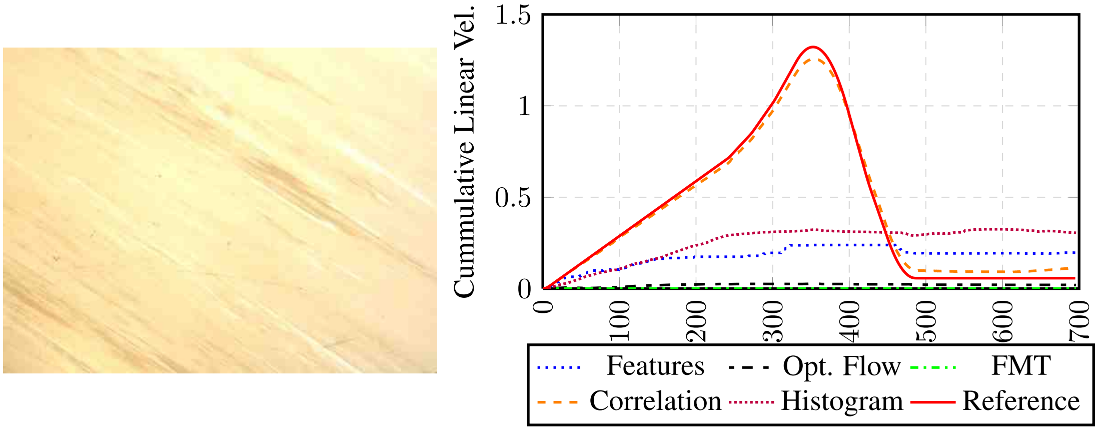


Figure 6.11: Floor-Patterns Institute

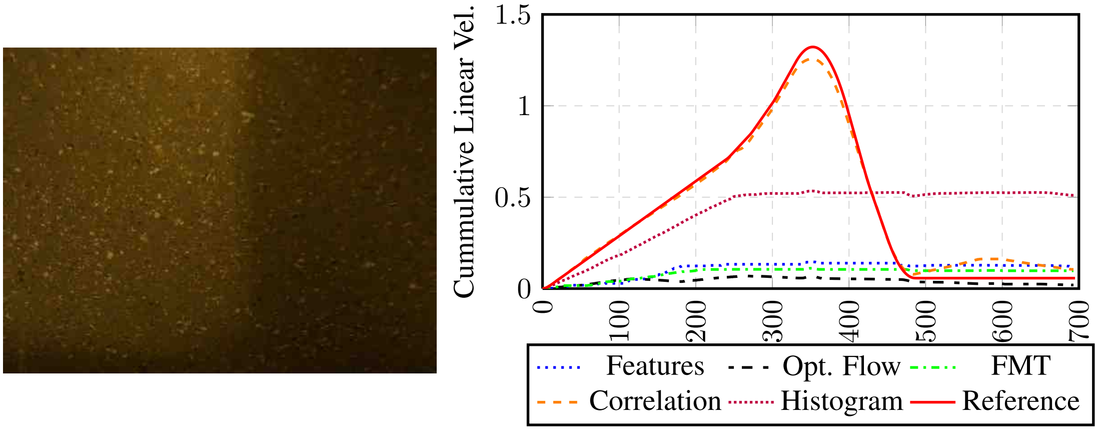


Figure 6.12: Floor-Patterns Kitchen

Indoor Tiles

The scenarios in Figure 6.13 and Figure 6.14 are from the staircase and the foyer of the institute building. The floor consists of tiles, the seams between the tiles have caused subtle vibrations in the robot when driving over them.

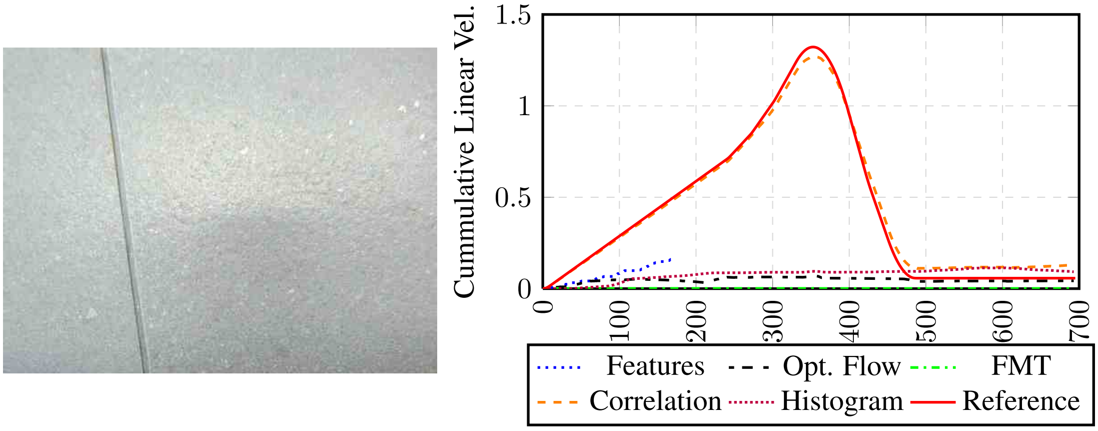


Figure 6.13: Floor-Patterns Foyer

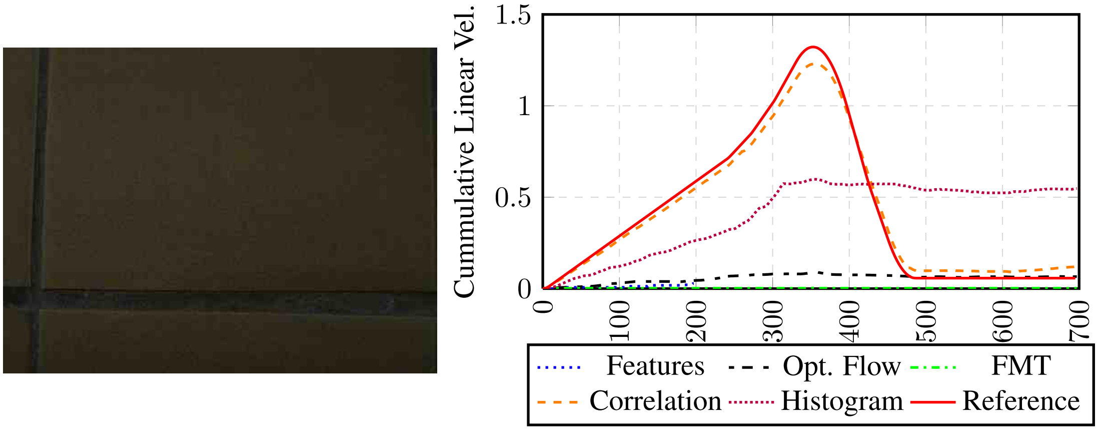


Figure 6.14: Floor-Patterns Staircase

Grass

The scenario in Figure 6.15 was recorded on a meadow. It is different from all the other scenarios in that it has more different colors than the floors seen before and irregular patterns. Another uniqueness is, that the height of the grass was up to 5cm, and therefore, the distance to the ground varies in the image. Furthermore, wind was blowing and some grass blades and leaves were moving because of the wind, causing irregular movement in the image.

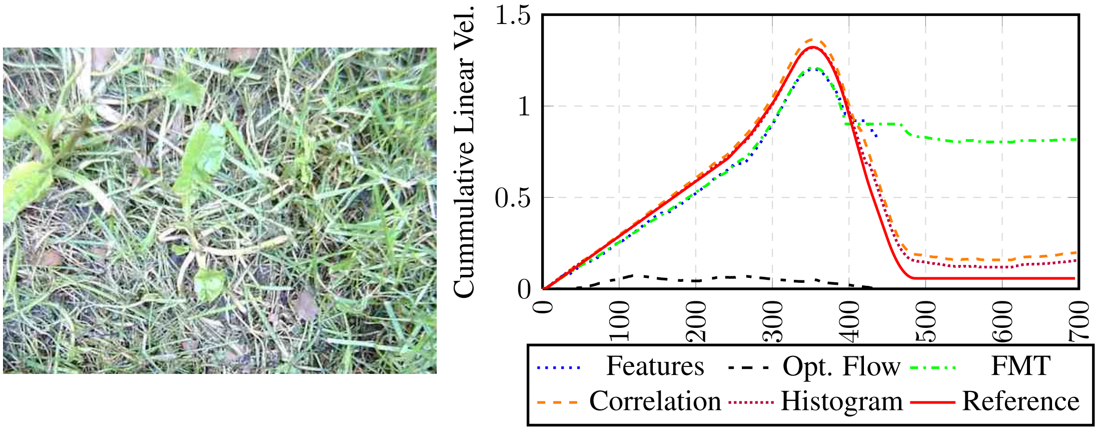


Figure 6.15: Floor-Patterns Grass

6.3 Results

6.3.1 General Results

The first obvious result when looking at the plots for different scenarios is, that only the *correlation* Tile Algorithm works consistently good in almost all scenarios. The results are always very close to the reference curve. The only exception is the *cobblestone* scenario (Figure 6.8)), where, as mentioned, the vehicle was vibrating very much, because of the uneven structure of the floor which caused a lot of erroneous movement in the camera frame.

The second finding is, that the *optical flow* method does not work well in any of these scenarios. The problem is illustrated in Figure 6.16: the whole image should be evenly colored in the same color. However, because of uniform structures, big parts of the image can not be matched, and the resulting parts are white, meaning that the movement at this point is calculated to be zero.

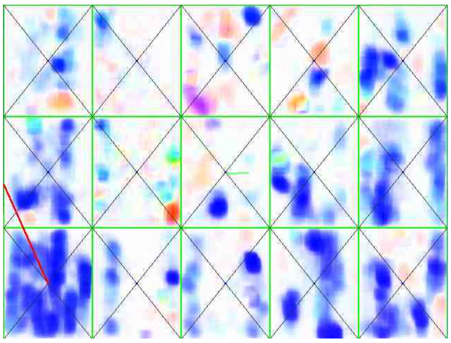


Figure 6.16: Debug-Visualization of Optical Flow Method. The color for each pixel represents the movement direction at that pixel, the saturation of the color represents the magnitude of the movement. White pixels correspond to zero movement.

For similar reasons, the *feature based* Tile Algorithm performs poorly, except in the *grass* scenario. Due to the texture, even though great numbers of features are detected in each frame, these features are very similar to each other and can not reliably be matched.

FMT and *histogram* methods perform reasonably well in some of the scenarios - seemingly in the *street* and the *grass* scenarios - and perform bad in some other scenarios. They only work up to a certain speed. The reason for this is illustrated in Figure 6.17 - at higher speeds, the image is smeared in the movement direction. This is because the shutter speed of the camera is too low and the vehicle moves too far in the timeframe of the exposure.

Since in the outdoor scenarios, the shutter speed was already as fast as possible, a faster camera would be necessary to fix this problem. In indoor scenarios, additional illumination would be necessary as well. Otherwise the image would get too dark with faster shutter speeds.

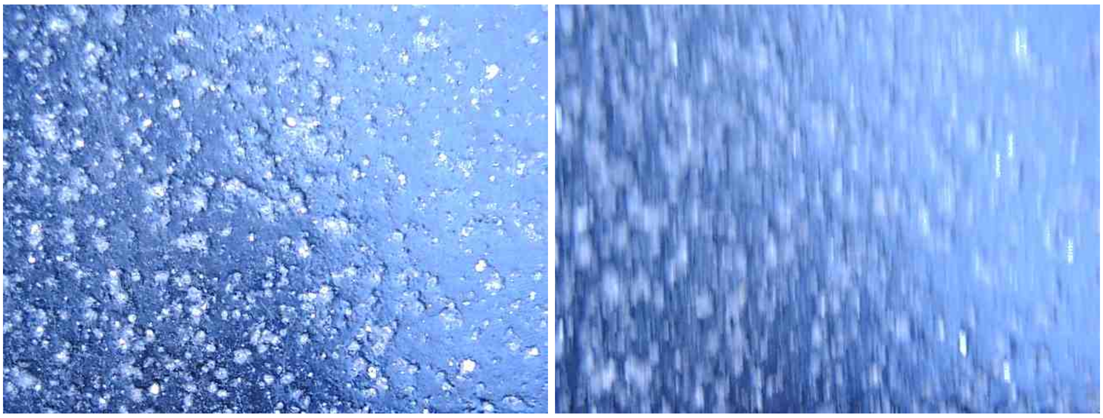


Figure 6.17: Illustration of image smearing at higher speed. The left image was recorded at low speeds and is sharp. The right image was recorded at higher speed and it is smeared in the movement direction.

Due to these findings, the following analysis will concentrate on the *correlation* methods with more detailled investigations.

6.3.2 Performance Results

Beside accuracy, the computational performance of the sensor is a major factor. These Performance-Metrics have been recorded while running the scenarios on an **2.6 GHz Intel Core i5 Macbook Pro with 8GB DDR3 RAM**. Obviously, the absolute performance values listed in Table 6.1 can vary depending on the machine, the sensor is running on. To compare the methods to each other, their relative performance results have also been determined. These relative results also stay the same on other hardware configurations.

It is not surprising, that the *Histogram* method with *Exact* Homography Estimation is the fastest configuration. First, this is because the *Histogram* method is computationally very simple as mentioned in subsection 3.6.1. Secondly, this is because the *Exact* Homography Estimation has the smallest area to compute.

The values in the relative time column in Table 6.1 are related to this configuration which acts as the reference configuration.

Configuration		Time/Frame $[ms]$	Avg. FPS $[s^{-1}]$	Relative Time Factor$[1]$
Feature Based		23.988	41.688	19.072
Opt. Flow	Grid	111.189	8.994	88.405
	A.Grid	51.643	19.364	41.061
	Least Squares	33.876	29.519	26.935
	Exact	14.837	67.399	11.797
FMT	Grid	30.951	32.309	24.609
	A.Grid	17.025	58.737	13.537
	Least Squares	8.470	118.065	6.734
	Exact	4.326	231.145	3.440
Correlation	Grid	12.057	82.941	9.586
	A.Grid	8.691	115.060	6.910
	Least Squares	3.243	308.360	2.578
	Exact	1.635	611.611	1.300
Histogram	Grid	7.838	127.581	6.232
	A.Grid	3.862	258.913	3.071
	Least Squares	2.606	383.730	2.072
	Exact	1.258	795.089	1.000

Table 6.1: Computational Performance Results. The Relative Time Factor is based on the reference configuration (Histogram + Exact).

The relative performances have been listed in Table 6.2 sorted by Tile Algorithm and Frame Algorithm. This table illustrates the order of the algorithms by computation speed from fastest to slowest:

$$\text{Histogram} < \text{Correlation} < \text{FMT} < \text{Optical Flow}$$

and

$$\text{Exact} < \text{Least Squares} < \text{Adaptive Grid} < \text{Grid}$$

The order of Tile Algorithms can easily be explained by the total area that has to be calculated. *Exact* only has two tiles of a fixed size, *Least Squares* has 4 tiles of the same size and therefore twice the area. As expected, the computation time is also almost exactly twice as long. On the other hand, *Grid* always calculates the whole area of the input image. While the worst case area of *Adaptive Grid* is also about the whole image, most of the time the tile size can reduced by the adaptive algorithm.

These orders also have been visualized in Figure 6.18 and Figure 6.19.

	Histogram	Correlation	FMT	Optical Flow
Exact	1	1,300	3,440	11,797
Least Squares	2,072	2,578	6,734	26,935
A.Grid	3,071	6,910	13,537	41,061
Grid	6,232	9,586	24,609	88,405

Table 6.2: Performance Comparison of different Configurations relative to the reference configuration (Historgram + Exact).

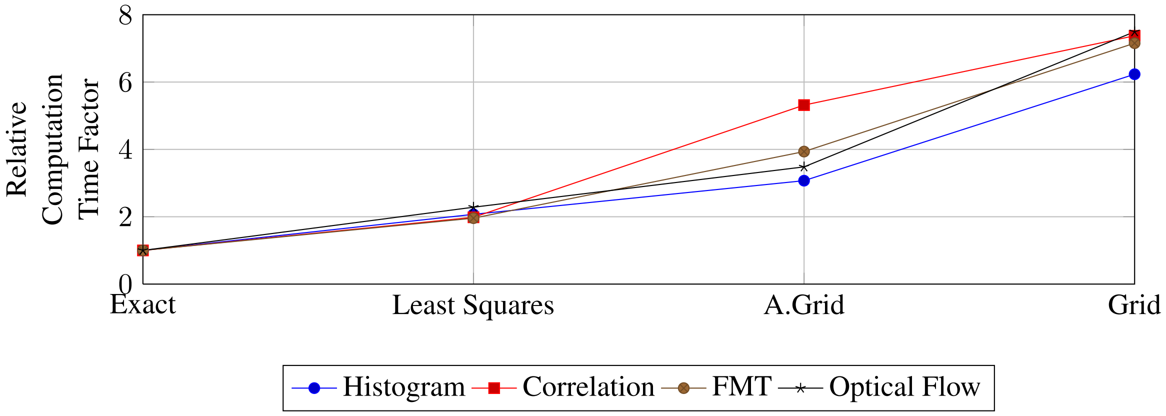


Figure 6.18: Comparison of computation time for various Frame Algorithms, relative to the reference configuration (Historgram).

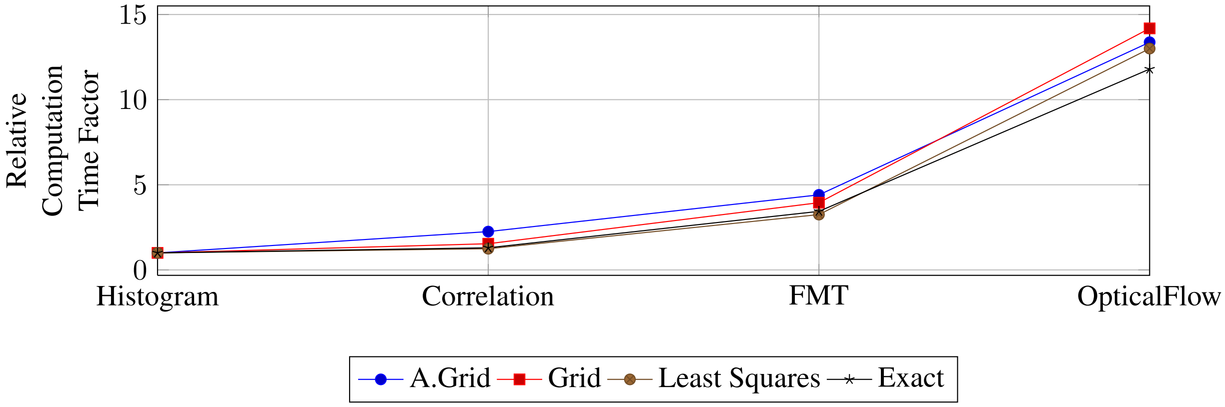


Figure 6.19: Comparison of computation time for various Tile Algorithms, relative to the reference configuration (Histogram).

6.3.3 Velocities and Confidence

The plots in Figure 6.20 display the results for *linear velocity*, *angular velocity* and *confidence* for each frame of the **street2** scenario. The configuration was the *Grid*-Frame Algorithm with *Correlation* tiles. It can be seen that the values fit the reference curve pretty good, but for both, linear and angular velocity, there is some jitter around the mean value. This jitter seems to be continuous like a pattern, alternating between over- and underestimating the actual value. It may be possible that this is caused by an image frame rate that is not perfectly constant. Possibly, this jitter could be eliminated by recording precise timestamps when a new frame is available, and using these timestamps to calculate the precise time between frames. Unfortunately, this correction was not possible with the recorded video.

Furthermore, it can be observed that the few outliers, especially in the angular velocity around frame 275 and in the linear velocity between frame 450 and 500, are detected by the confidence estimation and the confidence value decreases drastically as desired.

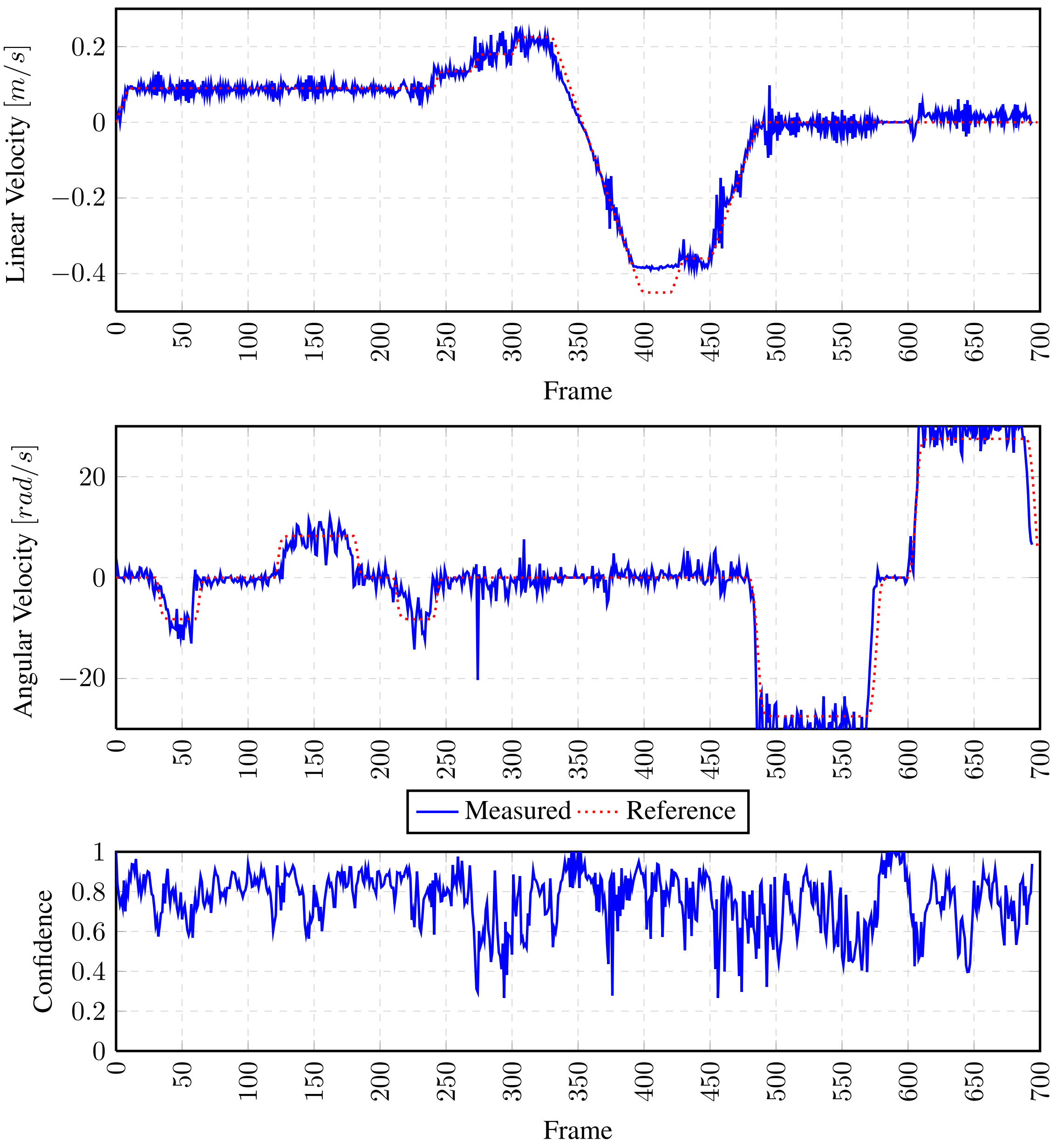


Figure 6.20: Linear- and Angular Velocity with Confidence for Scenario **street2** with Correlation in Grid Configuration

6.3.4 Least Squares vs. Exact Homography

In this experiment, the methods for homography estimation are compared to each other. As explained in subsection 4.2.2, there are two different methods: exact and approximation methods.

Because the former only requires the minimal number of correspondences, it was suspected that the exact estimation is more prune to outliers.

Figure 6.21 shows the results in scenario *street3* with the forward speed and confidence. The configuration with *exact match* homography has 2 *correlation* tiles, the configuration with *least squares* homography had 16 *correlation* tiles in *grid*-configuration.

Overall, the speed results are almost the same. However, at the times around frame 280 and around frame 415, outliers cause severe estimation errors in the exact homography. Still, these errors are detected by the confidence estimation, and as expected the confidence at these points is 0.

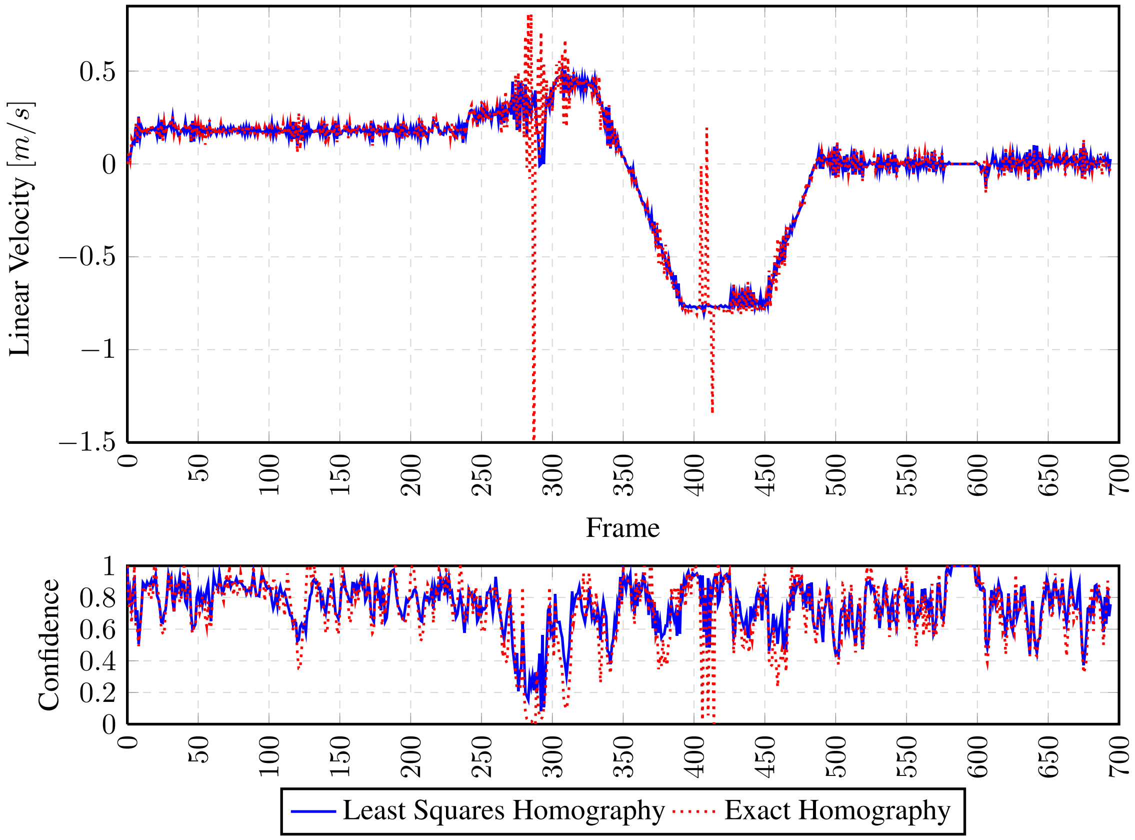


Figure 6.21: Comparison of Least Squares Homography Estimation vs. Exact Homography Estimation

6.3.5 Accuracy

In order to compare the accuracy of different configurations, different values have been calculated and compared to the reference from subsection 6.1.1. In Table 6.3, the final value of the cumulative *speed*, *angle* and *position* have been determined, and their deviations to the reference have been listed. The calculation for the cumulative velocities is very straight forward:

$$V_{\text{linear}} = \sum_{i=0}^{N} v_{i_{\text{linear}}} \cdot t$$

$$V_{\text{angular}} = \sum_{i=0}^{N} \varphi_i \cdot t$$

For the final position, the movement vectors for every frame have to be summed up respecting the current angular orientation, which is the cumulative rotation from the start to the current frame, taking the time between two frames t into account:

$$\Phi_i = \sum_{n=0}^{i} \varphi_i \cdot t$$

Again, the elapsed time between two frames t has to be taken into account for the translation

$$Pos_x = \sum_{i=0}^{N} v_{i_{\text{linear}}} \cdot \cos \Phi_i \cdot t$$

$$Pos_y = \sum_{i=0}^{N} v_{i_{\text{linear}}} \cdot \sin \Phi_i \cdot t$$

The relative values are in reference to the total traveled distance and the total rotation respectively:

$$V_{\text{total}_{\text{linear}}} = \sum_{i=0}^{N} |v_{i_{\text{linear}}}| \cdot t = 2,5866333274m$$

$$V_{\text{total}_{\text{angular}}} = \sum_{i=0}^{N} |\varphi_i| \cdot t = 197,3194444444^\circ$$

Another way to analyzie the accuracy is to calculate statistical parameters over the whole timeseries. The values in Table 6.4 show the deviations of the measured velocities from the reference velocities. Note that in addition to the *linear* and *angular velocities*, that are already familiar, the *lateral* velocity is listed as well. This is the velocity normal to the vehicles moving direction. In normal driving situations, this velocity is zero. An example of an abnormal driving situation where this velocity is not zero is if the car is drifting. As mentioned in subsection 4.2.1, the position of the sensor causes lateral velocity to be measured, which is then filtered out with the help of the sensors calibrated position and the measured linear and angular velocity. Therefore, these values are a good indicator of how accurate the linear and angular velocity are in relation to each other. For each velocity, the **mean** μ as well as the **standard deviation** σ^2 is included. Obviously, a value closer to zero is better for both measurements.

The velocity deviation values have also been visualized in Figure 6.23, Figure 6.24 and Figure 6.25. In these plots, excessively large values have been truncated such that the differences of the more accurate methods can be better compared.

Scenario	Config	Linear Error		Angular Error		Position Error	
		Abs. $[m]$	Rel.	Abs. $[°]$	Rel.	Abs. $[m]$	Rel.
Street1	A.Grid	0,106	4,10%	4,723	2,39%	0,101	3,92%
	Grid	0,114	4,39%	2,364	1,20%	0,118	4,56%
	Least S	0,172	6,67%	0,782	0,40%	0,171	6,63%
	Exact	0,164	6,35%	-8,224	-4,17%	0,188	7,26%
Street1	A.Grid	0,032	1,25%	4,095	2,08%	0,038	1,47%
	Grid	0,062	2,39%	3,818	1,94%	0,060	2,31%
	Least S	0,035	1,34%	-0,371	-0,19%	0,071	2,75%
	Exact	0,040	1,53%	2,674	1,36%	0,069	2,65%
Street1	A.Grid	-0,030	-1,16%	-5,567	-2,82%	0,039	1,52%
	Grid	0,033	1,29%	-7,538	-3,82%	0,061	2,36%
	Least S	-0,013	-0,50%	-4,736	-2,40%	0,053	2,05%
	Exact	0,038	1,47%	-14,355	-7,27%	0,147	5,67%
Street1	A.Grid	0,744	28,75%	98,769	50,06%	0,579	22,40%
	Grid	0,380	14,68%	9,040	4,58%	0,353	13,65%
	Least S	0,278	10,74%	5,560	2,82%	0,261	10,10%
	Exact	0,405	15,67%	-0,540	-0,27%	0,402	15,53%
Street1	A.Grid	0,025	0,97%	0,624	0,32%	0,140	5,41%
	Grid	-0,049	-1,91%	39,528	20,03%	0,340	13,15%
	Least S	-0,019	-0,73%	21,155	10,72%	0,082	3,16%
	Exact	-0,039	-1,50%	12,153	6,16%	0,210	8,13%
Street1	A.Grid	-0,026	-1,02%	56,624	28,70%	0,023	0,89%
	Grid	0,098	3,80%	-11,296	-5,72%	0,111	4,30%
	Least S	0,060	2,32%	-1,887	-0,96%	0,058	2,25%
	Exact	0,009	0,33%	32,708	16,58%	0,125	4,85%
Street1	A.Grid	0,017	0,66%	4,759	2,41%	0,048	1,86%
	Grid	0,056	2,17%	-1,125	-0,57%	0,057	2,19%
	Least S	0,121	4,68%	-7,379	-3,74%	0,143	5,53%
	Exact	0,321	12,39%	-15,657	-7,93%	0,283	10,94%
Street1	A.Grid	0,356	13,75%	157,178	79,66%	0,312	12,07%
	Grid	0,028	1,08%	-26,727	-13,55%	0,260	10,06%
	Least S	0,048	1,85%	-25,740	-13,04%	0,247	9,55%
	Exact	0,038	1,49%	-2,949	-1,49%	0,055	2,12%
Street1	A.Grid	0,123	4,77%	312,731	158,49%	0,118	4,56%
	Grid	0,031	1,18%	-0,715	-0,36%	0,072	2,80%
	Least S	0,072	2,80%	0,715	0,36%	0,083	3,21%
	Exact	0,094	3,64%	5,672	2,87%	0,104	4,00%
Street1	A.Grid	0,119	4,62%	169,235	85,77%	0,100	3,86%
	Grid	0,022	0,85%	2,334	1,18%	0,017	0,66%
	Least S	0,062	2,40%	-5,032	-2,55%	0,121	4,67%
	Exact	0,047	1,81%	5,659	2,87%	0,045	1,72%
Street1	A.Grid	0,073	2,82%	81,450	41,28%	0,577	22,32%
	Grid	0,068	2,63%	11,040	5,59%	0,068	2,63%
	Least S	0,140	5,42%	1,742	0,88%	0,139	5,36%
	Exact	0,259	10,03%	-3,551	-1,80%	0,270	10,45%

Table 6.3: Deviations of measured final cumulative values from measured values.

Scenario	Config	Lateral Error $[m/s]$		Lineral Error$[m/s]$		Angular Error $[°/s]$	
		μ	σ^2	μ	σ^2	μ	σ^2
Street1	A.Grid	0,018	0,014	0,005	-0,202	0,034	3,441
	Grid	0,004	0,011	0,005	-0,101	0,028	3,317
	Least S	0,010	0,012	0,007	-0,033	0,053	4,253
	Exact	-0,018	0,035	0,007	0,353	0,058	26,677
Street2	A.Grid	0,016	0,013	0,001	-0,175	0,029	4,067
	Grid	0,016	0,012	0,003	-0,164	0,026	3,480
	Least S	0,002	0,017	0,001	0,016	0,037	5,781
	Exact	0,018	0,020	0,002	-0,114	0,030	6,982
Street3	A.Grid	-0,011	0,018	-0,001	0,239	0,032	3,228
	Grid	-0,016	0,024	0,001	0,324	0,030	3,027
	Least S	-0,007	0,020	-0,001	0,204	0,040	4,686
	Exact	-0,059	0,054	0,002	0,616	0,062	10,209
Cobblestone	A.Grid	0,576	0,267	0,032	-4,239	0,193	83,071
	Grid	0,032	0,026	0,016	-0,388	0,125	10,148
	Least S	0,000	0,048	0,012	-0,238	0,108	7,358
	Exact	0,015	0,021	0,017	0,023	0,137	10,444
Lab Dark	A.Grid	0,024	0,023	0,001	-0,027	0,022	5,034
	Grid	0,170	0,107	-0,002	-1,696	0,029	7,539
	Least S	0,153	0,046	-0,001	-0,908	0,031	10,341
	Exact	0,218	0,161	-0,002	-0,521	0,058	31,528
Lab Bright	A.Grid	0,262	0,130	-0,001	-2,430	0,163	21,618
	Grid	-0,070	0,029	0,004	0,485	0,029	3,522
	Least S	-0,029	0,021	0,003	0,081	0,026	3,070
	Exact	0,123	0,047	0,000	-1,403	0,035	8,549
Institute	A.Grid	-0,002	0,042	0,001	-0,204	0,024	3,235
	Grid	-0,026	0,041	0,002	0,049	0,025	3,253
	Least S	-0,071	0,045	0,005	0,317	0,064	9,081
	Exact	0,086	0,254	0,014	0,672	0,106	70,349
Kitchen	A.Grid	0,863	0,387	0,015	-6,746	0,158	33,337
	Grid	-0,102	0,053	0,001	1,147	0,026	4,937
	Least S	-0,095	0,048	0,002	1,105	0,028	4,814
	Exact	-0,020	0,029	0,002	0,127	0,044	7,005
Foyer	A.Grid	1,605	0,671	0,005	-13,422	0,160	31,618
	Grid	0,011	0,022	0,001	0,031	0,027	3,232
	Least S	0,015	0,023	0,003	-0,030	0,025	3,049
	Exact	0,029	0,024	0,004	-0,243	0,037	5,893
Staircase	A.Grid	0,904	0,376	0,005	-7,263	0,161	30,215
	Grid	-0,032	0,023	0,001	-0,100	0,029	4,666
	Least S	-0,065	0,038	0,003	0,216	0,029	4,293
	Exact	-0,014	0,023	0,002	-0,243	0,048	7,684
Grass	A.Grid	0,179	0,234	0,003	-3,495	0,089	56,828
	Grid	0,021	0,052	0,003	-0,474	0,040	4,006
	Least S	-0,001	0,035	0,006	-0,075	0,045	5,630
	Exact	0,004	0,023	0,011	0,153	0,071	13,812

Table 6.4: Deviations of measured velocities from reference velocities

To compare the statistical results, they have been plotted in a form similar to boxplot. Figure 6.22 illustrates an example for the values from Table 6.5. In this example, there are two scenarios (Scenario 1 and Scenario 2) with 2 methods each (Method A and Method B). Method A in Scenario 1 has a mean of $\mu = 0,5$ and a standard deviation of $\sigma^2 = 0,1$. In the plot, the mean is presented by a mark on the horizontal axis at the value of μ, in this case at $0,5$. The standard deviation is represented by the line starting from the mean value and extending to the left to $(\mu - \sigma^2)$ and to the right to $(\mu + \sigma^2)$, from $0,4$ to $0,6$ in this example. All scenarios are listed in a vertical sequence, with every configuration grouped together for each scenario.

Scenario	Config	Example Error	
		μ	σ^2
Scenario 1	Method A	0,5	0,1
	Method B	0,0	0,75
Scenario 2	Method A	1,0	0,75
	Method B	1,0	0,25

Table 6.5: Data for Example Plot

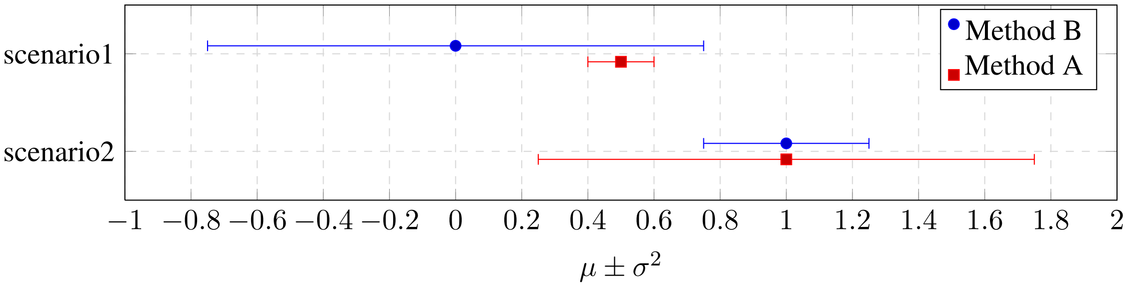


Figure 6.22: Example for Mean / Standard Deviation Plot. The Bar indicates the mean value μ, the line extending goes from $(\mu - \sigma^2)$ to $(\mu + \sigma^2)$

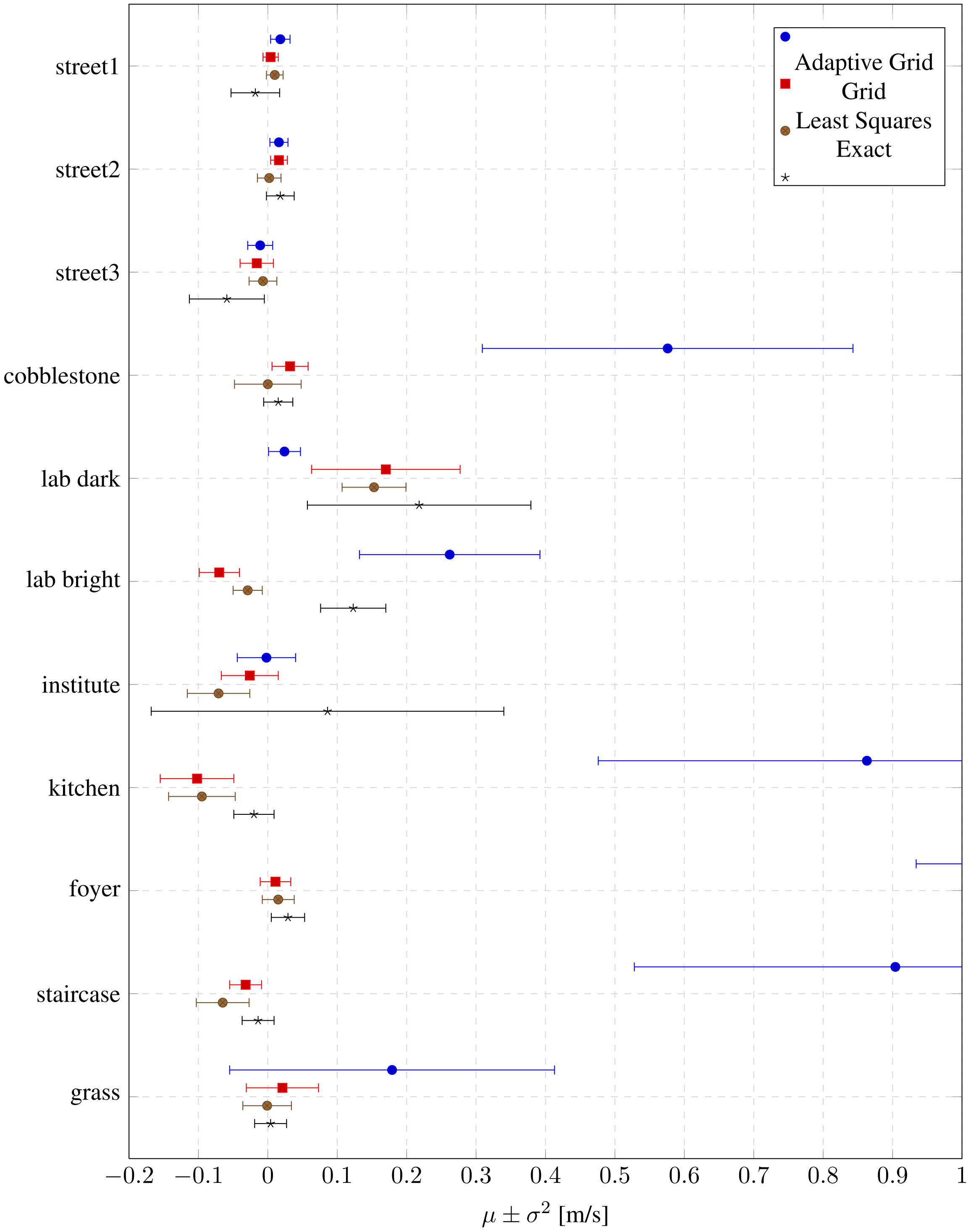


Figure 6.23: Lateral Deviations of Correlation Algorithms. The value range has been truncated such that the more accurate methods can be compared better.

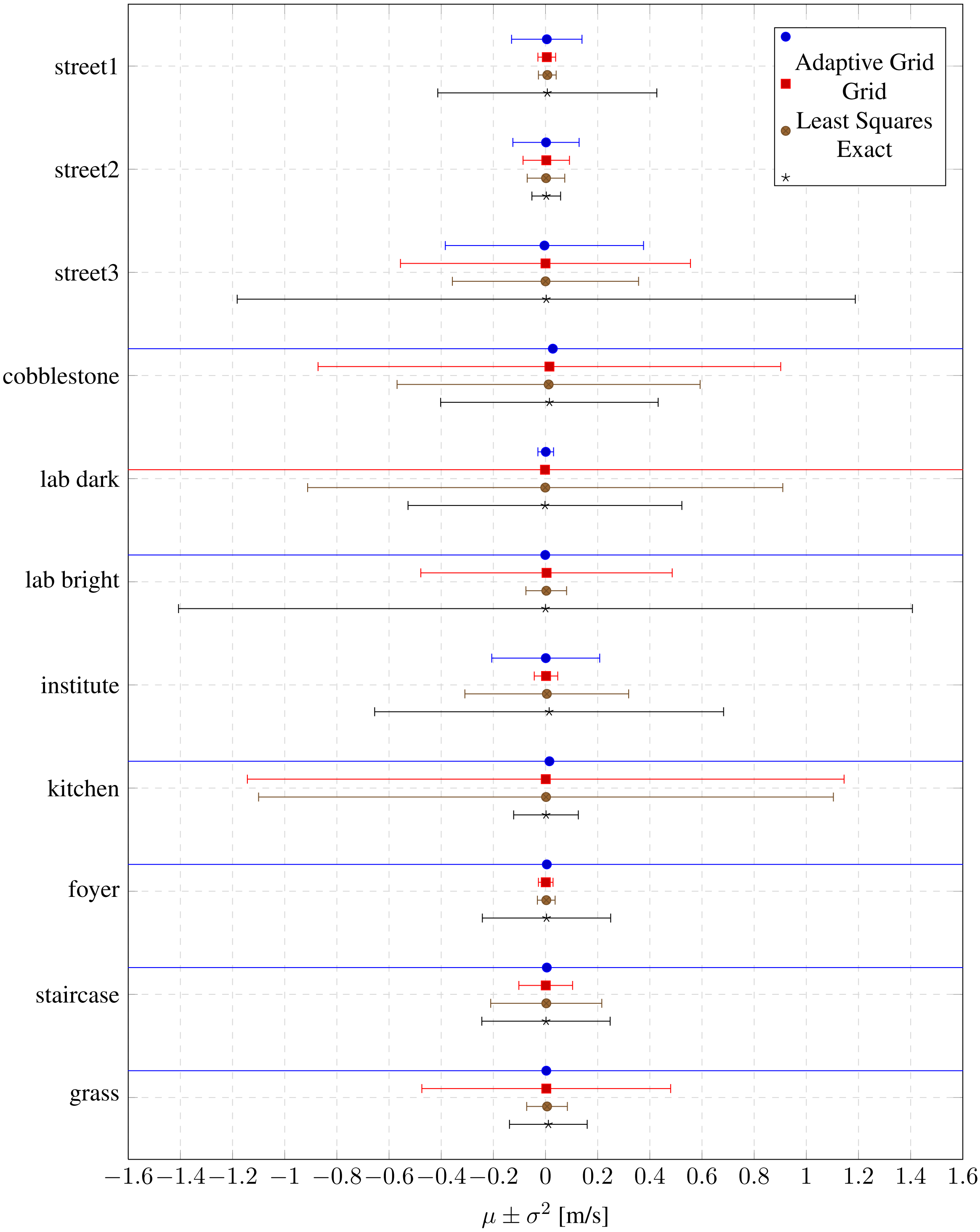


Figure 6.24: Linear Deviations of Correlation Algorithms. The mean values are relatively close to zero, but the standard deviation values vary widely. The value range has been truncated such that the more accurate methods can be compared better.

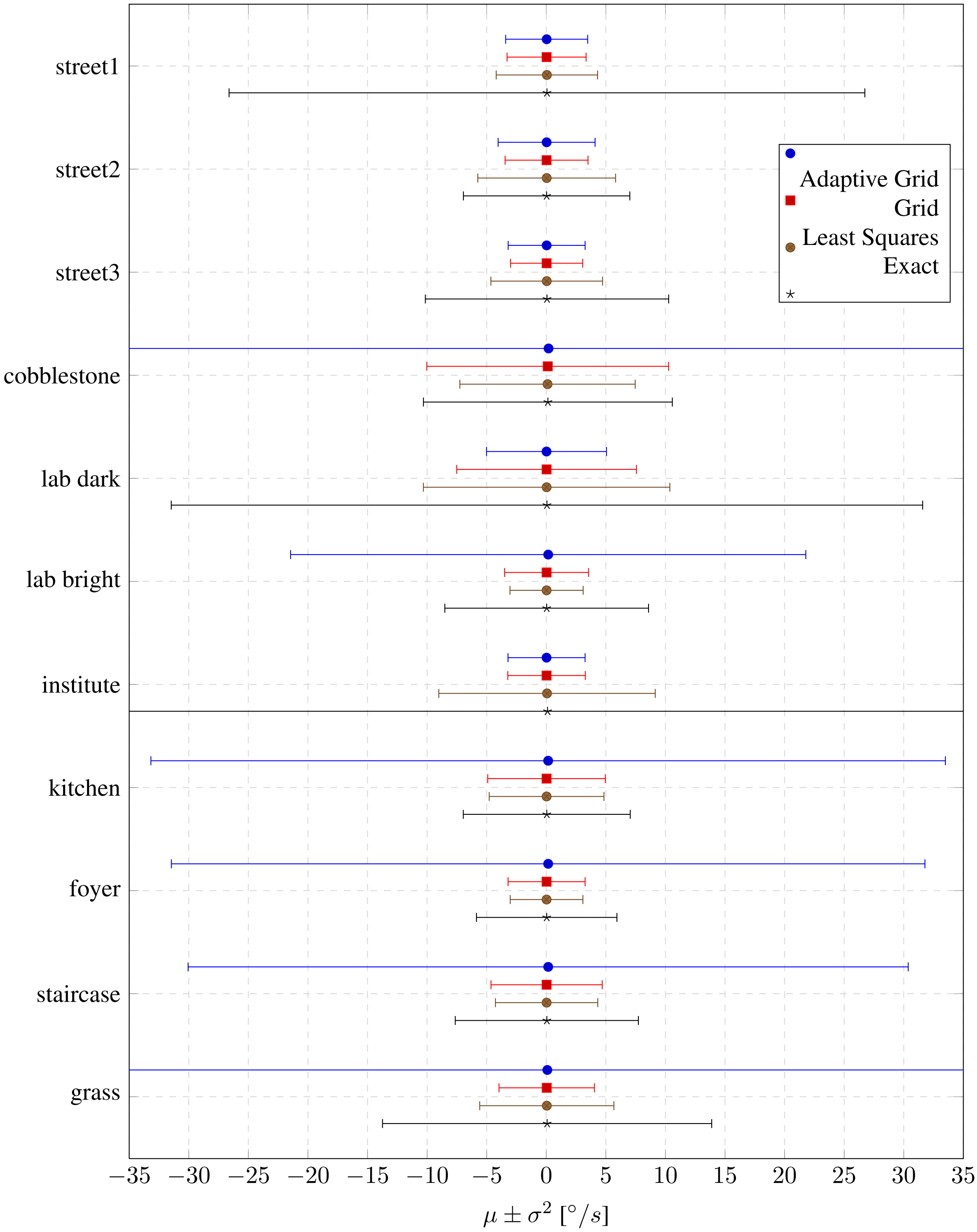


Figure 6.25: Angular Deviations of Correlation Algorithms. The mean values are relatively almost zero for every entry, but the standard deviation values vary widely. The value range has been truncated such that the more accurate methods can be compared better.

From the experiments it becomes clear that the performance of the Frame Algorithms heavily depends on the particular scenario as well. Overall, the Grid method is pretty solid, but especially in the dark scenarios (lab dark, kitchen) it performed worse than others. Surprisingly, the accuracy of the Adaptive Grid method was the worst in many of the scenarios. But precisely in the two dark scenarios, where the normal Grid failed, the Adaptive Grid performed exceptionally well.

Interestingly, the Least Squares method has produced more accurate results than the Grid method in most of the scenarios with a lot less tiles, as well as considerably better computational performance. Probably, this method should be used as a default, with the option to switch to another method if the confidence decreases. Another possibility would be to implement more sophisticated rejection strategies to filter out bad tile results on a frame-by-frame basis for the Grid method.

It was already predicted, that the cobblestone scenario would yield bad measurements due to sever vibrations of the vehicle caused by the uneven structure of the floor. More unexpectedly, the street3 scenario lead to inaccurate measurements as well.

For every other scenario, at least one Frame Algorithm can be found that performs with an accuracy better than 0.2m/s and 5 degree with the used experimental setup, for most scenarios even a lot better. It is expected, that the accuracy can be furthermore improved by using a better camera, as mentioned in subsection 6.3.1, as well as by circumventing or filtering out the jitter mentioned in subsection 6.3.3.

Note, that these statistical scores do not take the confidence value into account. For the most part, the confidence estimation was able to detect bad measurements, such that a following sensor fusion system would have been able to filter outlier measurements with low confidence.

CHAPTER 7

Conclusion

In this work, a novel motion sensor architecture has been developed that is based on image processing methods. To determine the movement parameters, the sequence of images from a camera, which is mounted at the vehicle, pointing to the ground, is analyzed by an image processing algorithm. Additionally, the algorithm also computes confidence values. A wide range of possible configurations has been derived.

The developed sensor was implemented as a software module for the Robot Operation System (ROS). Its implementation details and how the module can be used in ROS works have been described. A software architecture has been developed to support the reconfiguration of the sensor at runtime.

Finally, the sensor has been used in a real application scenario with a research rover. The results have been recorded and analyzed in detail. The main goal was to prove the viability of the developed sensor. The secondary goal was to gather data about the performance of different configurations in different scenarios.

7.1 Major Results

The most important result of this thesis is, that it was in fact possible to develop a new sensor that makes it possible to avoid systematic errors caused by wheel slipping. It was proved that the sensor is viable and for every scenario that was tested. At least one configuration was able to detect linear- and angular velocities with high accuracy. Furthermore, in all cases where it was not possible to achieve high accuracy, the sensor was able to detect the situation and indicate poor results by a low confidence value.

This means that the sensor can be used in a sensor fusion system along with other methods. Systematic errors of wheel based velocity sensors can thereby be covered by the optical sensor as desired.

The sensor system is ready for use in research rovers like the **Pioneer 3-AT** in the tested setup with the PS3 EYE camera and a lot of similar USB cameras supported by *gscam*. Thanks to the implementation as a **ROS**-module, it is easy to integrate it into most existing **ROS** system.

When a better camera is used (see **Future Work**), it is also feasible to deploy the sensor system in normal cars and use it at higher speeds as well.

7.2 Future Work

The implemented software architecture is a solid basis for future research and improvements. The goal of easy reconfiguration at runtime has been met, it is simple to implement new algorithms. The only requirement is that they comply with the defined interfaces. Obviously, future research can be focused on exploring additional algorithms and configurations than presented in this thesis. Furthermore, there are a lot of other topics that could be improved:

- **Using a better camera** In subsection 4.5.1, the maximal measurable speed was presented to be limited by the frame rate via geometrical constraints. It was discovered, that in real applications, the shutter speed further reduces the maximal speed. This was explained in Figure 6.17 with the smearing that occurs in the image, if the vehicle moves too far during the exposure of the image. This effect is sometimes also referred to as *motion blur*.It has been shown that some algorithms like *correlation* are relatively resilient to this blurring, while other methods like *fmt* or *histogram* are heavily impaired by this effect. To combat this, a faster camera would be necessary, but also additional lighting to counteract the brightness loss caused by shorter exposure.
  The methods that suffered from the *motion blur* should be reevaluated as soon as a better camera is used.

- **Illumination:** The scenarios from section 6.2 have displayed a large dynamic range in the image brightness. For some of the indoor scenarios, the lighting conditions already were almost too poor and the image noise caused by high image gain was already visibly lowering the image quality. To achieve more consistent results, artificial illumination is inevitable. Especially, if the frame rate is increased and the shutter time decreased, because these adaptions lead to even darker images. In order to implement illumination, additional hardware is necessary.

- **GPU Implementation:** The largest part of the computational complexity is necessary for the computer vision algorithms. These algorithms are very suitable for implementation on a GPU, because they are highly parallelizable. Implementing them on a GPU would not only boost performance and the achievable frame rate, but it would also reduce the energy impact significantly. Furthermore, CPU time would be freed up for other modules, because the sensor is after all not the only module that is active on an autonomous robot.

- **More sophisticated rejection strategy:** As mentioned at the end of chapter 6, using more tiles than necessary, coupled with more sophisticated rejection strategies could increase accuracy and confidence. The currently implemented rejection strategies rely mostly on the physical limits of the vehicle, which ensures that implausible results are rejected. It would be possible to use more details directly from the images in addition to find even outliers that comply with the physical limits, but are still wrong.

- **Exchange Algorithms at Runtime:** As discovered, for different scenarios, there are specific algorithms that work better than others. This knowledge could be used to adapt the sensor by switching the whole algorithm depending on the current scenario. In section 5.5, it is explained that a frame is processed by one, or potentially more tiles, via Tile Algorithms. It's easy to add new Tile Algorithms to a frame at runtime, as well as to remove existing ones. The frame is analyzed for defined characteristics, and then appropriate algorithms can be chosen for the situation at hand. The findings from chapter 6 can be used as a basis for deciding which algorithm to use in which scenario. This task might also be well suited for the application of machine learning methods.

- **Adaptive Optics:** Besides adaptations to the software of the sensor, hardware adaptations are possible as well. The maximum detectable speed is limited by the physical size of the floor patch perceived by the camera (see subsection 4.5.2). While the minimal speed, in the sense of the minimal movement distance still detected by the sensor, is limited by the pixel size, and therefore, by the resolution of the camera. Simply increasing the FOV and the resolution of the camera only work within small boundaries, and are limited by the characteristics of available cameras. Furthermore, increasing both of these parameters also increases the total number of pixels, and therefore, the computation time which is also undesirable. An optical system with which the zoom factor of the image can be changed dynamically, is able to solve these problems and extend the operation range of the sensor without affecting the computation time. The zoom is controlled by the speed

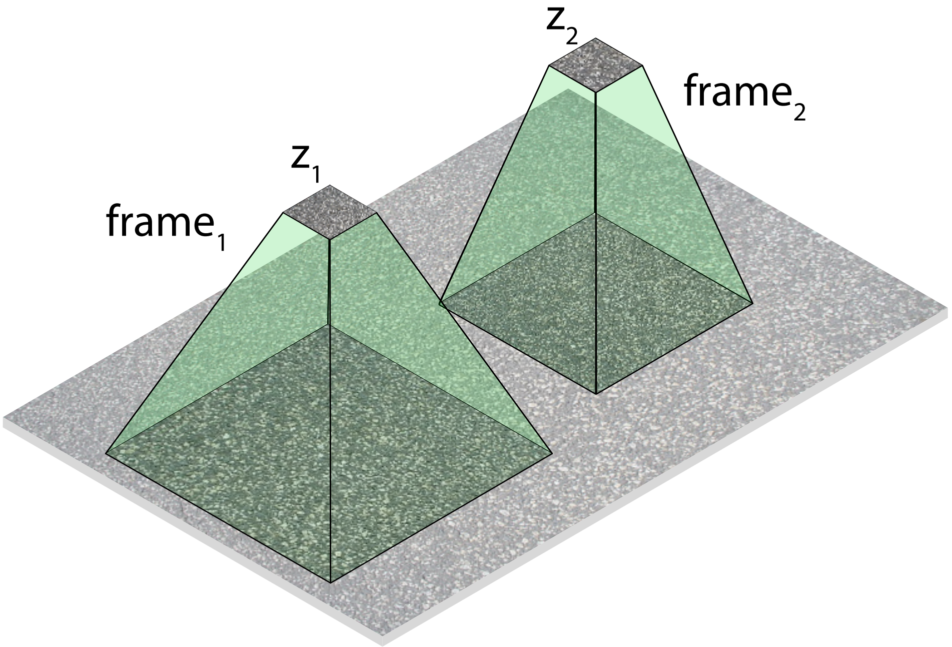


Figure 7.1: Dynamic adaption of zoom: frame 1 has a zoom-factor that captures a bigger *Field of View* than frame 2, but also the image details are smaller in frame 1 than in frame 2.

of the vehicle. If the vehicle is moving slowly, the optical system zooms in to the ground, projecting a much smaller area of the ground to the equally sized image frame. As a result, the distance of a single pixel corresponds to a much smaller distance than before, and the minimal detectable speed decreases. However, if the vehicle moves fast, the optical system zooms out, such that a much larger area is visible in the camera's frame. Thus increasing the maximal speed measurable. Of course, the output value and other values, that are used for adaptation, need to be scaled with the current zoom factor. Figure 7.1 displays the concept, where the zoom-factor in frame 2 is higher than in frame 1. Therefore, frame 1 captures a bigger FOV, but details are smaller. The zoom factor in frame 1 can be used for higher speeds than the zoom factor in frame 2.

APPENDIX A

Acronyms

AGV	Autonomous Ground Vehicle
CPS	Cyper-physical System
ROI	Region of Interest
ROS	Robot Operating System
TMR	Tripple modular redundancy
DLT	Direct linear transform
LTP	Log-Polar Transform
FOV	Field of View
LED	Light Emitting Diode
FPS	Frames per Second
DOF	Degrees of Freedom
SSD	Sum of Square Differences
NCC	Normalized Cross-Correlation
DoG	Difference of Gaussian
SIFT	Scale Invariant Feature Transform
FMT	Fourier-Mellin Transform
RANSAC	Random Sample Consensus

ANMS	Adaptive non-maximal suppresion
FOE	Focus of Expansion
TTC	Time to collision
HD	Heading Direction
SMC	Smoothness Motion Constraint
HMD	Head-mounted Display

Bibliography

[1] Bmw targets 2020 for self-driving cars. `http://www.autoguide.com/auto-news/2013/02/bmw-targets-2020-for-self-driving-cars.html`, October 2014.

[2] Daimler aims to launch self-driving car by 2020. `http://www.reuters.com/article/2013/09/08/us-autoshow-frankfurt-daimler-selfdrive-idUSBRE98709A20130908`, October 2014.

[3] General motors president sees self-driving cars by 2020. `http://www.cnet.com/news/general-motors-president-sees-self-driving-cars-by-2020/`, October 2014.

[4] The google autonomous car. `https://sites.google.com/site/parthvermapaper/home/the-google-car`, April 2014.

[5] gscam in the ros-wiki. `http://wiki.ros.org/gscam`, April 2014.

[6] gscam on github. `https://github.com/ros-drivers/gscam`, April 2014.

[7] Ieee spectrum - how google's self-driving car works. `http://www.ted.com/talks/sebastian_thrun_google_s_driverless_car?language=en`, April 2014.

[8] Opencv homepage. `http://opencv.org/`, April 2014.

[9] Pioneer3at datasheet. `http://www.mobilerobots.com/Libraries/Downloads/Pioneer3AT-P3AT-RevA.sflb.ashx`, October 2014.

[10] Ros camera calibration module. `http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration/`, September 2014.

[11] Ros homepage. `http://www.ros.org/`, April 2014.

[12] Ted talk - sebastian thrun: Google's driverless car. `http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works`, April 2014.

[13] Tesla motors blog: Dual motor model s and autopilot. `http://www.teslamotors.com/de_AT/blog/dual-motor-model-s-and-autopilot`, October 2014.

[14] Sameer Agarwal, Noah Snavely, Ian Simon, Steven M Seitz, and Richard Szeliski. Building Rome in a day. *2009 IEEE 12th International Conference on Computer Vision*, pages 72–79, September 2009.

[15] A. Avizienis. Faulty-tolerant computing: An overview. *Computer*, 4(1):5–8, Jan 1971.

[16] H Badino. A Robust Approach for Ego-Motion Estimation Using a Mobile Stereo Platform. *Complex Motion*, pages 198–208, 2007.

[17] Bruce G. Batchelor. Coming to Terms with machine Vision and Computer Vision: They're not the same.

[18] Irad Ben-Gal. *Bayesian Networks*. John Wiley and Sons, 2007.

[19] A Branca, E Stella, Istituto Elaboraxione, and Via Amendola. Mobile robot navigation using egomotion estimates 3D Motion Estimation. pages 533–537.

[20] Kai Briechle and Uwe D. Hanebeck. <title>Template matching using fast normalized cross correlation</title>. pages 95–102, March 2001.

[21] M. Brown, R. Szeliski, and S. Winder. Multi-Image Matching Using Multi-Scale Oriented Patches. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 1:510–517.

[22] P.J. Burt and E.H. Adelson. The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 31(4):532–540, Apr 1983.

[23] Z Chuan and T Da Long. A Planar Homography Estimation Method for Camera Calibration. *IEEE Internatinal Symposium on Computational Intelligence in Robotics and Automation*, pages 424–429, 2003.

[24] M Cimino and P R Pagilla. Location of optical mouse sensors on mobile robots for odometry. *Robotics and Automation ICRA 2010 IEEE International Conference on*, pages 5429–5434, 2010.

[25] Jiayong Deng and Youngjoon Han. A real-time system of lane detection and tracking based on optimized ransac b-spline fitting. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, pages 157–164, New York, NY, USA, 2013. ACM.

[26] Elan Dubrofsky. *Homography Estimation*. Master thesis, University of British Columbia, 2007.

[27] Wilfried Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2002.

[28] Dave Ferguson. Autonomous automobiles: Developing cars that drive themselves. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 383–383, New York, NY, USA, 2007. ACM.

[29] Hassan Foroosh, Josiane B Zerubia, and Marc Berthod. Extension of phase correlation to subpixel registration. *IEEE Transactions on Image Processing*, 11(3):188–200, 2002.

[30] Willow Garage. Opencv at willow garage homepage. `https://www.willowgarage.com/pages/software/opencv`, April 2014.

[31] Roland Goecke, Akshay Asthana, Niklas Pettersson, and Lars Petterson. Visual vehicle egomotion estimation using the fourier-mellin transform. *IEEE Intelligent Vehicles Symposium*, pages 450–455, 2007.

[32] Lisa Gottesfeld. A Survey of Image Registration. *International Journal of Image Processing IJIP*, 5(4):245–269, 2011.

[33] Kai Hamburger, Thorsten Hansen, and Karl R. Gegenfurtner. Geometric-optical illusions at isoluminance. *Vision Research*, 47(26):3276 – 3285, 2007.

[34] M Hansen, P Anandan, K Dana, G Van Der Wal, and P Burt. Real-time scene stabilization and mosaic construction. *Proceedings of 1994 IEEE Workshop on Applications of Computer Vision*, pages 54–62, 1994.

[35] Chris Harris and Mike Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[36] Richard Hartley and Andrew Zisserman. Multiple View Geometry in Computer Vision.

[37] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, August 1981.

[38] Ho Gi Jung, Dong Suk Kim, Pal Joo Yoon, and Jai Hie Kim. Stereo vision based localization of free parking site. In *Proceedings of the 11th International Conference on Computer Analysis of Images and Patterns*, CAIP'05, pages 231–239, Berlin, Heidelberg, 2005. Springer-Verlag.

[39] RE Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(Series D):35–45, 1960.

[40] Joni-kristian Kamarainen and Pekka Paalanen. EXPERIMENTAL STUDY ON FAST 2D HOMOGRAPHY ESTIMATION FROM A FEW POINT CORRESPONDENCES. 1, 2009.

[41] Kojiro Kato, Kris M. Kitani, and Takuya Nojima. Ego-motion analysis using average image data intensity. In *Proceedings of the 2nd Augmented Human International Conference*, AH '11, pages 9:1–9:4, New York, NY, USA, 2011. ACM.

[42] John Kerr and Kevin Nickels. Robot operating systems: Bridging the gap between human and robot. *Proceedings of the 2012 44th Southeastern Symposium on System Theory (SSST)*, pages 99–104, March 2012.

[43] Sungbok Kim and Sanghyup Lee. Optical Mouse Array Position Calibration for Mobile Robot Velocity Estimation. 2008.

[44] Bryan Klingner, David Martin, and James Roseborough. Street View Motion-from-Structure-from-Motion.

[45] J.C. Knight. Safety critical systems: challenges and directions. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, 2002.

[46] H Kopetz. Automotive Electronics 1. 2:1–9, 1995.

[47] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[48] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[49] Sooyong Lee and Jae-bok Song. Mobile Robot Localization Using Optical Flow Sensors. 2(4), 2004.

[50] Nancy G. Leveson. Software safety: Why, what, and how. *ACM Comput. Surv.*, 18(2):125–163, June 1986.

[51] David Lowe. The computer vision industry. `http://www.cs.ubc.ca/~lowe/vision.html`, April 2014.

[52] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.

[53] BD Lucas and T Kanade. An iterative image registration technique with an application to stereo vision. *IJCAI*, pages 674—-679, 1981.

[54] R.F. Lyon and Xerox Corporation. Palo Alto Research Center. *The Optical Mouse, and an Architectural Methodology for Smart Digital Sensors*. Xerox, Palo Alto Research Center, 1981.

[55] TC Mei. Understanding Optical Mice. *Avago Technologies*, 2006.

[56] Krystian Mikolajczyk and Cordelia Schmid. Scale & Affine Invariant Interest Point Detectors. 60(1):63–86, 2004.

[57] Hans P. Moravec. Towards automatic visual obstacle avoidance. In R. Reddy, editor, *IJCAI*, page 584. William Kaufmann, 1977.

[58] D Morgan. Computer Vision Algorithms and Architectures for an Autonomous Guided Road Vehicle A. pages 711–713.

[59] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS : an open-source Robot Operating System. (Figure 1).

[60] BS Reddy and BN Chatterji. An FFT-based technique for translation, rotation, and scale-invariant image registration. *Image Processing, IEEE Transactions ...*, 5(8):1266–1271, 1996.

[61] By Guillermo Sapiro. Image Inpainting. 35(4):2–3, 2001.

[62] Jignesh N Sarvaiya, Suprava Patnaik, and Kajal Kothari. Image Registration Using Log Polar Transform and Phase Correlation to Recover Higher Scale. *Journal of Pattern Recognition Research*, 7(1):90–105, 2012.

[63] D Sekimori and F Miyazaki. Self-localization for indoor mobile robots based on optical mouse sensor values and simple global camera information. *2005 IEEE International Conference on Robotics and Biomimetics ROBIO*, pages 605–610, 2005.

[64] Optical Mouse Sensor. ADNS-2610 Optical Mouse Sensor data sheet.

[65] Mubarak Shah. FUNDAMENTALS OF COMPUTER VISION1. 1997.

[66] P. Sturm and S Ramalingam. A generic concept for camera calibration. *Eighth European Conference on Computer Vision*, pages 1–13, 2004.

[67] Y. Suzuki, M. Koyamaishi, T. Yendo, T. Fujii, and M. Tanimoto. Parking assistance using multi-camera infrastructure. In *Intelligent Vehicles Symposium, 2005. Proceedings. IEEE*, pages 106–111, June 2005.

[68] Richard Szeliski. Image Alignment and Stitching: A Tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2(1):1–104, 2006.

[69] Richard Szeliski. *Computer Vision : Algorithms and Applications*. 2010.

[70] Bill Triggs. Detecting Keypoints with Stable Position , Orientation and Scale under Illumination Changes. 2004.

[71] Communications United States. Assistant Secretary of Defense for Command, Control and Intelligence. *Global Positioning System Standard Positioning Service Performance Standard*. Assistant Secretary of Defense for Command, Control, Communications, and Intelligence, 2001.

[72] N.F. Vaidya and D.K. Pradhan. Fault-tolerant design strategies for high reliability and safety. *Computers, IEEE Transactions on*, 42(10):1195–1206, Oct 1993.

[73] Jiesheng Wei, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Comparing the effects of intermittent and transient hardware faults on programs. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 53–58, June 2011.

[74] Joachim Weickert, Andr Es, and Christoph Schn Orr. Lucas / Kanade Meets Horn / Schunck : Combining Local and Global Optic Flow Methods. 61(3):211–231, 2005.

[75] George Wolberg and S Zokai. ROBUST IMAGE REGISTRATION USING LOG-POLAR TRANSFORM. *Image Processing, 2000. Proceedings. . . .*, 2000.

[76] Sibel Yenikaya, Gökhan Yenikaya, and Ekrem Düven. Keeping the vehicle on the road: A survey on on-road lane detection systems. *ACM Comput. Surv.*, 46(1):2:1–2:43, July 2013.

[77] Z Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1330–1334, 2000.

[78] Barbara Zitová and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.