Bachelorarbeit

TECHNISCHE UNIVERSITÄT DARMSTADT

Localisation Using Adaptive Feature Selection Strategy

Stefan Boschenriedter

02.11.2015

Betreuer:
M.Sc. Zaijuan Li

REGELUNGSMETHODEN UND ROBOTIK
Prof. Dr.-Ing. J. Adamy

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tage dem Prüfungsausschuss des Fachbereichs Elektrotechnik und Informationstechnik der Technischen Universität Darmstadt eingereichte Bachelorarbeit zum Thema

„Localisation Using Adaptive Feature Selection Strategy“

selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Darmstadt, den 02.11.2015

Stefan Boschenriedter

Abstract

This thesis is about the development and implementation of an adaptive visual odometry algorithm for robot localisation. The relevance of mobile robots has distinctly increased during the past years. For these robots it is essential to know their exact location in the environment. Visual odometry is the method of calculating the egomotion of one or multiple cameras, by only using the visual data provided by these cameras. In this approach, a stereo camera is used to generate 3D to 2D point correspondences. The features get detected and matched with ORB. Then 3D points get triangulated with the linear method. After that, the motion is estimated by the P3P algorithm and a RANSAC outlier rejection. In order to make our algorithm more robust than other common methods, an adaptive feature selection process is introduced. This means, that the robot will search for those areas with most feature points. Then such an area is selected and will be faced. So this avoids the situation that the robot looks at featureless regions. The algorithm will be programmed in C++ using the computer vision library OpenCV. Finally everything gets implemented on a real robot system using ROS. In the end, various experiments are performed to determine the accuracy of the visual odometry algorithm developed in this work.

Zusammenfassung

In dieser Thesis geht es um die Entwicklung und Implementierung von einem adaptiven visuellen Odometrie Algorithmus zur Lokalisierung von Robotern. Aufgrund von neuen Technologien ist der Stellenwert von mobilen Robotern in den letzten Jahren stark gestiegen. Für diese Roboter ist es wichtig, ihre genaue Position in der Umgebung zu kennen. Visuelle Odometrie ist eine Technik zur Bestimmung der Eigenbewegung einer Kamera. Dazu werden lediglich die aufgenommenen Bilder dieser Kamera verwendet. Bei dem Ansatz dieser Arbeit wird eine Stereokamera verwendet, um 3D zu 2D Punkt Korrespondenzen zu generieren. Die Merkmale im Bild werden mit ORB erkannt und übereingestimmt. Die 3D Punkte werden mit der linearen Methode trianguliert. Danach wird die Bewegung mit dem P3P Algorithmus bestimmt. Ausreißer werden dabei mit RANSAC heraus gefiltert. Damit unsere Vorgehensweise noch robuster ist als übliche Methoden, wird zusätzlich noch eine adaptive Merkmalserkennung eingeführt. Das bedeutet, dass der Roboter die Bereiche mit den meisten signifikannten Merkmalen in seiner Umgebung sucht. Dann richtet der Roboter sich zu einem solchen Bereich aus. Damit wird verhindert, dass der Roboter in Richtungen schaut, in denen es keine Punkte zur Orientierung gibt. Der Algorithmus wird in C++ programmiert und benutzt dabei die Bildverarbeitungsbibliothek OpenCV. Im Anschluss wird das gesamte Programm auf einem echten Roboter implementiert, wobei ROS verwendet wird. Zum Schluss werden verschiedene Experimente durchgeführt, um die Genauigkeit vom visuellen Odometrie Algorithmus zu bestimmen.

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, autonomous vehicles were getting more and more relevant in science and also in our daily lives. There are many capabilities of such vehicles. One field of application is the exploration or work in dangerous environments like burning structures, nuclear contaminated areas or even other planets. Another example of utilisation are service robots that can support invalid or elderly people. Furthermore, many accidents on the streets could be prevented by self-driving cars. These are just a few of many different possibilities. This is why it would be beneficial for us to develop well performing autonomous vehicles in the future.

One significant aspect of this development is a robust and accurate self-localisation of these robots. In order to fulfil different tasks, a correct location is required for navigation. Today, there are already many different possibilities of estimating the egomotion. One of the most common techniques is the global positioning system (GPS). While it is working quite good outside, it can not really be used in indoor environments. Furthermore, it is only accurate within a few meters, so GPS might not be suitable for every application. Two other options for localisation would be inertial measuring units (IMUs) or wheel speed sensors. However, reliable IMUs are quite expensive and wheel odometry only works if there is no wheel slippage, which can not always be ensured.

Since the human eye is one of the most important sensors for human orientation, it is also imaginable to take advantage of a camera for localisation. The process of determining the egomotion from an image sequence is called Visual Odometry (VO). Cameras are becoming more accurate and cheap during the past few years, which makes them a suitable sensor for autonomous vehicles. So this thesis will be about implementing a visual odometry algorithm for self-localisation. An adaptive feature selection strategy will be applied to improve the results of the VO.

1.2 Thesis Overview

For this work, a mobile robot system by Festo Didactic [1] is going to be used. It is called "Robotino" and it is further equipped with a set of stereo camera from VRmagic for supplying the video data. The image processing will be performed by a standard consumer grade notebook that is mounted on the Robotino. The robot will be controlled by a program, which is implemented in the Robot Operating System (ROS). The VO algorithm will be programmed in C++ using the computer vision library OpenCV.

This thesis will provide all information on how the localisation algorithm works and how it is implemented. It consists of the following five chapters:

1. The first chapter gives a short introduction to the topic and illustrates the structure of the thesis.
2. This section provides basic information and concepts that are used in computer vision, which includes the image formation, camera calibration, rigid body motions and the geometry of multiple views.
3. The main chapter of this work will explain how the whole algorithm works. It starts with general feature detection and matching. Then the actual VO algorithm will be introduced. After that, the adaptive feature search will be described. Finally the robot control is illustrated.
4. In this section, the hard- and software that are used for the experiments will be presented in detail. Furthermore, it is shown how the localisation algorithm is implemented and how it performs in our tests.
5. The last part is concluding the results and gives an outlook of what can be still done in this research area.

1.3 Previous Work

In literature there are already many different approaches of localising vehicles and mobile robots. As mentioned above, the GPS is one of the most popular ways of determining the own position. It is mainly used for outdoor navigation and it works by receiving information from space satellites [2]. So when the signal is lost, the GPS will fail, which can easily happen by entering closed structures like buildings or tunnels. Since our mobile robot should operate in closed rooms, GPS is not eligible for the localisation task in this case.

For indoor navigation, there are laser range scanners, which are often utilised to create a high resolution map of the environment by scanning the whole scene. These LIDAR (portmanteau of "light" and "radar") systems are sending laser rays in all directions to estimate their distance towards surrounding objects by analysing the reflected light [3]. This information is then used for determining the own position in a room.

Another popular localisation approach is the egomotion estimation by inertial measurement units (IMUs), which are usually consisting of accelerometers and gyroscopes. By integrating the acceleration over time, the current velocity and position are determined. This method is also known as dead reckoning. So the position and orientation is calculated by adding up the currently measured values to the previously estimated pose [4]. This is why measurement errors are propagating greatly over time when using this technique. In order to get feasible results, the accuracy of the sensors has to be really good, which makes IMUs quite expensive.

Further, there is another way of calculating the own position based on previous and current motions. Visual Odometry employs a camera as the measuring sensor and applies various image processing algorithms for self-localisation. Even if image processing and computer vision are relatively new research areas, a lot of useful concepts and implementations have been published already. One prominent application of visual odometry is its utilisation for the Mars rovers by the NASA [5].

The general topic of visual odometry is extensively explained and discussed in [6] and [7]. Furthermore there are also many different implementations tested in the literature. [8] illustrates a monocular approach, while [9], [10] and [11] are working on different methods with two cameras.

In addition to that, there are also some other important projects to note, like "Photo Tourism" [12]. This is about reconstructing a 3D world scene out of many pictures that were taken by some tourists. This technique is also referred to as structure from motion (SfM), since the structure of the real world is reconstructed by pictures from different positions (motion between images).

Chapter 2

Preliminaries

This chapter will introduce some basic techniques that are used for image processing in general. It will start with a model of the camera and how images are created. Some basic mathematical concepts will be described as well. The following part is about the camera calibration and how it can be used in further computations. The last section will illustrate the geometric relations between two different camera images.

2.1 Image Formation and Camera Model

2.1.1 Pinhole Camera

The basis of all computer vision approaches are images taken by a camera. In order to use these pictures for different applications, it is essential to know what images are exactly. So at the beginning, the process of forming the reality of the world into an image will be explained. Therefore, the so called pinhole camera model is introduced [13]. Figure 2.1 illustrates how it works.

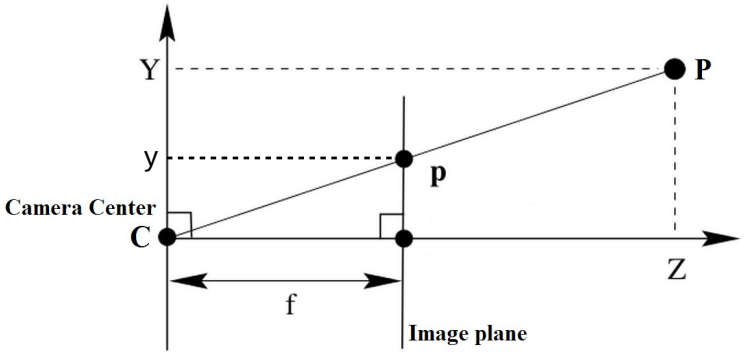

Figure 2.1: Pinhole Camera Model

The sensor of the camera measures the light rays that are coming from the 3D point P. The image of this point is formed in the image plane. For simplicity we will first have a

look at a 2D example. The point $P = (Y,Z)^T$ with the distance Z to the camera center and the position Y is mapped to the point $p = (y)$ in the image. The focal length of the camera is f. According to the intercept theorem, the following relation holds[14]

$$\frac{y}{Y} = \frac{f}{Z}. \tag{2.1}$$

This formula takes into account that the focal length f is very small compared to the distance Z of the point P to the camera. This assumption holds true for most cases in reality, since the focal length is just a few centimetres long. So the transformation from world (3D) to image (2D) would be written as:

$$(X,Y,Z) \rightarrow (\frac{f}{Z}X, \frac{f}{Z}Y) \tag{2.2}$$

This model is called ideal pinhole camera model. All through this work homogeneous coordinates will be used for simplicity, which are widely used in the projective geometry. This means, that the coordinates are augmented with an additional coordinate, which contains a scaling factor for this point [15]. So a 2D point in Cartesian coordinates (x,y) would be $(u,v,w) = (wx,wy,w)$ in the homogeneous representation.

Furthermore, the intercept theorem equations relate a real world point to a point on the camera image, using the camera coordinate system. For image processing it is important to take advantage of the image coordinates in pixels that start with (0,0) in the upper left corner of the image. To translate the center of the optical axis to the center of the image coordinate frame, there will be an offset (o_x,o_y) for the x- and y-coordinates.

All together, the equation that transforms a real 3D world point $\mathbf{X}$ into a 2D image point $\mathbf{x}$ will be:

$$\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & o_x \\ 0 & f & o_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \Leftrightarrow \lambda \cdot \mathbf{x} = \mathbf{K} \cdot \mathbf{\Upsilon} \cdot \mathbf{X} \tag{2.3}$$

where $\mathbf{K}$ is the camera matrix and Υ is simply a 3×4 identity matrix, which is important for modelling rigid body motions of the camera. λ is a scaling factor for the image point. So for now, we have an equation that maps a real world point to a 2D point in the camera image.

2.1.2 Rigid Body Motion

In many cases, it is desired to describe the world coordinates relative to a real world system instead of the camera frame. For example, the position of a robot should be

estimated relative to the room, but not relative to a camera which is filming the robot. In order to do that, the camera coordinates system gets transformed into the world coordinates system. That is called an Euclidean transformation or a rigid body motion.

When doing such a transformation, there are six degrees of freedom: First the system can be moved (translated) in the x- ,y- and z-direction. This translation is written as the 3×1 translation vector $\mathbf{t}$. Furthermore, the coordinate system can be rotated around all three axes. Such a rotation can be written as a rotational 3×3 matrix $\mathbf{R}$.

$$\mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{pmatrix} \tag{2.4}$$

The rotation matrix is represented by making three consecutive rotations around the coordinate system axes with the angles Φ, Θ and Ψ [16]. These three rotation angles Φ, Θ and Ψ form a 3×1 rotation vector $\mathbf{r}$. However, for most calculations the 3×3 rotation matrix $\mathbf{R}$ is used. With the help of the rodrigues formula, it is possible to transform $\mathbf{r}$ into $\mathbf{R}$ and vice versa [17]. The entries $r_1 ... r_9$ of $\mathbf{R}$ are calculated by multiplying the three rotational matrices around each axis.

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\Phi) & sin(\Phi) \\ 0 & -sin(\Phi) & cos(\Phi) \end{pmatrix} \cdot \begin{pmatrix} cos(\Theta) & 0 & -sin(\Theta) \\ 0 & 1 & 0 \\ sin(\Theta) & 0 & cos(\Theta) \end{pmatrix} \cdot \begin{pmatrix} cos(\Psi) & sin(\Psi) & 0 \\ -sin(\Psi) & cos(\Psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

So when having the rotation matrix and translation vector, the point $\mathbf{X}_C$ in the camera coordinate system can be transformed into the point $\mathbf{X}_W$ relative to the world coordinate system by applying

$$\mathbf{X}_W = \mathbf{R} \cdot \mathbf{X}_C + \mathbf{t}. \tag{2.6}$$

Figure 2.2 shows such an euclidean transformation from camera to world coordinate system.

In the homogeneous representation, the transformation $\mathbf{g} = (R,t)$ can be written as the following

$$\bar{\mathbf{g}} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.7}$$

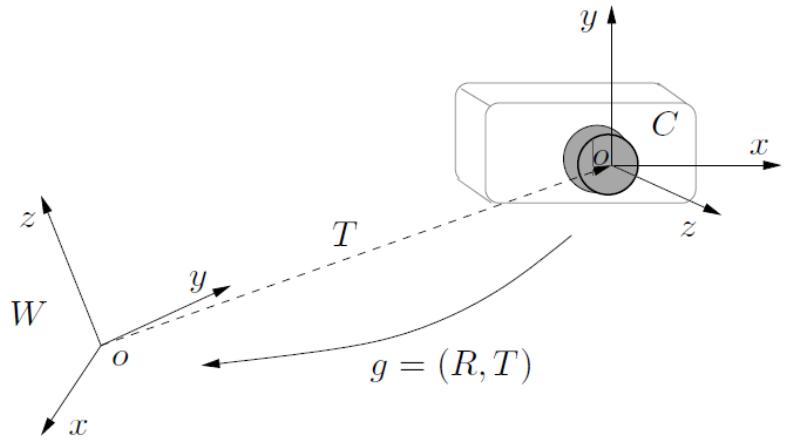

Figure 2.2: Euclidean transformation [18]

Knowing about these transformations, they can be applied to the camera projection equation (2.3). So when replacing $\boldsymbol{\Upsilon}$ with $\mathbf{g}$, we finally get a relationship between the 3D coordinates of a point $\mathbf{X}_0 = (X_0,Y_0,Z_0)$, that is relative to the defined world frame (and not the camera frame any more), and their projected points in the camera image $\mathbf{x}_0 = (x_0,x_0)$:

$$\lambda \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & o_x \\ 0 & f & o_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \end{pmatrix} \cdot \begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{pmatrix} \Leftrightarrow \lambda \cdot \mathbf{x}_0 = \mathbf{K} \cdot (\mathbf{R}|\mathbf{t}) \cdot \mathbf{X}_0 \quad (2.8)$$

2.2 Camera Calibration

2.2.1 Camera Intrinsics and Extrinsics

In the previous chapter, it was shown how an image point is formed by a point in the real world. In order to do this calculation, the camera matrix $\mathbf{K}$, which contains the focal length (f) and the optical center point (o_x,o_y) of the camera, is required. Furthermore, the rigid body transformation $\Upsilon = (R|t)$ which describes how the camera system is arranged relative to the world system, needs to be found out.

In this context, the camera matrix entries are called intrinsic parameters, because they describe the specific values inside a certain camera, while the $(R|t)$ values are called

extrinsic parameters, since they are determining the external pose of the camera relative to the world frame.

In order to estimate these intrinsic and extrinsic parameters, a camera calibration [14] is performed. To do so, pictures of a known plane object which is called calibration rig are taken. In most cases, a checkerboard is used for the calibration. A corner detection algorithm finds all the corners of the black and white squares of the checkerboard in the image. This generates a set of image points $\mathbf{x_i}$ that corresponds to the corners of the checkerboard. Furthermore, the square size of the checkerboard should be known, so a set of the real world coordinates of all the checkerboard corners $\mathbf{X_i}$ can be built. Traditionally, the bottom left corner of the checkerboard is defined as the world coordinate system center. In this case, all the corner points $\mathbf{X_i}$ will be depicted as $(X_i,Y_i,0)$.

Having two sets of image points $\mathbf{x_i}$ and corresponding real world points $\mathbf{X_i}$, both the intrinsic and extrinsic parameters [19] can be calculated. This is done by inserting these values in equation 2.8 and solve for all unknown parameters.

2.2.2 Full Camera Model with Distortion

In all previous calculations, the ideal pinhole camera model was utilised. In reality, there is not a hole at the projection center of a camera, but a thin lens. This lens leads to (mainly radial) distortion in the image [20]. There are different types of distortion that can modify an image. The most important effects are called barrel and pincushion distortion due to their characteristic form. Figure 2.3 illustrates how it looks like.

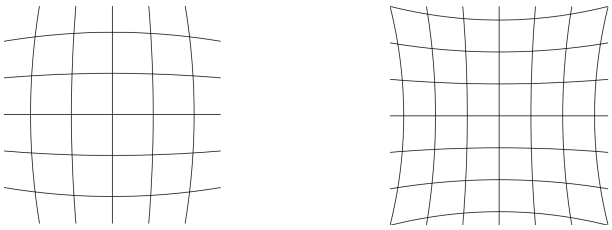

Figure 2.3: Barrel (left) and Pincushion (right) Distortion

Especially at the edges of the image the distortion effect is pretty large. So it also influences the image formation and therefore needs to be included in the calculations that relate image and real world points. Thus Zhang [19] introduced the distortion coefficients $\mathbf{k} = (k_1,k_2)^T$ that model the effect of the distortion like the following:

$$\breve{x} = x + x[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \tag{2.9}$$

$$\breve{y} = y + y[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \tag{2.10}$$

In this equation, ($\breve{x}$,$\breve{y}$) are the real image coordinates with distortion, while (x,y) is the ideal image point (without distortion). These coefficients (k_1,k_2) also belong to the intrinsic camera parameters and get calculated during the calibration process as well.

A nonlinear optimization with the Levenberg-Marquardt algorithm [21] finds a camera matrix $\mathbf{K}$, distortion coefficients $\mathbf{k}$ and a rigid body motion ($\mathbf{R}|\mathbf{t}$) that minimizes the so called "reprojection error"

$$\sum_{i=0}^{n}\sum_{j=0}^{m} \|m_{ij} - \breve{m}(K,k,R_i,t_i,M_j)\|^2 \tag{2.11}$$

where n is the number of images and m is the number of known points in the calibration rig. m_{ij} is the image point j in the image i while $\breve{m}$ is the projection of point M_j in image i according to the projection equation 2.8 and including the distortion coefficients $\mathbf{k}$.

So this algorithm estimates those intrinsic and extrinsic parameters, which make the best match for all the taken images. The more pictures (from different positions and angles) of a calibration rig are used, the better results will be obtained by this optimization. Furthermore, when the distortion coefficients have been determined, it is possible to remove the distortion of the pictures taken by the camera. After doing that, the disruptive effect of distortion can be eliminated for further calculations.

2.2.3 Stereo Calibration

For this work, a set of stereo cameras is used instead of a monocular one. It has important advantages for scene reconstruction and localisation which will be described later. The two cameras of a stereo pair are mounted rigidly on the robot, so the relative pose between both cameras will not change during the experiments. In order to process the data from the two cameras correctly, it is important to know this euclidean transformation ($R|t$) between the two cameras. In most cases, the cameras are mounted in a way, that the pose is only a lateral translation, so $\mathbf{R} = \mathbf{I}$ and $\mathbf{t} = (b,0,0)^T$. The distance b between the two cameras is called baseline. For estimating the exact transformation ($R|t$), a stereo calibration can be performed [15]:

First, both cameras are getting calibrated individually to receive their intrinsic parameters. After that, the two cameras take pictures of a known calibration rig at the same time. The external parameters of both cameras towards the calibration rig can be computed with these images. Then these parameters can be used to calculate the pose ($R|t$) between both cameras. In this thesis, the position of the left camera will be defined as the center of the camera frame.

2.3 Hand-Eye-Calibration

When localising the position of the robot, the information that is provided by the cameras which are mounted on the robot itself is used. Due to the fact that the cameras are fixed on a certain position of the robot, the camera center cannot be treated as the center of the robot. This has to be taken into account for navigating the robot. Otherwise, there might occur collisions with some obstacles in the environment, because the robot already bumps into something, that is not (yet) touching the camera.

To avoid obstacles a "Hand-Eye-Calibration" must be performed. This refers to the relation between the "hand" and the "eye" of a system. In our case, the eye is the camera that observes the scene and the hand is the robot that is holding the camera. So this calibration measures the relative pose between the camera coordinate system and the robot coordinate system. On account of the fact, that the camera is mounted rigidly on the robot, this relation is also a rigid body transformation $(R|t)$, as described in the previous sections.

Knowing exactly where the camera is placed on the robot, the position of the robot itself is known as well, if the cameras pose was estimated previously [22]. That makes it possible to control the robot, based on its real position by simply using the pose of the camera.

The Hand-Eye-Calibration works as follows: A calibration rig (e.g. a checkerboard) is placed in a room and defines the world coordinate system center. Then the robot is positioned somewhere around this calibration rig, so the camera is facing it. After that, the exact pose (distance in (x,y,z) and orientation (Φ,Θ,Ψ)) of the robot center relative to the world coordinate frame is measured. This transformation is going to be called $(R|t)_W^R$ ($(R|t)$ from the world to the robot). This measurement is done manually and might cause some inaccuracies within a few millimetre or centimetre. Therefore, we measured many different robot poses to improve the results by calculating an average later. If both the robot and the calibration rig are placed on a plane ground, the measurements that have to be made are reducing from six parameters (three translations and three rotations), to only three parameters (x and y translation and the rotation around the z-axis).

Having measured $(R|t)_W^R$, the relative pose from the camera to the world frame $(R|t)_C^W$ is required next. This can easily be estimated by taking a picture of the calibration rig with that camera. Then a calibration program can be used to calculate the extrinsic parameters of the (previously calibrated) camera relative to the world frame, which is the calibration rig. Now having $(R|t)_W^R$ and $(R|t)_C^W$, the relation between camera and

robot $(R|t)_C^R$ is derived by a simple multiplication:

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_C^R = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_C^W \cdot \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_W^R \tag{2.12}$$

Figure 2.4 shows how these rigid body transformations work.

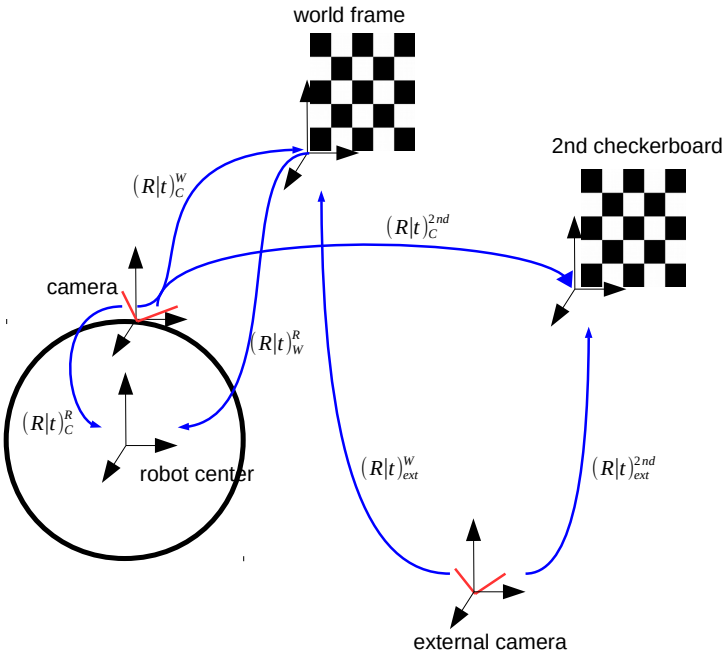

Figure 2.4: Transformations used for the Hand-Eye-Calibration

As mentioned before, the manual measurement of the robot center pose relative to the world frame $(R|t)_W^R$ is not very exact. Whereas the measurement of the cameras pose which is done by calculating the extrinsic parameters from an image, is quite accurate. So in order to get better results for the Hand-Eye-Calibration, we made five measurements in different robot poses. For each pose, there was a picture taken of the calibration rig, which leads to five different camera to robot transformations. Calculating the average of these five $(R|t)_C^R$ values, this might give a better result concerning the manually measurement uncertainties. In order to get even more values for $(R|t)_C^R$, an external camera and a 2nd calibration rig was introduced (see Figure 2.4).

The external camera takes an image of both checkerboards. Then the pose of the external camera relative to the 2nd checkerboard $(R|t)_{ext}^{2nd}$ and to the world frame checkerboard $(R|t)_{ext}^W$ is calculated. With those transformations, the pose between the two checkerboards is known.

For each of the five robot poses, the robot camera also takes a picture of the 2nd checkerboard. So the transformation $(R|t)_C^{2nd}$ can be estimated using that image. Having all these relative positions to each other, it is possible to calculate the desired pose between camera and robot $(R|t)_C^R$. In this case, that pose is determined with the additional help of the 2nd checkerboard and not only directly over the world frame checkerboard. According to Figure 2.4 this calculation can be represented as

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_C^R = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_C^{2nd} \cdot (\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_{ext}^{2nd})^{-1} \cdot \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_{ext}^{W} \cdot \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_W^R. \tag{2.13}$$

Taking pictures of five different positions of the 2nd checkerboard for each of the five different robot poses that were measured, there is a total of $5 \cdot 5 = 25$ estimations for $(R|t)_C^R$. After taking the average of all these 25 values, this should give a quite good result for the Hand-Eye-Calibration.

2.4 Geometry of Stereo Views

In chapter 2.1 it was figured out, that when taking an image of a scene, a 3D world point X_0 is projected to a 2D point x_0 in the picture. This relation is given by

$$\lambda \cdot \mathbf{x}_0 = \mathbf{K} \cdot (\mathbf{R}|\mathbf{t}) \cdot \mathbf{X}_0. \tag{2.14}$$

Due to the transformation from 3D to 2D, the information about the depth of a scene is lost. Furthermore, with only a single image it is not feasible to recover the scaling factor λ. That is why it is impossible to calculate a real 3D world point, having only one picture of a scene. However, it is essential to know about 3D scene structure for the localisation of a robot in an unknown environment. Fortunately, it is possible to recover this depth information from multiple images, which will be described in this section.

As a human, we are able to see in 3D, which means that we can estimate the depth of the world around us. This is possible due to a simple fact: We are not only having one, but two eyes. So two eyes will also create two images of the same scene with the same 3D world points, but from different views. Finding a point in the image of the left and the right retina, our brain can recover the depth of this point in the world. Exactly the same works as well, when using two cameras (stereo cameras). Figure 2.5 shows the relations of the two view geometry.

Knowing the exact distance and orientation between the two cameras (rigid body transformation $(R|t)$), the image points x_1 and x_2, that correspond to the same real world

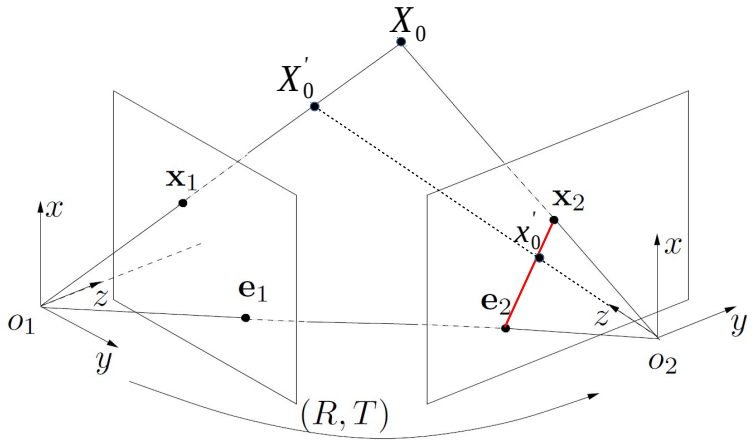

Figure 2.5: Two view geometry [18]

point X_0, need to be found. Then the point X_0 can be estimated by triangulation with that information. There are several ways of triangulating in computer vision, which will be further described in chapter 3.3.2. Though, the basic principle is to solve for the intersection point of the two lines that connect the camera centres o_1 and o_2 with the image points x_1 respectively x_2. So with that technique, it is possible to calculate the 3D coordinates of the point X_0. This is how depth data of the scene can be obtained by using two cameras. The only thing that needs to be known before is the relative pose $(R|t)$ between the two cameras that are taking images. This can be found by doing a prior stereo calibration as described in chapter 2.2.3.

In case, where the two considered images are taken by a stereo camera, the $(R|t)$ will always be known. The other case of two view geometry is, having two images taken by the same camera. When moving a single camera, this also provides two (or more) images of the same scene. Now the problem is, that it is unknown how the camera pose has changed during the two pictures. However, it might be useful to know this consecutive motion. In order to recover this $(R|t)$ from two images, the epipolar geometry is taken into account, which will be explained in the next chapter.

2.4.1 Epipolar Geometry

This section is about how the poses of a camera, that took two images from different views, are related. First, it needs to be determined, how the image points x_1 and x_2 of the first and second picture, that correspond to the same scene point X_0, are related to each other. The following equation describes this relationship and is known as the

epipolar constraint [23]:

$$x_2^T \mathbf{F} x_1 = 0 \tag{2.15}$$

$\mathbf{F}$ is a 3×3 matrix and is called fundamental matrix. This fundamental matrix $\mathbf{F}$ can be calculated, by taking advantage of multiple corresponding image points $x_{1,i}$ and $x_{2,i}$. For example, having eight of these point correspondences, the eight-point-algorithm, which is described in [18] can be used to find $\mathbf{F}$.

Knowing that x_1 and x_2 are formed by a multiplication of the camera matrix $\mathbf{K}$ and the rigid body motion $(R|t)$ with the real world point X_0, it can be shown that the fundamental matrix can be also written as

$$\mathbf{F} = \mathbf{K}^{-1^T} \cdot [\mathbf{t}]_\times \cdot \mathbf{R} \cdot \mathbf{K}^{-1} \tag{2.16}$$

where $[\mathbf{t}]_\times$ is the matrix representation of the cross product with $\mathbf{t}$ [24].

The important thing to notice is that the fundamental matrix consists of the (inverted) camera matrix and the rigid body motion $(R|t)$ which the camera has done between the two pictures. After computing the fundamental matrix $\mathbf{F}$ with some point correspondences like described above, it is possible to extract the $(R|t)$ by singular value decomposition (SVD) [24]. Therefore, the camera matrix must be known as well. So in the end, the motion of the camera can be recovered by using two consecutively taken pictures. This works with a single camera and becomes very useful since it is possible to triangulate 3D points using that information or even calculate the trajectory of the camera.

Another important application of the fundamental matrix is that it can generate epipolar lines [23]. In Figure 2.5, there is a line connecting the two camera centers o_1 and o_2. The points e_1 and e_2, where this line interacts with the image planes, are called epipoles. When a point X_0 is found in the left image (image point x_1), the next step is to find the corresponding point x_2 in the right image. To reduce the computational cost of finding this point, one does not need to scan the whole image. Having x_1 implies that the point X_0 must lie somewhere on the line which goes through o_1 and x_1. On this line, there are many possible positions for X_0 like X_0' that would fit for the left image. If all these possible positions are projected to the right image, they are all lying on a line that goes through the epipole e_2 (red line in Figure 2.5). Knowing this line, the search of the point x_2 in the right image can be reduced to one dimension instead of the whole image. This makes it easier and quicker to find x_2. This line l_2 is called epipolar line and can be simply calculated with the help of the fundamental matrix $\mathbf{F}$:

$$\mathbf{l}_2 = \mathbf{F} \cdot \mathbf{x}_1 \tag{2.17}$$

Figure 2.6 shows an example of how the epipolar lines work. Two images of the same object were taken from different positions. An OpenCV function calculates the fundamental matrix and draws the epipolar lines on the images. It can be seen that all corresponding image points are lying on specific epipolar lines.

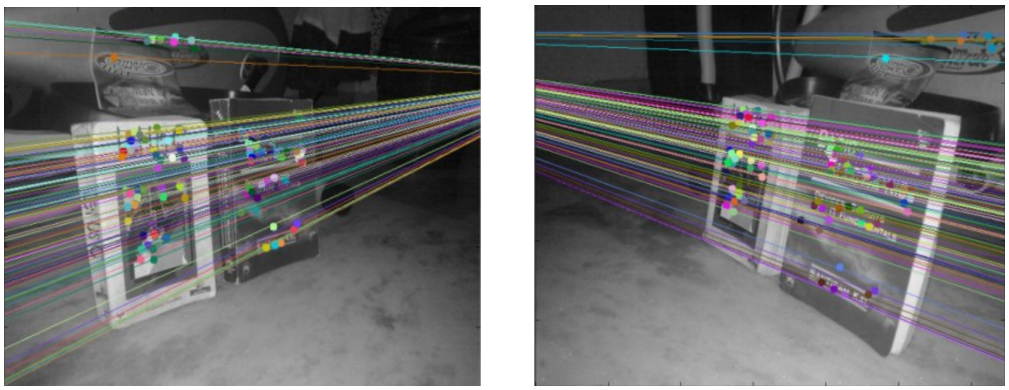

Figure 2.6: Two images of the same object with epipolar lines drawn with OpenCV [25]

2.4.2 Image Rectification

After knowing about epipolar geometry and the fundamental matrix, these techniques can be exploited to further accelerate the search of image points that belong to the same world point. In order to find such a corresponding point, one only needs to search along the previously computed epipolar line. The pose between both cameras of a stereo rig is not changing over time, therefore this also yields a fixed fundamental matrix and fixed epipolar lines. There is the possibility to rotate an image in such a way, that all of these epipolar lines are horizontal and parallel to each other. In this case, all corresponding points of two images will have the same vertical coordinate and so we only need to search along the x-axis. This image transformation that makes all epipolar lines of the left and right image parallel to the horizontal axis is called rectification. It is further explained in [26]. This section will describe how the image rectification works.

For rectifying one image, there are mainly two steps: The first one is to rotate the image, so that the optical axis of the image is perpendicular to the baseline (the line connecting the two optical centres of the stereo cameras O and O'). This perspective transformation is called H_1 which is a 3×3 rotation matrix. Figure 2.7 is illustrating the setting.

After this first transformation, the image plane will be parallel to the baseline. In the second step, this image plane gets twisted around the optical axis, so that the epipolar lines

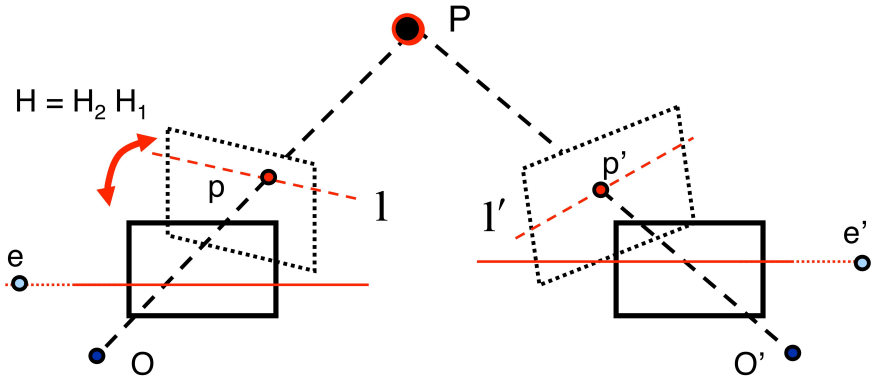

Figure 2.7: Principle of rectification [27]

are parallel to the baseline. This rotation is called H_2 and it will move the epipole e to infinity on the x-axis, so it will have the homogeneous coordinate $e = [1,0,0]^T$. Applying these two transformations H_1 and H_2, this results in the rectification transform

$$H_{rect} = H_1 \cdot H_2. \tag{2.18}$$

Having calculated H_{rect}, this transformation can be applied to the left image to rectify it. Then the right image is rotated the same way to recover the original geometry. In the last step, the right image is rotated with R^{-1}, where $(R|t)$ is the rigid body transformation between the left and right camera. This will lay the left and right image planes into a common plane. So in order to rectify stereo images, first the transformation H_{rect} has to be found. Then it gets applied to the left and right image and finally the right image is back rotated with R^{-1}.

The calculation of H_{rect} can be done the following way. It is constructed of three orthogonal unit vectors $\mathbf{e_1}$, $\mathbf{e_2}$ and $\mathbf{e_3}$. The goal of the transformation is to move the epipole e to infinity on the x-axis. So the new x-axis must have the direction

$$\mathbf{e_1} = \frac{\mathbf{t}}{\|\mathbf{t}\|} \tag{2.19}$$

where $\mathbf{t}$ is the translation vector between O and O'. The next constraint is, that the y-axis must be orthogonal to $\mathbf{e_1}$. So the normalized cross product of $\mathbf{e_1}$ and the optical center axis can be calculated using:

$$\mathbf{e_2} = \frac{1}{\sqrt{t_x^2 + t_y^2}}[-t_x,t_y,0]^T \tag{2.20}$$

Finally the z-axis is simply determined by $\mathbf{e_3} = \mathbf{e_1} \times \mathbf{e_2}$. Thus, the rectification transform can be defined as

$$H_{rect} = \begin{bmatrix} \mathbf{e_1^T} \\ \mathbf{e_2^T} \\ \mathbf{e_3^T} \end{bmatrix}. \tag{2.21}$$

Figure 2.8 demonstrates the results of a stereo calibration after rectification. First of all, both images get undistorted according to the distortion coefficients that are estimated during the calibration. Then the images get rectified. The epipolar lines are all parallel to the x-axis and are drawn as green lines. It can be seen that all corresponding points of both images are lying on the same horizontal epipolar line. Due to the undistortion and rectification, the images are not having a rectangular shape any more. That is why the images get cropped at the red lines, which are terminating the actual region of interest. After this processing step, the images will have a smaller size and are ready for a further correspondence search.

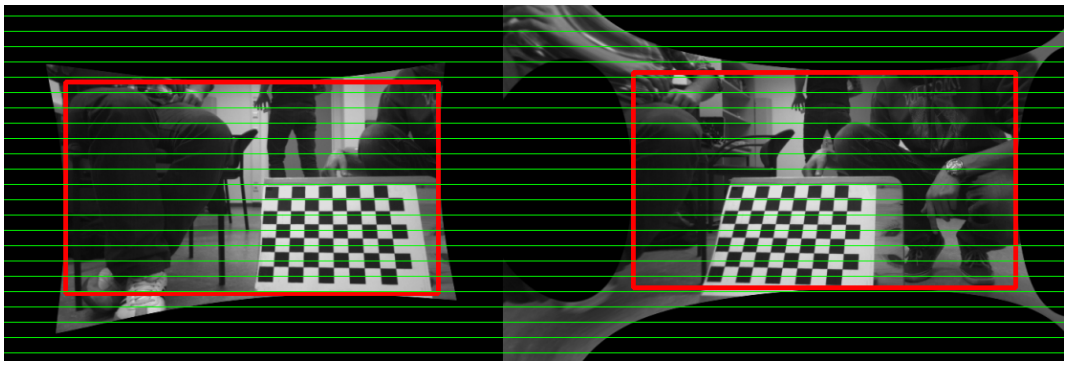

Figure 2.8: Undistorted and rectified images after stereo calibration

2.5 Summary of Foundations

At the end of this chapter, the main aspects of the preliminaries will be concluded. By using the pinhole camera model, it is possible to estimate how a point in the real 3D world is mapped to a 2D image taken by a camera. The calculation of this projection contains some intrinsic parameters of the camera (focal length and optical center point) that are stored in the camera matrix $\mathbf{K}$. Furthermore, it also contains the pose $(R|t)$ of the camera coordinate system relative to the world coordinate system, which is called external parameters. Doing a calibration of the camera with a plane known object like a checkerboard, the camera matrix $\mathbf{K}$ and the extrinsic parameters $(R|t)$ can be determined. In addition, the distortion of an image that is caused by the lens can be computed. This distortion can be removed after the calibration step. For controlling the robot based on camera data, it is important to know about the exact relation between the camera coordinate system and the robot motion center. This is estimated by performing the Hand-Eye-Calibration. Finally, a stereo calibration calculates the relative pose between a fixed camera pair. The epipolar constraint provides a relation between corresponding image points of two pictures, which is given by the fundamental matrix $\mathbf{F}$. This makes it possible to estimate the motion between images and to calculate epipolar

lines for efficient correspondence search. The rectification of stereo images is another technique for simplifying this search process.

So now we have the basics for describing positions and orientations in the world and how they relate to each other. Knowing how the real world projects to an image, the image data can be used for further calculations. By taking advantage of the multiple view geometry, the scene can be modelled by triangulating 3D points and localising the camera in it.

Chapter 3

Adaptive Visual Odometry

After learning the basic techniques for computer vision, this chapter will be about the actual approach that is implemented in this work. The main goal of this thesis is to find an algorithm that estimates the location of a robot. This should be done by using image sequences that are provided by a stereo camera which is mounted on the robot itself. The estimation of the egomotion will be carried out by visual odometry, which is further improved by an adaptive feature search. So this chapter will describe all the relevant steps of the VO algorithm and the control of the robot. This includes the following topics:

- Explaining what features are and why they are important. Then various feature detectors are illustrated and discussed.
- Pointing out, how to find feature correspondences between different images.
- Showing how the motion of the camera can be estimated from multiple images. This contains different VO methods, triangulation of 3D points, solving the PnP problem and using RANSAC for outlier rejection.
- Describing the approach of an adaptive feature selection and how it can improve the results.
- Presenting how the robot is controlled to find features and to get to the desired locations.

3.1 Feature Detection

In visual odometry, two or more consecutive pictures are compared to calculate the motion of the camera. In order to do this, it needs to be found out, how the positions of projected 3D points are changing across the different images. Therefore, it is required to find certain points in one image and check how they have moved in the other image. Due to the fact that a computer should be able to find these points in both pictures, they have to be unique and outstanding. This is why these points are often called features or

keypoints. Figure 3.1 illustrates what we call the “aperture problme”.

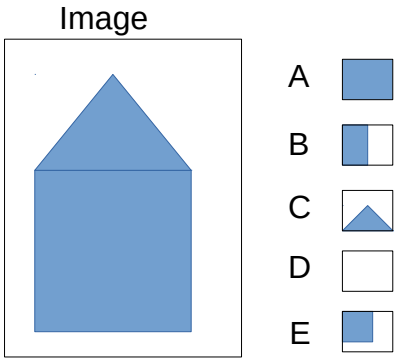

Figure 3.1: Image with different kinds of feature points

The image is showing a simple house, assuming the patches A, B, C, D and E were found in another image of the same scene. Now these cutouts need to be matched in the original image. The patches A and D contain a homogeneous area. It is impossible to say, where exactly these patches can be found in the image. For example, A could be at the top of the house, but also at the bottom; there are infinite possible locations. So this kind of feature is ambiguous to match, since it is not unique at all. Patch B shows an edge of the house. There are only a few positions at the right side of the house, where it might be located, but it is still not distinct. Only the patches C and E, that represent corners of the house, can be explicitly located in the image. They can be matched to an exact position. So corners might be a good kind of feature points for computer vision.

Today there are many approaches of finding robust features in images. There are mainly two different types of feature detectors: First there are corner detectors (like that one in the example above) and second, there are blob detectors [28]. Blobs are regions in an image that differ in properties from its adjacent regions. When selecting a certain feature detector for an application, there are several aspects that need to be taken into account. Important properties for feature selection are

- Robustness (invariance of rotation, translation, scale and illumination),
- Accuracy of feature point localisation,
- Computational cost of the algorithm.

The following chapter will present some different feature detectors and discuss their up- and downsides.

3.1.1 Harris Corner

One of the first feature detector approaches is the Harris Corner Detector [29]. It was introduced by C. Harris and M. Stephens and aims at finding corners in an image. The general idea is that, considering a grayscale image, the intensity of the pixels around a corner is changing dramatically. So in order to find a corner in a picture, the intensity $I(x,y)$ of an image point (x,y) is compared to the intensities of a patch of neighbouring pixels. $I(x+u,y+v)$ is the intensity of a pixel shifted with u in x-direction and v in y-direction from the original pixel in (x,y). In other words, one has to find the point (x,y) that maximizes the following function for all u and v of the considered patch:

$$\sum_{x,y}[I(x+u,y+v) - I(x,y)]^2 \tag{3.1}$$

In order to find out where the intensity is changing the most, the image point intensities are differentiated in both directions, applying the first order Taylor expansion. This leads to the Hessian matrix $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} I_x I_x & I_x I_y \\ I_y I_x & I_y I_y \end{pmatrix} \tag{3.2}$$

An assertion about the change of the intensity can be made after calculating the eigenvalues λ_1 and λ_2 of $\mathbf{M}$. If both λ_1 and λ_2 are small, the intensity is not changing at all, so the region seems to be flat. In case, one eigenvalue is much bigger than the other, the pixel intensity is changing in one direction but not in the other. That means that there is an edge at this point. Finally, if both eigenvalues λ_1 and λ_2 are large there is a corner, due to a change of intensity in both directions. So Harris introduced a score

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 = det(M) - k \cdot trace^2(M) \tag{3.3}$$

that represents the status of the eigenvalues of $\mathbf{M}$ and therefore the change of intensities [30]. Since R can be calculated by only estimating the determinant and trace of $\mathbf{M}$, there is no need of decomposing the actual eigenvalues. That saves some computation time which is a good property for a feature detector. In the end, the score R is calculated for every pixel of an image to find the corners. Each point, that gets a value for R which is higher than a certain threshold, can be determined as a corner.

A modification of the Harris corner detector is the "good features to track" algorithm by J. Shi and C. Tomasi [31] which is also widely used. The only difference is the definition of the score function R. They state

$$R = min(\lambda_1, \lambda_2). \tag{3.4}$$

This gives better results under certain circumstances for stable corner detection. Nevertheless, it requires an explicit eigenvalue decomposition of $\mathbf{M}$, which increases the computation time.

3.1.2 FAST

In addition to the Harris Corner detector, there are many other corner detectors. One notable feature detector is the "Features from Accelerated Segment Test" (FAST) that was introduced by E. Rosten in 2006 [32]. According to the name, the main advantage of this corner detector is its good speed in computing the corners. This chapter will describe how FAST works.

In order to determine if a pixel p is a corner, a so called segment test is performed. To do so, a circle around the pixel p is taken into account. This circle with the radius 3 consists of 16 pixels (see Figure 3.2). These pixels are enumerated clockwise to identify them better. The radius can also be chosen differently, but in most cases it will be 3 pixels.

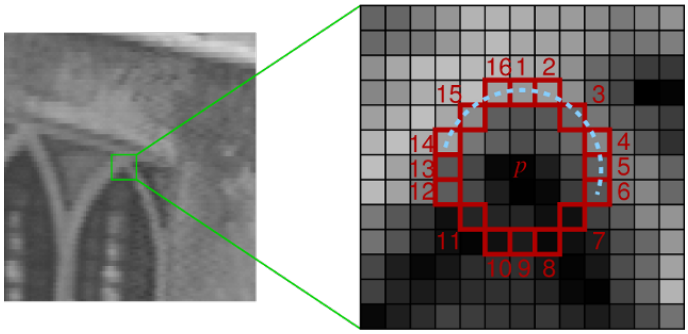

Figure 3.2: Circle around the corner candidate p [32]

For validating that p is a corner, the surrounding 16 pixels are checked in the following way: If there are n pixels in the circle that are all brighter than the intensity I_p of p plus a certain threshold t or all darker than $I_p - t$, then p is considered as a corner. So all that needs to be done, is to compare the intensities of the surrounding pixels of p. The number n of pixels that need to be brighter or darker than $I_p \pm t$ is often set to 12, which means that three quarters of the circle pixels have a much different intensity compared to p. Anyway, n can also be chosen in another way, depending on the circumstances and how many corners should be found.

Thus in total, there are 16 comparisons made per pixel. Due to the fact that most pixels will not be a corner, a previous high speed test is performed to exclude a large number of non-corners. For this test only four pixels of the circle are examined. These are the pixels with number 1, 5, 9 and 13 (the pixels at 0°, 90°, 180° and 270°), so all four directions are covered. These four pixels are checked like in the normal intensity test. If three out of the four pixels are brighter than $I_p + t$ or darker than $I_p - t$, the pixel p passes the high speed test and the full segment test will be performed. This

procedure makes this corner detector very fast, so it saves some precious computation time compared to the Harris corner.

The downside of this algorithm is that it generates a set of many corners, which cannot be ranked. Some corners might be better than others, however, there is no score for the corners, like we have seen at the Harris detector. This will lead to the fact, that there are going to be multiple features detected adjacent to one another. To tackle this problem, a non-maximal suppression can be used. So a score function V is introduced and will be computed for every detected corner. Rosten names three different possible definitions for that response function V:

1. The maximum value of n for which p is still a corner.
2. The maximum value of t for which p is still a corner.
3. The sum of the absolute difference of intensities between the pixels in the circle and the center pixel.

Having such a score function, it is possible to compare the quality of the corners and only pick those with the best score to gain a good set of corners. Another simple approach is to identify the corners with the two stage FAST segment test and then applying the Harris score function to find the most distinguished set of corners. Rosten also introduces a machine learning extension to make the corner detector even better.

So all together, the FAST corner detector is an algorithm that is capable of detecting feature points really fast, which is very important for a real time application. Setting the threshold values t and n it can be made quite robust. Introducing a score function and non-maximal suppression, the corners can be ranked by quality as well, which makes this feature detector very suitable for our application.

3.1.3 Blob Detectors

The Harris and FAST algorithms aim at finding corners as prominent feature points. Figure 3.1 showed that corners work really well for identifying them in a scene. This work is about a moving robot that is taking pictures while moving. So even if the scene stays the same, the viewpoint changes. We still want to detect and match image points from two different views, even if the image of a scene has changed. This is why the feature points need to be invariant to translations, rotations, scale and also illumination. Obviously, corners are invariant to translations as well as rotations. If a corner is translated or twisted around, it still remains a corner. Indeed changing the scale makes a difference. Figure 3.3 illustrates this problem. On the left there is clearly a corner.

When zooming into that corner, it can be seen, that no corner part remains but only more or less straight lines.

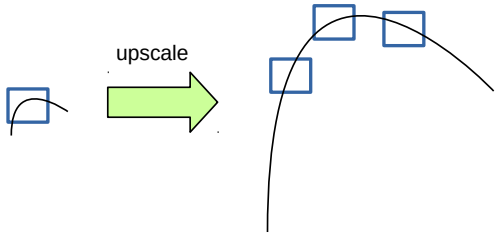

Figure 3.3: Example of scale invariance

Hence corner detectors are not invariant to scale. That can be a problem, if the camera is moving towards an object very quickly, so it becomes larger and larger. This might cause that corners cannot be tracked any more after the camera has moved.

This is where blob detectors come into play. In 2004 D. Lowe came up with a new algorithm called Scale Invariant Feature Transform (SIFT)[33]. Like the name denotes, this feature detector is supposed to be invariant to scale. It works as follows:

First it is important to note, that for detecting a small corner, a small observing window is feasible. That can be seen in the figure above. In order to detect a large corner, it is necessary to use a large window as well. This can be achieved with scale space filtering. To do so, the Laplacian of Gaussian (LoG) is applied to the image. First the general Gaussian filter function $f(x, y)$ for a 2D image is denoted by:

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.5}$$

Then the Laplacian operator is applied to receive the LoG filter $g(x, y)$:

$$g(x,y) = \Delta\, f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} = -\frac{1}{\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}} \left(1 - \frac{x^2+y^2}{2\sigma^2}\right) \tag{3.6}$$

This LoG filter works as a blob detector. Blobs are regions of an image, where all points in this blob do not differ much from each other (e.g. same brightness or colour). Applying it to an image, it finds blobs with a certain size. This size is determined by the σ of the function $g(x, y)$. So varying this σ for multiple LoGs, the algorithm will find blobs of different sizes relating to the σ. So in short, σ acts as a scaling factor. For the example in figure 3.3 this means, that both the small and the large corner can be pointed out by changing the σ of the filter function. So it is possible to find the local maxima across the scale and space. After operating the LoG using many different σ values, this yields a list of (x,y,σ) extrema points, which means that there is a potential keypoint at (x,y)

at σ scale.

Since the calculation of the LoG for many different σ is very time consuming, the SIFT algorithm takes advantage of the Difference of Gaussian (DoG), which is an approximation of LoG. It is calculated by first filtering the image with the normal Gauss filter $f(x, y)$. This is done for five different scaling factors σ. Such a set is called an octave. Finally the DoG is obtained by the subtraction of two Gaussian filtered images with different scaling factors σ. Lowe recommends to choose the factor k between different scaling factors σ as $k = \sqrt{2}$. Figure 3.4 points out that principle.

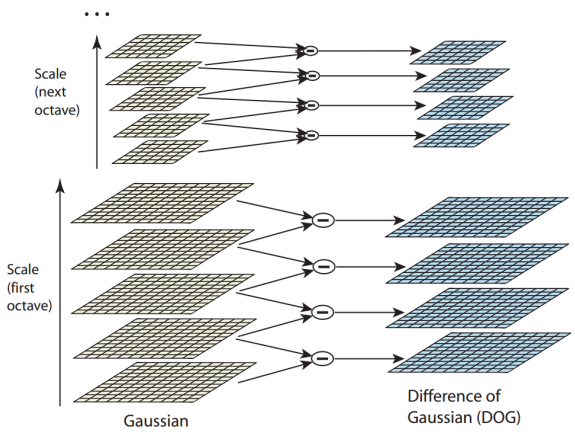

Figure 3.4: Calculation of DoG [33]

After calculating the DoGs, potential keypoints can be searched in the image. First, the value of a pixel is compared to its neighbouring pixels. If it is a local extrema, it is then compared to the surrounding pixels in the previous and next scale. If it is still an extrema, it can be established as a keypoint. So a scale is found, which fits the best for this feature point. Furthermore, the DoG value of the keypoint should be higher than a certain threshold to eliminate weak feature points.

In conclusion, the SIFT finds feature points (blobs) in images that are both invariant to scale and rotation, which makes it more robust than the normal corner detectors. The problem with SIFT is, that the calculation of all the Difference of Gaussians for many different scaling factors σ is very time consuming. That is why various improvements to SIFT were presented in recent years. One of the most popular ones is the approach "Speeded Up Robust Features" (SURF) by H. Bay et al [34] in 2006. It makes some modifications of the SIFT algorithm. The most important change is the utilisation of a box filter, instead of the Difference of Gaussian filter. The authors claim that SURF is about three times faster than SIFT, while it is providing the same good performance and robustness of feature detection.

For this work, SIFT and SURF feature detectors were tested. Both techniques work quite good and achieve robust results, since they are invariant to scale and rotations. However, even if SURF is a speeded up version for scale invariant feature detection, the calculation time was much longer than for the Harris Corner or FAST (about five times slower). Since the localisation task of the robot is a real time problem, a short computation period of the algorithm is essential for a good performance. In our application, image points from the left and right camera are compared. The baseline between both cameras is not very large in relation to the distances in the observed scene. Additionally, the cameras have the same orientation. Due to these facts, there will not be a large change in scale or rotation between the left and right camera image. Thus, the main advantage of the blob detectors SIFT and SURF, namely the invariance to scale, is not really required in our application. So in the end, a modified version of the FAST feature detector was chosen for our algorithm, because of its good computation time.

3.2 Feature Correspondence

The previous section was all about feature detection. There are many different techniques to extract feature points from images. Every algorithm has its own strengths and weaknesses. The reason why we were looking for unique and outstanding feature points is that we want to find those points in different images which were taken from diverse positions. Having found these keypoints, they need to be matched to each other in two or more images. There are mainly two different approaches for solving this correspondence problem: Feature tracking and feature matching.

Feature tracking means that a keypoint is pursued over multiple images. The first step to do so, is detecting feature points in the initial image, like described in the previous section. Then the assumption is made, that these feature points are not moving much and that they will look the same in the next image. Some further calculations make it possible to track a feature that was found once, from one image to the next. One of the most important approaches of feature tracking is the Kanade-Lucas-Tracker. It will be explained in chapter 3.2.4. According to the assumption, that the two images do not change a lot when tracking features, this technique is especially suitable for consecutively taken images from one camera. Since the movement of the camera is not very large between two images, this technique works pretty well in this case.

Nevertheless, finding point correspondences between two successive images is only one part of this work. It is also necessary to find correspondences between the left and right image points of the stereo camera. On account of the fact, that the distance between the two cameras is not that small, feature points that are close to the stereo camera might appear at quite different positions on the left and right image. In order to find the

points that correspond, descriptor based feature matching is performed. So this method is going to be used for matching the stereo images. It basically contains the following steps:

First image features are detected in the left and right image individually. Then a certain area around this feature point is extracted to describe the keypoint. Therefore, this area of pixels is called feature descriptor. There are many different approaches of defining these descriptors which will be briefly explained in the following section. After having detected two sets of feature points with their descriptors, a correspondence search follows based on these descriptors. Those which give the best match are establishing a point correspondence.

3.2.1 Feature Descriptors

As announced before, it is essential for feature matching to have descriptors of the detected feature points. The easiest descriptor would be a simple square around the feature point. This would not work well, since such a square is neither invariant to rotation nor to scale. So if the feature point is a bit rotated in the other image, this technique would not find a good match. Thus, it is important to choose a suitable feature descriptor.

The same requirements that are needed for a good feature detector are required for descriptors as well. So the following properties are desired:

- Invariance to transformations (translation, rotation),
- Invariance to scale,
- Invariance to illumination and blur,
- Low memory requirements.

Usually not all of these points can be fulfilled at the same time. Often there is a trade off between a robust descriptor (invariances) and a low memory demand (fast computation time).

For example the SIFT algorithm has a well defined descriptor, where a 16x16 pixel neighbourhood around the feature point is taken into account. This area is divided into 16 sub blocks with size 4x4. Then the orientation of these sub blocks is computed by calculating their gradient. It is saved in a histogram with eight bin values as the descriptor. So in the end this yields a vector of $16 \cdot 8 = 128$ bin values (usually floating point numbers) as the SIFT descriptor. This method is very robust and therefore performs well for matching. On the other hand 128 values for each feature point is quite a lot to compare, so the matching process will take a relatively long time.

Due to the fact, that the SIFT and SURF feature detectors were too slow for our application, another method for feature descriptors is going to be used. Instead of storing a vector of many floating point numbers (with relative large memory requirements), in this approach the descriptor vector should only consist of binary values. This reduces the demand of memory greatly and the matching process will be much faster as well. This kind of descriptors are called binary feature descriptors.

The following paragraph illustrates how such a binary descriptor works, using the example of "Binary Robust Independent Elementary Features" (BRIEF) [35], which is a popular binary descriptor.
In the first step, a patch of size $S \times S$ around the feature point is smoothed by a Gauss filter (see chapter 3.1.3), which reduces the image noise of single pixels. This is important, because in the next step, pixel intensities of certain points are compared to each other. To do this, a binary test τ in the image patch p is introduced:

$$\tau(p; \mathbf{x_1}, \mathbf{x_2}) = \begin{cases} 1 & if \quad I(\mathbf{x_1}) < I(\mathbf{x_2}) \\ 0 & if \quad I(\mathbf{x_1}) > I(\mathbf{x_2}) \end{cases} \tag{3.7}$$

In this equation $I(\mathbf{x_1})$ and $I(\mathbf{x_2})$ are the intensities of the points $\mathbf{x_1}$ and $\mathbf{x_2}$. In order to get a robust descriptor, a number of n_d of these tests are performed for n_d different point pairs $\mathbf{x_1}$ and $\mathbf{x_2}$. The results of these tests are stored in a bit string of size n_d, which forms the BRIEF descriptor. It is computed by the following formula:

$$f_{n_d}(p) = \sum_{i=0}^{n_d} 2^{i-1} \cdot \tau(p; \mathbf{x_{1,i}}, \mathbf{x_{2,i}}) \tag{3.8}$$

Usually $n_d = 256$ is chosen to get a solid descriptor. This means, that the memory demand amounts only 32 byte, which is much less than the SIFT descriptor mentioned above. The point pairs $\mathbf{x_1}$ and $\mathbf{x_2}$ can be chosen in five different ways, which are described in the BRIEF paper. Their performance differs for various applications. For example, one way of choosing the point pairs is taking normally distributed points with a certain variance.

Summing up, the BRIEF descriptor is just a simple string of bits. It randomly chooses point pairs around the feature point and compares their intensities. The result is stored in a bit string. This is a very simple and memory saving technique of describing a feature. The paper says, that this descriptor is invariant to scale and also to illumination and achieves good feature recognition rates. These properties are very useful for the real time application of this work. However, the BRIEF descriptor still has problems with rotations that are in plane. The following section is showing a way to counter this problem.

3.2.2 ORB

In the last two sections we have seen some different feature detection and descriptor approaches. All of them have their up- and downsides. For the localisation task of this thesis, an algorithm is needed that is both fast and robust. The idea is, to take a fast algorithm and modify it in a way, so that it is solid enough to satisfy our requirements. After various tests with different feature detectors and descriptors, one method was verified as the best performing technique for our application. It is called "Oriented FAST and Rotated BRIEF" (ORB) [36] by E. Rublee et al. It was published in 2011, so it is quite a modern solution. As the name states, it is an extended version of the FAST feature detector and the BRIEF descriptor. This section will explain the modifications that were made.

The first part will be about oriented FAST. As we have seen in chapter 3.1.2, the FAST detector is a quick method to extract corner points in an image (with low computation time). There are still some critical points with that: The corners are only slightly rotation invariant, they are not scale invariant and there is no built-in quality mark to find the "best" corners. All these three problems are solved in the ORB approach.

Rotation invariance:
In order to make a feature point invariant to rotations it is essential to find the orientation of a feature (like we have seen in SIFT). To do so, the method of the intensity centroid is used according to their paper. It is assumed that the intensity of the area around the corner differs from the center. So foremost, a patch around the corner center O is taken. Then the moments for all points (x,y) of this patch can be calculated with the following equation

$$m_{pq} = \sum_{x,y} x^p \cdot y^q \cdot I(x,y) \tag{3.9}$$

where x and y are the coordinate points of the pixels in the patch around the corner center and $I(x,y)$ is the intensity of this point. The intensity centroid of the patch around the corner center is then defined by

$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}). \tag{3.10}$$

The orientation of the corner is simply given by the vector from the corner center O to the centroid of the patch C, which is $\overrightarrow{OC}$. The orientation angle θ of the corner can be simply computed by $\theta = atan2(m_{10},m_{01})$. So with this modification, it is possible to get FAST corners that are invariant to rotations.

Scale invariance:
One of the biggest problem with corner detectors is, that they are not invariant to scale. So FAST also does not produce multi-scale features on its own. To tackle that problem

a scale pyramid of the image is employed for ORB. Figure 3.5 points out how such an image pyramid is generated.

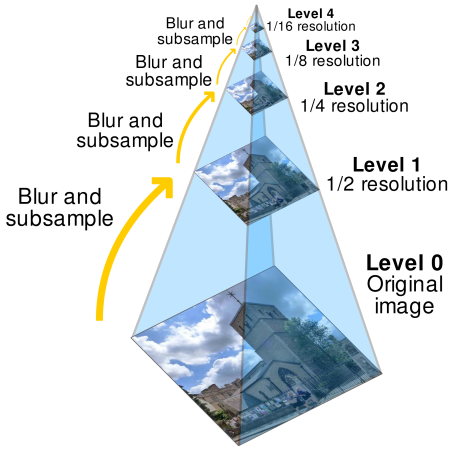

Figure 3.5: Image pyramid [37]

This method generates multiple versions of an image at different scales. Usually, the resolution is bisected for each level of the pyramid. After having a certain number of different scale images, the FAST detector will run over all levels of the pyramid, extracting feature points in a multi scale space. This makes the feature detector more invariant to scaling.

Quality score for corners:
Finally the FAST corner detector is good at detecting corners, but it gives no indicator of how good a corner is. This leads to the fact that many corners will be detected of which only a few are really good and unique. In his paper E. Rosten already gives some possible ways to find a score function to rank the corners. However the authors of ORB prefer the Harris score function to examine the best corners. So after finding a corner with the FAST detector, the Hessian matrix $\mathbf{M}$ is computed at this corner point (see chapter 3.1.1). Then the Harris score function

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 = det(\mathbf{M}) - k \cdot trace^2(\mathbf{M}) \tag{3.11}$$

is applied to the corner, which gives a quality mark for this feature. Having such a score for each feature, only the best corner points can be filtered out.

The second part of this section illustrates the idea behind the rotated BRIEF (rBRIEF). Chapter 3.2.1 already showed how the BRIEF descriptor works. It is a binary descriptor that executes simple binary intensity tests. That makes it pretty fast and it also has a relative low demand on memory. The biggest drawback of this descriptor is that

it is quite sensitive to rotations. So after creating a feature point that is invariant to rotations, it is necessary to adapt this invariance for the descriptor as well. This is why the authors of ORB introduced rBRIEF. The basic idea behind this method is actually pretty simple: The patch for the test points around the feature center that is used to generate the descriptor, gets steered into the direction of the detected oFAST corner. So when having a set of n binary tests at the locations (x_i, y_i), the $2 \times n$ matrix $\mathbf{S}$ can be defined:

$$\mathbf{S} = \begin{pmatrix} x_1 & \dots & x_n \\ y_1 & \dots & y_n \end{pmatrix} \tag{3.12}$$

Then the corner orientation θ that was previously computed by the intensity centroid method of oFAST is used to calculate the corresponding rotation matrix $\mathbf{R}_\theta$. Afterwards the rotated set of test locations $\mathbf{S}_\theta$ can be estimated as:

$$\mathbf{S}_\theta = \mathbf{R}_\theta \cdot \mathbf{S} \tag{3.13}$$

With this oriented set $\mathbf{S}_\theta$ the BRIEF descriptor is steered into the direction of the corner point. Thus we have a feature detector and descriptor that is invariant to rotations.

All together the three extensions to the FAST feature detector make it much more robust for many applications. The biggest weakness of the BRIEF descriptor is eliminated by rotating it into the direction of the corner. Since these modifications only cost a little bit more computation time, it is beneficial to apply them. So in the end the oriented FAST and rotated BRIEF (ORB) is still much faster than SIFT or SURF, while having similar robust performance results, according to the authors' paper.

3.2.3 Matching Procedure

This section is about the actual feature matching of two images (e.g. the left and right image of a stereo pair). After having found feature points in both images, the feature descriptors of these points are extracted. These feature descriptors are usually vectors of numbers (e.g. 128 bin values for SIFT) or a simple vector or string of binary values (when using binary descriptors like BRIEF). In order to find the best match of a feature point in the first image, its descriptor is compared to all feature descriptors of the second image. So basically, the values of the descriptor vectors are compared. The result of this calculation is called dissimilarity score. For this computation mainly two different approaches are existing: the L^1- and L^2-distance [38].

The L^1-distance sums up the absolute difference of the two descriptor entries. Thus, it is also often referred to the sum of absolute difference (SAD). The L^2-distance is calculated by squaring this difference of descriptor values. Hence, it is denoted as the sum of squared differences (SSD). This squaring of the error between the two descriptors leads

to the effect, that small differences are not really important, while big errors are heavily weighted. This can be beneficial in many cases, but it also increases the influence of noise. So depending on the application and circumstances, either the L^1- or L^2-distance should be chosen for descriptor matching.

In both cases it is obviously that the smaller the calculated distance between two descriptors is, the better is the match. So after estimating the distances between one descriptor of the first image with all descriptors of the second image, the two descriptors with the lowest distance are determining a point correspondence. This procedure can be iterated until all detected feature points are connected with their best match in the other image. Due to the fact, that not all of these correspondences will be true matches, it is possible to only choose matches with a distance below a certain threshold. This is how some mismatches can be removed.

While the L^1- and L^2-distance works well for normal descriptors that store numbers in their descriptors like SIFT and SURF, binary descriptors are a special case [39]. Since only binary values are stored in the descriptors, there is no point in calculating absolute or squared values for distances. It would still work, but it is a waste of computational resources. For these binary descriptors, the so called "Hamming" distance is used. It is calculated by a simple Exclusive-Or (XOR) operation between the two descriptor vectors. Therefore, the matching process for binary descriptors is also much faster than for ordinary descriptors.

Selecting only matches with a distance that is below a threshold value makes it easy to remove most of the bad matches. However, there will be still some wrong matches that remain. Two popular ways of doing so are the cross check process and the ratio test which will be explained in the following.

Cross Check:
The cross check is a simple method that makes the matching process more robust. In the normal case, a descriptor of the first image is compared to all descriptors of the second image. The descriptors with the lowest distance are establishing a valid match. When performing the cross check, the same method is applied, but in the other direction as well. So in addition to the first comparison, all descriptors from the second image are also compared to the descriptors from the first image. Only if two descriptors are marked as the best match in both of these tests, this match is determined as valid. This requires some additional computation time, but results in less false matches.

Ratio Test:
The other technique of improving the matching results was introduced by Lowe [33]. It is called ratio test and deals with the following problem: In some scenes, there will be many good matches for one descriptor. Figure 3.6 illustrates this problem of very similar

descriptors.

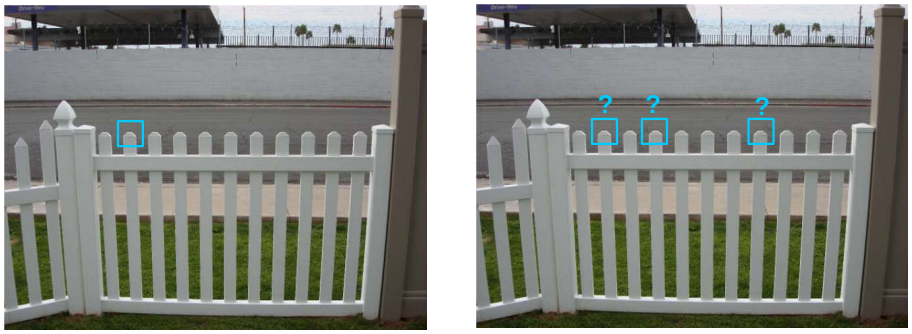

Figure 3.6: Similarity Issue

When trying to find the best match for the descriptor in the left image (blue square), there are many possible descriptors in the right image that are suitable. In other words, the dissimilarity score or distance between the best and second best match is quite the same. That makes it hard to match the descriptors in a robust way and it would be better to choose another feature point that is more unique. In order to discard such feature points/descriptors which are not unique enough, the ratio test is performed. For that purpose, the dissimilarity score of the best and second best match is calculated. Then both values are divided, which results in the ratio of these two scores. If it is close to one, this means that the descriptor is not unique in this scene and should not be utilised. Hence, all feature points that achieve a ratio above a certain threshold value get discarded. All other descriptors can be clearly assigned to each other and thus are marked as a good match.

In conclusion feature matching is performed by comparing the feature descriptors of two images. Defining a certain dissimilarity score or distance, the best match of two feature points can be found. The cross check and ratio test are two techniques that make the matching process more robust by removing uncertain point correspondences.

3.2.4 Feature Tracking

Besides the feature matching, which is performed to find point correspondences between the left and right image of the stereo camera, it is also important to find correspondences of feature points between consecutively taken images in a video stream. Since these images are taken at a high frame rate (e.g. 10 images per second), the scene will not change a lot between two images, even if the camera is moving. Since there is not a big change in rotation, scale and illumination, no robust feature matching algorithm is needed, but the feature points can be tracked over multiple images. The Lucas-Kanade-Tracker [40]

was already presented in 1981 and is still a widely used and important feature tracking algorithm. This chapter will illustrate the basic idea of this concept.

At the beginning, there are three key assumptions made for the properties of two consecutively taken images:

1. There is only a small motion, so points do not move very far.
2. The brightness is constant, so points look the same in each frame (same intensity).
3. There is a spatial coherence, so points that are near to each other make the same movement.

When tracking feature points over multiple images of a video stream, all these assumptions hold true in most cases. So the next step is to consider how a feature point is moving from one image to the next.

First the image point is at the position (x,y). Then it moves a little bit with (u,v), so in the next frame it will have the position $(x+u,y+v)$. This yields the brightness constancy equation

$$I(x,y,t) = I(x+u,y+v,t+1) \tag{3.14}$$

where $I(x,y,t)$ is the intensity of point (x,y) at time t. Now the Taylor expansion is applied to the right side to linearise. After rearranging, the brightness constancy equation is inserted:

$$I(x+u,y+v,t+1) \approx I(x,y,t) + I_x u + I_y v + I_t \tag{3.15}$$

$$\Leftrightarrow I(x+u,y+v,t+1) - I(x,y,t) \approx I_x u + I_y v + I_t \tag{3.16}$$

$$\Rightarrow 0 \approx I_x u + I_y v + I_t \tag{3.17}$$

In order to track a point, we want to estimate the movement (u,v) of this point by taking advantage of equation 3.17. The problem is that there is only one equation, but two unknowns u and v. This is where the third key assumptions becomes important. All neighbouring pixels are assumed to have the same movement (u,v). For example taking a 3×3 neighbourhood patch of a feature point into account, this yields the following equation where p_1 ... p_9 are these surrounding pixels:

$$\begin{pmatrix} I_x(p_1) & I_y(p_1) \\ \dots & \dots \\ I_x(p_9) & I_y(p_9) \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix} = - \begin{pmatrix} I_t(p_1) \\ \dots \\ I_t(p_9) \end{pmatrix} \tag{3.18}$$

This overdetermined system can then be used to find the best u and v to satisfy the equation. Therefore, the new location $(x+u,y+v)$ of a feature point can be estimated with this method. That makes it possible to track a certain point over multiple images.

For our application this approach works really well, so this method is used for finding point correspondences between the images taken successively.

3.3 Pose Estimation

The major goal of this thesis is to implement an algorithm, that localises a robot by using visual data from two cameras. After learning how to detect special feature points, matching them in other images and knowing their relations in multiple image frames, the next step is to recover the rigid body motion $(R|t)$ of the robot between two consecutively taken images. This process of estimating the motion of a camera based on a video is called Visual Odometry (VO). Davide Scaramuzza has published a comprehensive overview about visual odometry in general and some different implementation methods [6]. There are mainly three approaches according to his paper. There is the 2D-2D, the 3D-3D and finally the 3D-2D technique that can be utilised. All these methods will be briefly explained and discussed.

The oldest and easiest way is the monocular 2D-2D approach. This name is chosen, because in this case, 2D image features are compared with other 2D image features. So the general algorithm works like the following: First an image is taken and feature points are detected. Then the next image is taken (with the same camera) and features are detected again. So this will provide two sets of 2D image features which can be matched. After that matching process, the fundamental matrix of the two images can be calculated (see chapter 2.4). That matrix is then decomposed to the rigid body motion $(R|t)$ between the two consecutively taken images. This whole method is pretty straight forward and therefore quite robust. Another advantage is that only one camera is required for taking the pictures. However, a significant disadvantage that comes with that monocular technique, is the so called scale ambiguity. That means that even if the calculated rotation will be correct, the estimated translation is only true up to an unknown scaling factor. That comes, because there is no reference for the motion of the camera. For example it can be calculated, that the camera has moved five units in the x-direction, but the relation of one unit and one meter is unknown. This is a big problem, since the robot should be localized in a room, knowing its absolute position. There are possibilities to solve this problem (e.g. with a reference run), though in this work another technique is used.

The next approach avoids this scale ambiguity and is denoted 3D-3D. As the name indicates, two sets of 3D points are compared to recover the pose of the robot. In order to receive 3D points out of 2D images, a stereo camera is used. Again, feature points are detected and matched between the left and right camera. After the stereo calibration, the exact pose between the two cameras is known. So with that information, triangulation

(will be further explained in chapter 3.3.1) can be performed to get the 3D coordinates of the detected feature points. Due to the fact, that we have a scaled knowledge about the pose between the two cameras, there is no scale ambiguity any more. This means that after calculating the 3D points, their position is determined in metric units. After this calculation, the two cameras take pictures again at the next timestep. Now the 3D points of the next images are getting triangulated again, which yields another set of 3D points. By comparing these two sets, it is possible to recover the motion that the stereo camera has made. By using two cameras, it is possible to estimate an absolute position in meters, which makes this approach suitable for many VO applications. In [9] a novel implementation of this practice is illustrated. However, in reality there is always some image noise and other uncertainties. That is why the triangulation step will not be totally accurate and introduces some error. Since the triangulation is performed two times for pose recovery, this error is added twice as well.

The last visual odometry algorithm, that will be presented here, is the 3D-2D method. In their paper [7] Nister et al. compare the different VO approaches and get the conclusion that the 3D-2D technique is the best and most accurate way of estimating the location of a camera. Similar to the 3D-3D method, 3D points get triangulated by a stereo camera in the first step. In the second timestep, only the image of one camera (e.g. the left one) is taken into account. Feature points are detected and matched to the previously triangulated 3D keypoints. Then the rigid body motion is calculated by comparing the 3D and 2D point sets. This method is referred to as the perspective n point (PnP) problem. The advantage of the 3D-2D procedure is that the triangulation error is only introduced once, while the results are still scaled, due to the usage of a calibrated stereo camera.

So this approach seems to be the best way of calculating the robots position. Therefore, this 3D-2D algorithm is chosen for the implementation of this thesis. The next two chapters will illustrate how the triangulation and pose recovery work (PnP).

3.3.1 Triangulation

This section will describe how the 3D world coordinates are calculated from the 2D image feature points of the left and right image. Under optimal conditions, this is a trivial problem. As mentioned in chapter 2.4, the only thing to do would be calculating the intersection point of the two lines that are going through the camera centres and the corresponding image points (see Figure 2.5). However, due to the discrete nature of images and some noise which always occurs at real cameras, these two lines will not intersect. Therefore, there are various approaches described in [41] for finding the optimal 3D coordinates. These methods comprise for example depth from disparity, linear triangulation, the mid point method and some iterative extensions. We tested some dif-

ferent techniques and came to the result, that the linear triangulation works best for our application of indoor localisation. So a derivation of this approach will be presented here.

A 3D world point $\mathbf{X}$ is projected to a 2D image point $\mathbf{x}$ by the following equation:

$$\mathbf{x} = \mathbf{K} \cdot (\mathbf{R}|\mathbf{t}) \cdot \mathbf{X} = \mathbf{P} \cdot \mathbf{X} \tag{3.19}$$

Using homogeneous coordinates $\mathbf{x} = w(u,v,1)^T$, where (u, v) are the image point coordinates and w is the scale factor. Then 3.19 yields the following equations, where $\mathbf{p}_i^T$ is the i-th row of the projection matrix $\mathbf{P}$ (which is known from stereo calibration):

$$uw = \mathbf{p}_1^T \mathbf{X} \tag{3.20}$$
$$vw = \mathbf{p}_2^T \mathbf{X} \tag{3.21}$$
$$w = \mathbf{p}_3^T \mathbf{X} \tag{3.22}$$

Applying the third to the first and second equation, we receive the following system:

$$u \cdot \mathbf{p}_3^T \mathbf{X} = \mathbf{p}_1^T \mathbf{X} \quad \Leftrightarrow \quad (u \cdot \mathbf{p}_3^T - \mathbf{p}_1^T) \cdot \mathbf{X} = 0 \tag{3.23}$$
$$v \cdot \mathbf{p}_3^T \mathbf{X} = \mathbf{p}_2^T \mathbf{X} \quad \Leftrightarrow \quad (v \cdot \mathbf{p}_3^T - \mathbf{p}_2^T) \cdot \mathbf{X} = 0 \tag{3.24}$$

Considering two views, the left image provides the point coordinates (u,v) and projection matrix $\mathbf{P}$, while the right image has the corresponding point (u',v') and matrix $\mathbf{P}'$. This provides the equation

$$\mathbf{A}\mathbf{X} = 0 \quad \Rightarrow \quad \begin{pmatrix} u \cdot \mathbf{p}_3^T - \mathbf{p}_1^T \\ v \cdot \mathbf{p}_3^T - \mathbf{p}_2^T \\ u' \cdot \mathbf{p}'^T_3 - \mathbf{p}'^T_1 \\ v' \cdot \mathbf{p}'^T_3 - \mathbf{p}'^T_2 \end{pmatrix} \cdot \mathbf{X} = 0 \tag{3.25}$$

that needs to be solved for the 3D point $\mathbf{X}$, where $\mathbf{A}$ is a 4×4 matrix. This can be done by performing the singular value decomposition (SVD) of $\mathbf{A}$. This technique will find a $\mathbf{X}$ that minimizes $\|\mathbf{A} \cdot \mathbf{X}\|$.

Figure 3.7 shows an image taken by our robot, in which the linear triangulation method is applied. Feature points are drawn into the scene where the colour of each point indicates the depth of the 3D point. Red means that the point is close to the robot, whereas yellow represents a point that is far away.

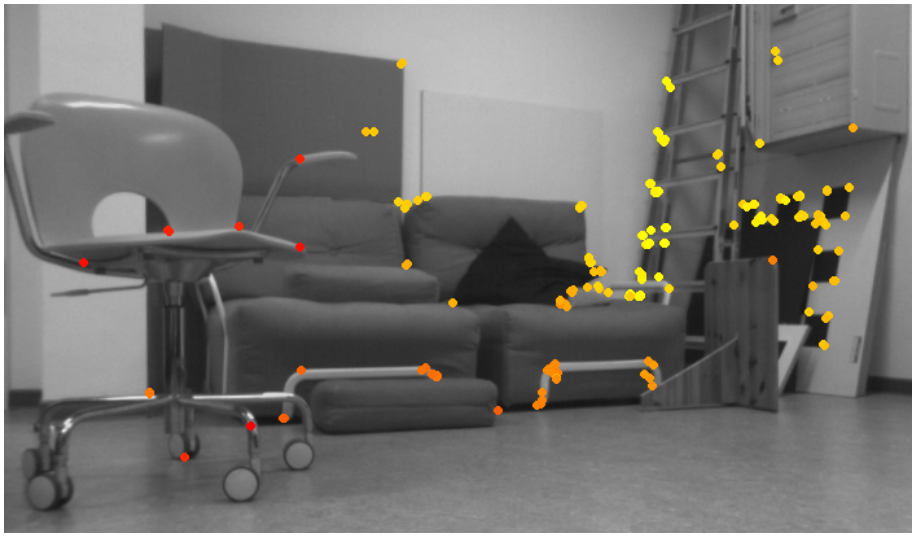

Figure 3.7: Scene with triangulated feature points

3.3.2 PnP Problem

After triangulating 3D points, the next step of the visual odometry algorithm is to calculate the rigid body motion of the robot $(R|t)$ between two consecutively taken image frames. For clarification, the images of the left and right camera at timestep k are denoted as $I_{l,k}$ and $I_{r,k}$. The next picture taken by the left camera at timestep $k+1$ is going to be represented as $I_{l,k+1}$. The idea is to first triangulate the feature points of $I_{l,k}$ and $I_{r,k}$. In the next step these features are tracked in $I_{l,k+1}$. So there is one 3D feature point set $\mathcal{F}_{3D,k}$ and one 2D set $\mathcal{F}_{2D,k+1}$. Then the 3D points are back projected according to equation 3.19 in order to receive 2D points again. In the last step, the $(R|t)$ of 3.19 is found which minimizes the reprojection error

$$\min_{(R|t)} \sum \left\| \mathcal{F}_{2D,k+1} - K \cdot (R|t) \cdot \mathcal{F}_{3D,k} \right\|^2 . \tag{3.26}$$

This approach is also described in [42]. Having N points in the 3D and 2D feature set, the estimation of the pose is called Perspective-N-Point problem (PNP). The reprojected point (u,v) is calculated from the 3D point (x,y,z) by the following equation [43]:

$$u = \frac{P_{11}x + P_{12}y + P_{13}z + P_{14}}{P_{31}x + P_{32}y + P_{33}z + P_{34}} \tag{3.27}$$

$$v = \frac{P_{21}x + P_{22}y + P_{23}z + P_{24}}{P_{31}x + P_{32}y + P_{33}z + P_{34}} \tag{3.28}$$

where the projection matrix P is defined by $P = K \cdot (R|t)$. It can be seen that each corresponding feature point supplies two equations. The optimisation 3.26 has got six degrees of freedom, due to three rotation angles and three translation directories. So

to solve this problem, there are at least three corresponding points required in the 3D and 2D feature sets. The minimal solution is therefore called the P3P method. This approach takes advantage of the cosine rule and also some other elimination steps which will not be further illustrated in this thesis. [44] describes how it works in detail.

There are many other PNP methods as well, that make use of more than three feature correspondences (e.g. the P5P approach). Anyway, we found out in our tests that the P3P algorithm is most suitable for our application, due to its low computation time.

In summary, the pose estimation is done by reprojecting previously triangulated points from the left and right camera and find the best $(R|t)$ that fits for the feature points which are detected in the left image one timestep later. Thus, in the second step, the right image is not used at all, although it might be better to use any image information that is provided. That is why the pose estimation was also performed for the features of the right image. Comparing this with the output of the left image, it did not really improve the total results. In order to save that additional computation time for the right image, the P3P algorithm is only performed for the left image frame.

3.3.3 RANSAC

When calculating the motion of the camera using a PNP algorithm, 2D image points are matched with 3D world points. Due to image noise, triangulation errors and other measurement uncertainties, it is impossible to find a transformation $(R|t)$ that perfectly fits all points in the set. In fact, a transformation should be found, which minimizes the reprojection error for all points. Since the algorithm that matches left and right image features, as well as the keypoints from two consecutively taken images, will fail sometimes, there will be some mismatches between certain feature points. That is why it is important to know how to deal with these mismatches (outliers). Thus, Fischler and Bolles [45] introduced the Random Sample Consensus (RANSAC) method to reject outliers from the set of keypoints. It is used in our visual odometry system to make it more robust against mismatches and other types of outlier points. This section will give a description of how it works.

RANSAC is an iterative method to calculate the parameters of model (in this case the PNP camera pose estimation) using observed data which contains some outliers. It makes the following two assumptions: First, there are enough good points (inliers) in the data set that fit to the actual model. Second, the noisy or mismatched points (outliers) are distributed randomly. If these two assumptions hold, the RANSAC algorithm is able to extract a set of points that is only containing all the inliers of the original data. This is done by randomly choosing a subset of points and supposing that those are all inliers. Based on this subset, a model is created which gets verified or falsified by comparing it

with all the remaining other points. Figure 3.8 shows a simple example that illustrates the approach.

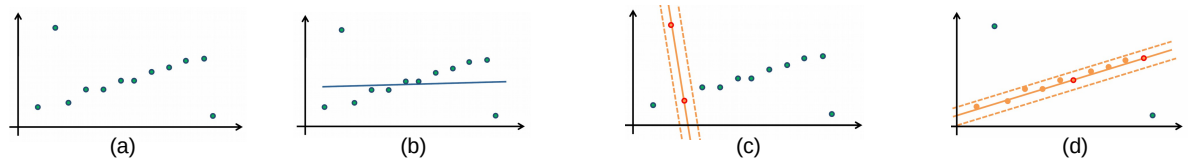

Figure 3.8: Example of RANSAC method [46]

In this example (a) shows a data set of some 2D points. In this particular model all the good points should lie on a straight line. Most points fit to that model, but there are two outliers. In (b) there is a line that minimizes the squares of the errors between the line and the data points (least squares method). One can easily see that it does not fit the points well, due to the two outliers. That is why it is so important to reject these outliers first before performing some optimisation algorithms. In (c) one iteration step of RANSAC is performed. Two points of the data are chosen randomly and are used for estimating the line. Then it is checked how many of the data points lie on this line within a certain deviation. In case (c), there are only two inliers (the two chosen points themselves) in that surrounding. In case (d) there are two other points chosen randomly. Now most of the data points are inliers. So this new line gives a good model for the given scenario.

For this work, we want to estimate the motion between two camera positions by using a PNP algorithm. It calculates the relative pose with 3D to 2D point correspondences. The RANSAC method is exploited to remove outliers (e.g. mismatches) in the 3D and 2D point sets. It operates in the following way:

1. Randomly select a minimal subset of s points to estimate the relative pose (in case of the P3P algorithm $s = 3$).
2. Determine the motion $(R|t)$ of the camera from these points.
3. Calculate the reprojection error of all the other data points for the previously estimated $(R|t)$. All points with an error below a certain threshold are stored in a set of inliers, while the rest are outliers.
4. Repeat step 1-3 for N iterations.
5. Select the largest inlier set that is found during the iterations, then optimize the estimation of the motion $(R|t)$ using this inlier set.

With that method it is feasible to reject all the outliers of a dataset and get a robust estimate for the camera motion $(R|t)$. The problem is, that many iterations will cost some precious computation time. The number N of iterations, which are needed for good results, relates to a few factors. The most important one is the ratio of inliers and outliers of the data set. In case, that there are a lot of outliers, it becomes less likely to randomly choose some points which are all inliers. The probability of choosing such a "good" set is

$$\omega = \frac{number\ of\ inliers}{number\ of\ all\ points} \qquad with \quad 0 < \omega \leq 1 \tag{3.29}$$

When requiring a minimum of s points in order to perform a certain algorithm, the probability of choosing s points consecutively that are all inliers is ω^s. If RANSAC should find such a set of only good points with the probability p, N iterations have to be performed:

$$N = \frac{log(1-p)}{log(1-\omega^s)} \tag{3.30}$$

In most cases, p is chosen very large (i.e. 0.999) to make sure that there is no outlier in the optimal point set. So the two factors inlier ratio ω and minimal subset points s are mainly determining the number of iterations. To keep this number as low as possible, ω needs to be maximized (close to 1) while s is minimized. Choosing the P3P algorithm for pose estimation, the smallest subset $s = 3$ for all possible PNP algorithms can be used. In order to make ω large, there should be a small amount of mismatches in the data set. This can be achieved by using a preferably exact feature selection and matching.

3.3.4 Concatenated Motion

The final part of the visual odometry algorithm is the estimation of the total pose of the robot. By solving the PNP problem, it is possible to calculate the motion of the robot between the last and current image. Hence, this rigid body motion is called $(R|t)_{current}$. In order to localise the robot, it is necessary to determine the whole trajectory from the beginning. To do so, the initial pose of the robot is defined as the reference coordinate system. The total pose of the robot is then calculated by simply concatenating the current motion and the total pose:

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_{total} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_{total} \cdot \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}_{current} \tag{3.31}$$

This calculation is done for each timestep, when the algorithm gets a new image from the video stream.

After all these calculation steps the actual visual odometry algorithm is completed. Estimating the total pose of the robot, it can be localised with reference to the starting point. However, there is a problem with calculating the total pose by concatenating all the current motions. Since the algorithm should work at high frame rates of the video stream, the total pose is updated very often. In case, there is only a little error in the previously estimated current pose, this error will accumulate. For example, the input video stream runs with 10 frames per second and the PNP algorithm generates a translational error of 1mm. After only 10 seconds, the total translation error might be already at 100mm ($1mm \cdot 10fps \cdot 10sec$).

This problem leads to the fact that the estimated position drifts away from the real pose, which is a general odometry problem. Since there is no data available to correct the calculated pose, little errors will propagate more and more over time. Figure 3.9 illustrates this problem. It shows the results of an visual odometry implementation in [47]. The red line shows the calculated trajectory of the VO algorithm. The blue dashed line shows, where the vehicle really is, according to GPS data. It can be easily seen that the VO performs quite good at the beginning, but the accuracy decreases a lot over time.

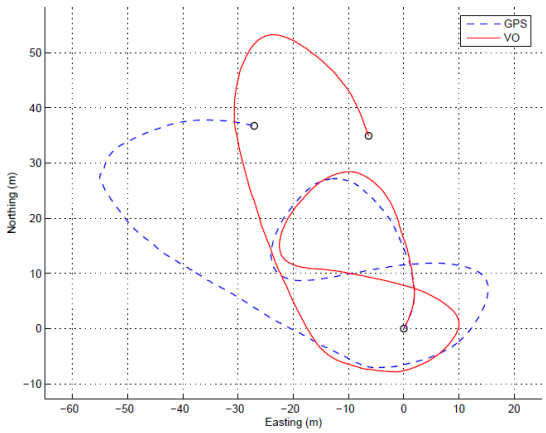

Figure 3.9: Drift of a VO implementation

A simple method to reduce this drift, is selecting only certain "keyframes" for the pose calculation. This means that current motions are only concatenated, if they are large enough. For example, while the robot stands still, the PNP algorithm yet estimates a small motion due to image noise. In this case, that image frame would be simply discarded, setting the motion of this frame to zero. Only if the motion is larger than a certain threshold (the robot is actually moving), the calculated $(R|t)$ is taken into account. This trick reduces the propagation of inaccurately estimated small motions caused by image noise, while the robot is not moving. On the other hand the threshold must not

be chosen too large. Due to a high threshold value, actual movements of the robot might be discarded as well, if they are very slow. That is why the minimum motion of the robot, which can be performed within one frame, is used to specify the motion threshold.

There are also many other approaches to reduce the issue of drifting results. Since these methods are partly quite complex, they are not implemented in this work, but they will be discussed in the last chapter.

3.4 Adaptive Feature Search

The previous sections illustrated how to estimate the position of the robot by using visual odometry. Furthermore, the problems of this technique were pointed out as well. The goal of this thesis is to implement an adaptive feature selection in order to improve the results of the VO algorithm. This means, that the robot should face various sectors of the environment, depending on its current position. Some regions are providing more features and some supply less. So the robot should search for those areas with most feature points, to get the best performance. Two different approaches of performing this feature search will be presented and tested in this work.

One of the biggest problems in all computer vision applications is that there are not many distinctive feature points in the image. For example, when looking at a plain white wall, the VO algorithm is not able to find significant points to match and orientate on. That is why the situations, where the field of vision is not containing enough feature points, have to be avoided. The robot which is used for this work, the Robotino, has an omnidirectional driving system, which facilitates the direct move in all directions and rotations on the spot. These properties are employed to find the areas with most feature points. While the robot is moving to a specified location in the room, it can rotate around its own axis. This technique is utilised in the first approach. Whenever the number of useful features that are in the field of view is falling below a certain threshold, the robot will rotate itself, until there are enough feature points in the camera's direction. This is a simple and very dynamic method, since the feature search is performed during the drive. So no time is dissipated for this task.

The problem of this dynamic method i, that the robot is only rotating when the feature points drop below a threshold. This only ensures that the robot is facing a reasonable area, otherwise it rotates. It does not make sure that the robot is facing the best area with most feature points. Therefore, the second approach is introduced. This more static method starts with an initial feature search in the beginning. First, the robot rotates 90° to the left and then 90° to the right, in order to scan the whole scene that is in front of the robot. It collects data of how many feature points are at which rotation

angle. After finishing the search, the direction with most features in it is known. Until this initial feature search is completed, the robot is standing on the same spot all the time, simply rotating around its own axis. This prohibits the introduction of translation errors during the scan. After facing the area with most feature points, the robot starts to move towards the target location. This static feature search should be performed again after some time, or if the number of feature points in the current direction is falling below a certain value. One problem which could occur during the feature search is that the robot is rotating into an area with no features at all. In order to avoid this situation, the robot only rotates so far that there are always still enough features in its field of vision. Otherwise it will rotate back to its original position.

So these two approaches of an adaptive feature selection ensure that the robot will not lose track of feature points while moving. This is an important extension to the VO algorithm to make it more robust for operations in unknown environments. In the end, both methods are tested and compared in some experimental scenarios, to find out which one works the best.

3.5 Robot Control

The last part of the algorithm of this work is the control of the robot. The location of the robot is previously estimated by visual odometry. Having this position information, the robot can drive to any desired position in the room. There are two constraints that should be fulfilled for the robots movement:

1. The robot should move from the starting point to the destination in a straight line, since this is the shortest path.

2. The control of the robot should be independent from its current rotation angle, because it might rotate during movement for finding more features.

The first step of designing a suitable control algorithm is to define a coordinate system for the robot. The visual odometry algorithm calculates the pose of the camera in a 3D space. However, in this work the robot is simply driving on a plane ground, which reduces the degrees of freedom (DoF) from six to only three DoF. The robot can only move in the x- and y-direction. Furthermore, there is only one rotation possible, which is going to be called θ. All the other pose information that are provided by VO will not be used in this case. Nevertheless, they might be relevant to some other applications, for example controlling a quadrocopter that is flying in the 3D space. Figure 3.10 shows the coordinate system of the VO algorithm (a) and the defined coordinate system of the Robotino (b).

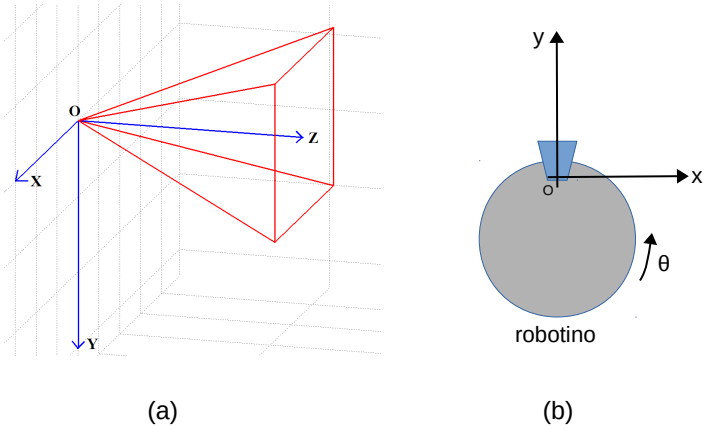

Figure 3.10: Coordinate systems of the camera (a) and the Robotino (b)

(b) shows the robot from the bird's eye view. The coordinate system is centred at the camera, so the hand eye calibration will be neglected at this point. While the x-axis remains the same in (a) and (b), the new y-axis is the z-axis of the VO algorithm. The rotation θ in (b) is the rotation around the y-axis in (a). After defining all these relations, the actual control algorithm can be explained.

The Robotino is controlled by setting its velocity in the x- and y-direction. In order to describe how these desired velocities are calculated, an example will be introduced. Figure 3.11 shows a possible scenario: The robot is at the position (1,1) of the world coordinate system. Furthermore, it is rotated by $\theta = 45°$. The desired destination is the point $P(4,3)$.

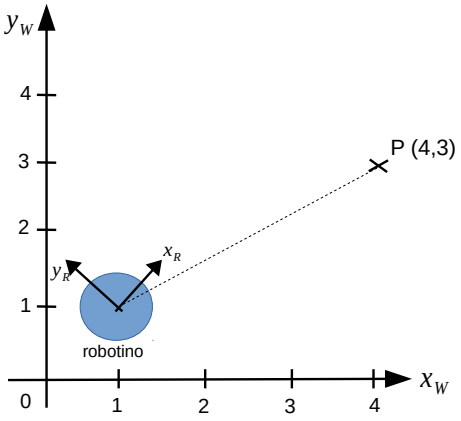

Figure 3.11: Example situation for velocity calculation

The first step of estimating the x- and y-velocities is to calculate the vector that is pointing from the Robotino to the destination. In this case, it would be $\overrightarrow{RP} = (3,2)^T$. This is the straight direction, the robot should be moving. Since this vector is relative to the world coordinate system, but the x- and y-velocity commands for the Robotino are only applicable for the Robotino coordinate system, this vector $\overrightarrow{RP}$ needs to be rotated. This is done by multiplying it with the 2D rotation matrix

$$\overrightarrow{RP}_{robotino} = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \cdot \overrightarrow{RP}_{world}. \tag{3.32}$$

In our example this would yield $\overrightarrow{RP}_{robotino} \approx (3.53, -0.70)^T$, where 3.53 would be the x-velocity and -0.70 the y-velocity. The last step is to scale these two values, preserving their ratio, to get suitable velocity values for the robot. Calculating the velocities for the Robotino in this way ensures that the robot moves in a straight line and that it can rotate around its own axis during movement, since it is independent from its rotation angle θ.

When the robot gets close to the defined destination, the control algorithm will slow it down a bit, so the Robotino can safely approximate the desired area. Finally, the whole algorithm terminates when the destination is reached.

Chapter 4

Implementation

This chapter is about the actual implementation of the adaptive visual odometry algorithm on a real robot. It gives an overview about the experimental setting and what the robot is supposed to do. Furthermore, the hardware (robot and stereo camera) which is used in the tests will be introduced. Subsequently, the structure of the software that implements the algorithm is shown. This also includes a detailed instruction of all the computation steps which are performed by the visual odometry algorithm. Then the implementation and the results of the calibration are pointed out. In order to evaluate the accuracy of the algorithm, a surveillance system for determining the ground truth is set up. Finally, everything is tested in various situations and test scenarios. The results of the experiments are presented and compared in the end.

4.1 Experimental Setting

The task of this work is to move the robot to a specified position in the room. The initial starting point of the robot in the room is known. For this task, it is required to know the exact position of the robot at every timestep. This self-localisation should be calculated by the visual odometry algorithm. Therefore, the robot is equipped with a stereo pair of cameras. Knowing its own position, the robot can be controlled by an algorithm to move to the desired destination. It is supposed to drive in a straight line, since this is the easiest and shortest path to get there. However, the robot is free to rotate around its own axis while moving. This can be done for changing the direction of view of the cameras.

As described in the previous chapters, the robot needs special feature points in its surroundings to estimate its own location. One special challenge for this work is that the laboratory room, where the experiments take place, only provides feature points in certain areas. Other areas of the room just consist of plain walls, which do not supply much distinctive points. So it is also part of the task, that the robot finds the rich-feature-area of the room. After finding those, it should rotate so the cameras are facing the locations with many features to orientate on. The laboratory room has a quadratic

form with 5m x 5m. The robot should only move in a certain part of this room, which is going to be called the robot environment. Figure 4.1 is showing the ground plot of the laboratory. The red marks are displaying how the feature points are distributed in the room.

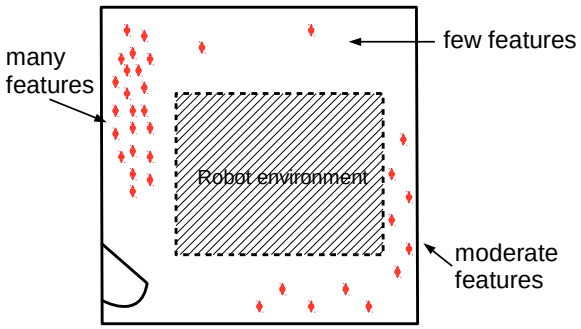

Figure 4.1: Ground plot of the laboratory room and the robot environment

4.2 Hard- and Software of the Experiment

4.2.1 Robotino

The robot that is used for this work is called “Robotino” from Festo Didactic. It is a mobile robot with many built-in sensors and interfaces [1]. Figure 4.2 shows a picture of the modified Robotino that is utilised in our experiments. The small modifications that have been made, are a protective metal housing and the attachment of a stereo camera. This camera is a stereo camera system from VRmagic that generates synchronised images, with a resolution of 754x480. The kinect in the picture is not used for this work, so it can be neglected. The visual odometry algorithm runs on the laptop that is lying on the Robotino as well. On top of the robot is an “Aruco marker”, which is needed for determining the ground truth (see chapter 4.5).

The most notable feature of this mobile robot are its omnidirectional wheels. These wheels make it possible for the Robotino to drive in every direction on the ground with no need to turn. Furthermore, it can rotate around itself staying in one place. This properties are really useful for our application, since the Robotino makes it easy to drive wherever it is needed without detours. The Robotino itself contains an embedded computer which is the interface to other computers for controlling and programming the Robotino. Moreover, an internal controller for the omnidirectional wheels is included, so it is not necessary to control each wheel by itself. One can rather assign velocity commands to the internal controller of the Robotino. This means that the robot is controlled by setting its velocity in x- and y-direction. Commands to rotate around its

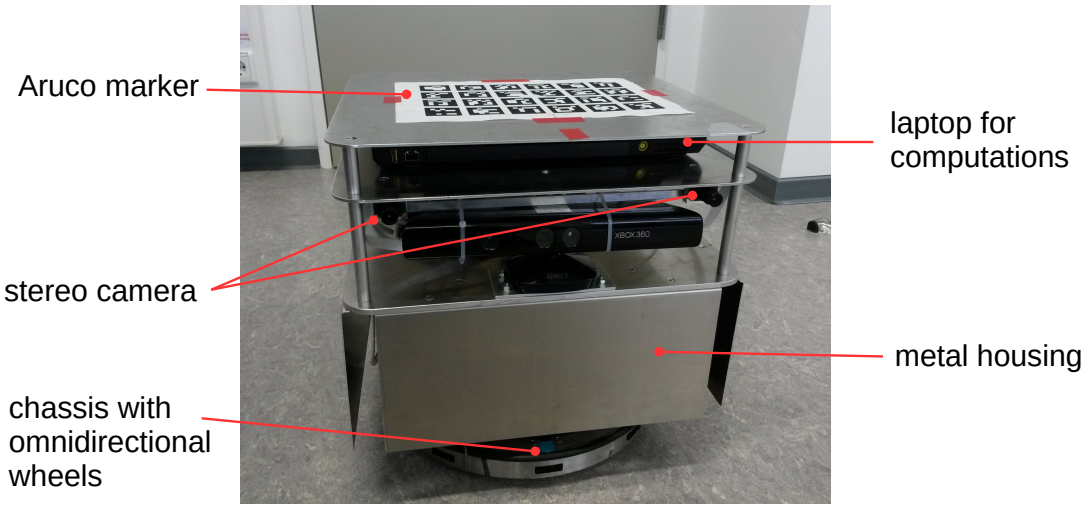

Figure 4.2: Robotino

own axis are possible as well. Another feature of the Robotino is that it has incremental encoders for each wheel, which make it possible to do a probabilistic prediction of the robots position according to the turning of the wheels (wheel odometry). In this work that information is not utilised, since the location should be only estimated by the visual data of the stereo camera. However, for other applications this information could be compared with the results of the implemented localisation algorithm. The Robotino has even more useful properties and sensors. They are not mentioned here, since they will not be used in this thesis. The focus of this work should be on employing visual odometry for localisation.

In order to communicate with the Robotino, it can be connected via WLAN/LAN. The interface makes it possible to use plenty of different programming languages like C++, Java, Labview, Matlab and many more. There is a framework called Robotino view as well that is especially designed for this robot. In our application, the robot will be programmed using the robot operating system ROS which is explained in the next section.

4.2.2 ROS

The Robot Operating System (ROS)[1] is a flexible framework especially for developing software for robots. It can be used for robots industry by programming in a modular way. There are many packages available that contain certain functionalities for robots and also hardware drivers. These packages often consist of one or more so called “Nodes”.

[1] `http://www.ros.org/`

Nodes are something like an executable program which perform a specific task. There is already a package for the Robotino available which contains many useful nodes. For example, these nodes implement that the sensors of the robot can be read out or that the robot can be controlled with a keyboard or joystick. Figure 4.3 shows the structure of the ROS system that is used in this work.

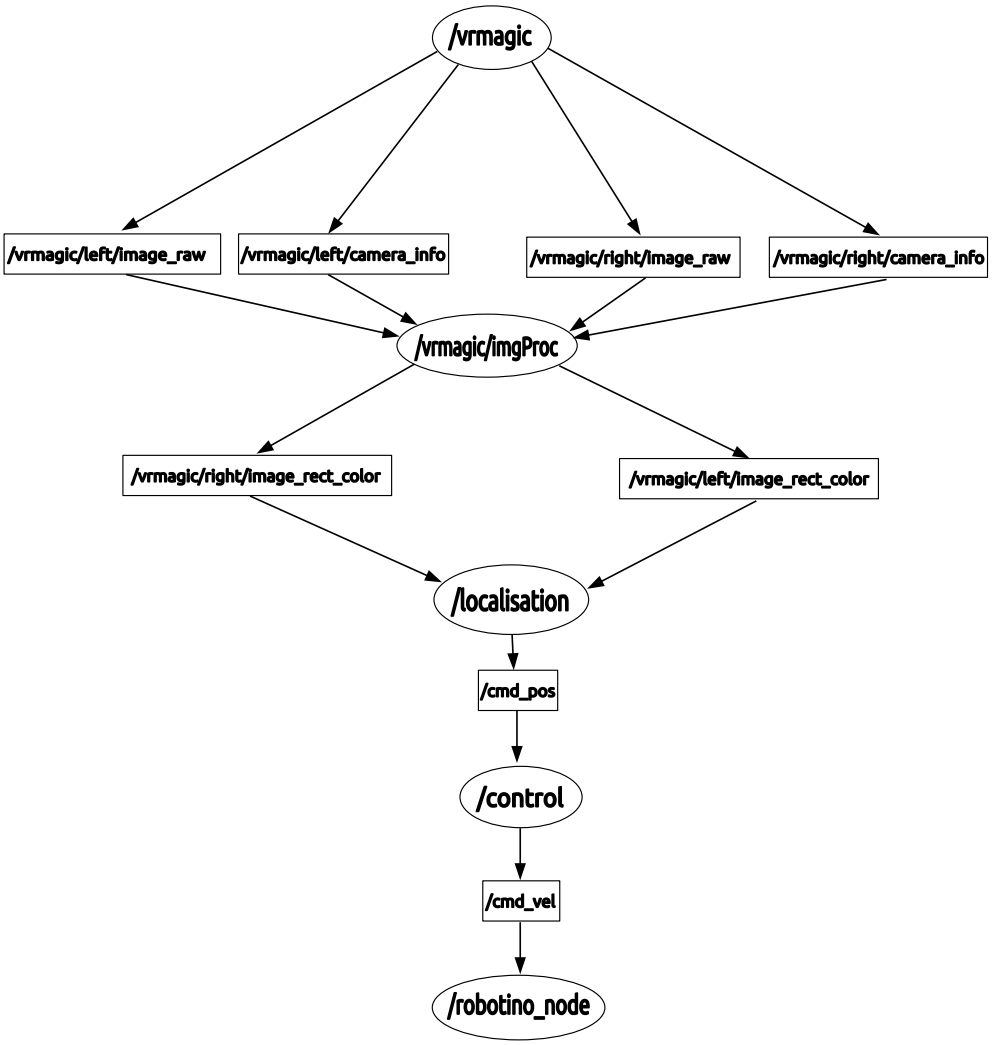

Figure 4.3: Structure of the ROS system

`vrmagic`, `imgProc`, `localisation`, `control` and `robotino_node` are the nodes that are employed in our system. They can communicate with each other using ROS messages. There are different types of ROS messages. For example, these can be simple strings, pose information (a set of $(R|t)$ values) or also video streams. Messages are published

via a "Topic". The node which is sending the information is depicted as a "Publisher" and the receiver is denoted as a "Subscriber". Nodes are marked by a circle, while topics are framed with a rectangle.

The first node of the program is `vrmagic`. This is the driver for the stereo camera. It receives the image data and publishes the raw left and right image to the `imgProc` node. Furthermore, it sends `camera_info` which is containing the calibration data of the stereo camera. So `imgProc` receives the raw video stream and then does a pre-processing with the images. Since it also has the calibration data of the cameras, it undistorts the two images and after that it rectifies them for further calculations. The node `imgProc` is already provided by the ROS community, so it is not written as a part of this thesis. After the processing, the node passes the rectified images (`image_rect_colour`) to `localisation`. This node is the central part of this work, since it contains the implementation of the visual odometry algorithm. It receives a stream of rectified images and then calculates the current location of the robot with that information. It also performs the adaptive feature search. After determining the pose information (`cmd_pos`), this gets published to the node `control`. Thus, `control` knows about the position and orientation of the robot. Moreover, the destination of the robot is set in this node as well. Therefore, it calculates the desired x- and y- velocities for the robot that move it to the target location. These velocity commands (`cmd_vel`) are then published to the `robotino_node`, which is the interfacing node to the Robotino. It subscribes the velocity commands and runs the robot with the specified velocities.

4.2.3 OpenCV

While the Robotino is controlled with ROS, most of the actual image processing work and computer vision algorithms are programmed in C++ code. The ROS package "roscpp" implements this C++ code in ROS nodes, so it can be used with other ROS Nodes and Topics. Thereby, it is finally possible to control the robot with a C++ program via ROS. The reason why the programming language C++ was chosen for the implementations is simple. Besides its good computation speed, there is a big community behind it, which generates a lot of support and offers useful open source libraries for computer vision. One of those libraries is OpenCV[2], that is mainly used for this work.

OpenCV contains many programming functions and algorithms for computer vision applications. That includes

- Loading images and videos from files or live cameras,
- Processing images (e.g. change color space, extract regions of interest etc.),

[2] `http://opencv.org/`

- Easy matrix calculations,
- Creating a graphical user interface (GUI),
- Performing camera calibrations,
- Feature detection and matching,
- Pose estimation from feature points,
- And many more.

Furthermore, OpenCV is designed for computational efficiency and has a strong focus on real-time applications. This becomes very important due to the fact, that image processing has very high computational costs when live video streams at a high frame rates are used for the calculations.

4.3 Algorithm Implementation

This section will point out the implementation of the actual localisation algorithm. The first step is the undistortion and rectification of the raw images. This is done by the ROS node `imgProc`. These pre-processed images are then passed to the `localisation` node, where the pose of the robot is computed. The advantage of this partition is that both nodes (`imgProc` and `localisation`) can run in parallel. That saves some time for the visual odometry algorithm, since it does not need to rectify images anymore. The further computation steps of the `localisation` node are the following:

1. Grab the rectified and undistorted images from the left and right camera.
2. Turn images into grayscale for further processing.
3. Detect ORB features in the left and right image and extract feature descriptors.
4. Match these feature points with a brute force matcher using the hamming distance and a cross check.
5. Selecting only the best matches by performing these three steps:
   - Since images are rectified, only choose matches that are lying on the same y-coordinate (± 3 pixel).
   - Only select matches with a distance below a threshold.
   - Perform RANSAC to find the fundamental matrix of the left and right image and remove outlying points.

6. Triangulate 3D points from the set of matched feature points with the linear triangulation method.
7. Ttrack the selected set of feature points in the next image using the Lucas-Kanade tracker.
8. Estimate $(R|t)$ of the stereo camera with the P3P algorithm and RANSAC.
9. Check if this $(R|t)$ is large enough, otherwise discard it (keyframe selection).
10. Calculate the movement of the robot from the movement of the camera by applying the Hand-Eye-Calibration.
11. Concatenate the calculated $(R|t)$ to receive the total pose.
12. Draw feature points into the images for visualisation.
13. Publish pose information and start the loop again for the next image frame.

The visualisation of this node shows the images of the left and right camera. In the left image all feature points, that could be matched, are drawn. Furthermore, it shows a line between the location of a feature point in the last and current frame. This illustrates the motion of the camera. Figure 4.4 is an example of this output. The image of the right camera is shown in another window. Feature points are drawn into the image as well. In this case they are coloured according to their depth in the scene like in figure 3.7.

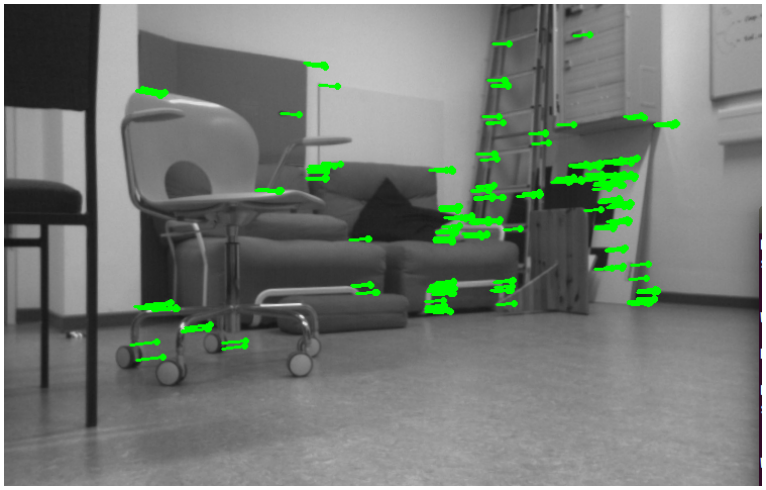

Figure 4.4: Visualisation of the left camera image

The `localisation` node also performs the adaptive feature search. Therefore, it compares the number of matched feature points with a specified threshold value. If there are

not enough feature points in the image, the node publishes the command to rotate the Robotino for finding more features like described in chapter 3.4. After all, the published pose information is processed by the `control` node. This node calculates the desired x- and y-velocities as depicted in chapter 3.5.

4.4 Calibration Results

The very first step for all of our computer vision applications is a camera calibration. In order to do this, two different camera calibration implementations were tested:

1. The Camera Calibration Toolbox for Matlab from Jean-Yves Bouguet[3].
2. The calib3d module of the open source C++ library OpenCV[4].

Both implementations work quite the same way. The calibration rig is a standard checkerboard which is printed on a plane surface. About 20 pictures of this checkerboard were taken from different angles and positions for the camera calibration. After loading these images to the program, the height and width of the board (number of black and white squares) has to be set as well as the square size. Subsequently, both implementations detect the corners of the checkerboard in the images automatically and then calculate the intrinsic and extrinsic parameters according to [48]. Even though, the results of both implementations using the same pictures were not exactly the same, they fit quite good together. So it will not make a big difference which program is used for calibrating the cameras.

For most calibrations, the OpenCV implementation was employed, because it automatically generates a .xml file with all camera parameters written in it. That is useful for future computations that also work with OpenCV, since that file can be loaded directly into the C++ program. Nevertheless, the Hand-Eye-Calibration was done with the Matlab toolbox due to its easy matrix calculations that are possible in Matlab. Figure 4.5 shows the GUI of the Matlab calibration toolbox.

[3] `http://www.vision.caltech.edu/bouguetj/calib_doc/`

[4] `http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html`

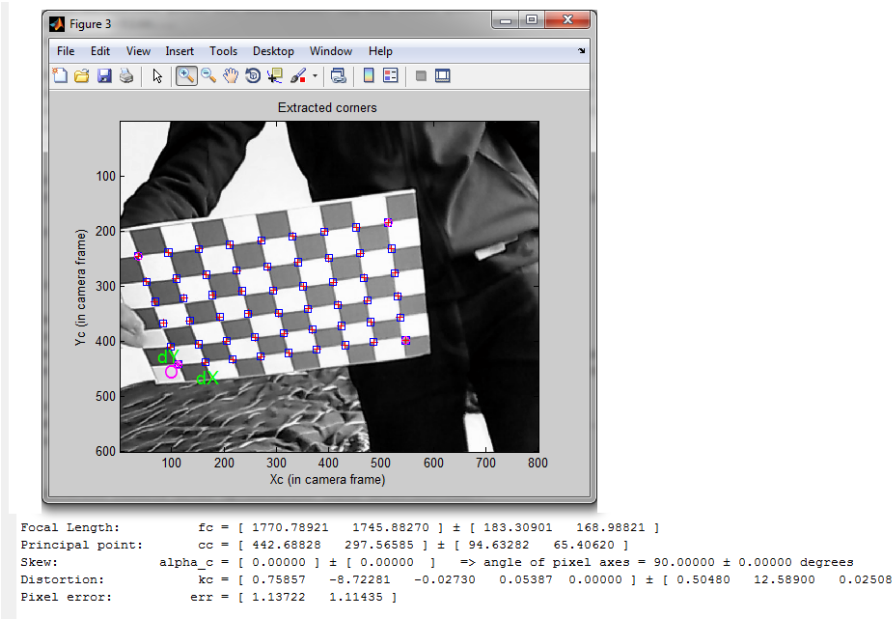

Figure 4.5: Matlab calibration toolbox showing the detected corners of the checkerboard and calibration results

The calibration of the stereo camera yields the following results for the left and right camera matrix $\mathbf{K_l}$, $\mathbf{K_r}$ and distortion coefficients $\mathbf{d_l}$, $\mathbf{d_r}$:

$$\mathbf{K_l} = \begin{pmatrix} 689.67 & 0 & 377.21 \\ 0 & 689.17 & 241.97 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{K_r} = \begin{pmatrix} 689.08 & 0 & 384.31 \\ 0 & 687.27 & 241.74 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{d_l} = \begin{pmatrix} -0.45661 & 0.19879 & 0.00169 & -0.00227 & 0 \end{pmatrix}$$
$$\mathbf{d_r} = \begin{pmatrix} -0.45797 & 0.17587 & -0.00084 & -0.00729 & 0 \end{pmatrix}$$

The rigid body motion from the Robotino center to the left stereo camera (Hand-Eye-Calibration) is

$$(\mathbf{R}|\mathbf{t})_{\mathbf{R}}^{\mathbf{C}} = \begin{pmatrix} 0.9998 & 0.0182 & -0.0086 & -188.0 \\ -0.0181 & 0.9997 & 0.0369 & 270.0 \\ 0.0088 & -0.0135 & 0.9998 & 140.0 \end{pmatrix}.$$

4.5 Ground Truth Setup

The localisation of the robot is performed by using the images of the stereo camera, which is mounted on the robot itself. Therefore, the algorithm of this work tries to make

an estimation of the robots pose. However this estimation will not be totally accurate. In order to evaluate how well the algorithm performs, it is important to know where the robot really is. This real trajectory of the robot is called ground truth[49]. For setting up this ground truth data, a camera surveillance system was built up above the robot environment. It consists of four logitech webcams that cover the whole area where the robot is moving. The field of view of one or two cameras would be too small to cover this whole area. That is the reason why four cameras, arranged in a rectangle, are used for setting up the ground truth. Technically one camera would be enough to track the robot and calculate its pose in the room. Figure 4.6 shows the setup of the four surveillance cameras.

Figure 4.6: Setup of the surveillance system

In order to track the position of the robot with the cameras, Aruco[5] markers are employed. These markers consist of black and white blocks in a known plane formation (see figure 4.7), so it is easy to detect them in an image. There is an Aruco library that implements an algorithm which is tracking these markers. Once the algorithm has found a marker, it computes the extrinsic parameters of the camera relatively to the marker. This is possible, because the arrangement of the marker is known and so the pose of the camera can be extracted like in a normal calibration. [50] describes how the Aruco markers work in detail. Hence, the only thing that has to be done is attaching such a marker on top of the Robotino in a way, that the surveillance cameras can register it. With the help of the Aruco library, it is possible to find the pose of the robot relatively to the surveillance camera system in real time. This rigid body transformation will be denoted as $(R|t)^{M}_{SC}$ (pose from surveillance camera to the marker).

[5] `http://www.uco.es/investiga/grupos/ava/node/26`

Figure 4.7 shows a modified example program from the Aruco library, where an Aruco marker which is detected in a live video stream can be seen. The console updates the rigid transformation $(R|t)$ from the camera to the marker in real time.

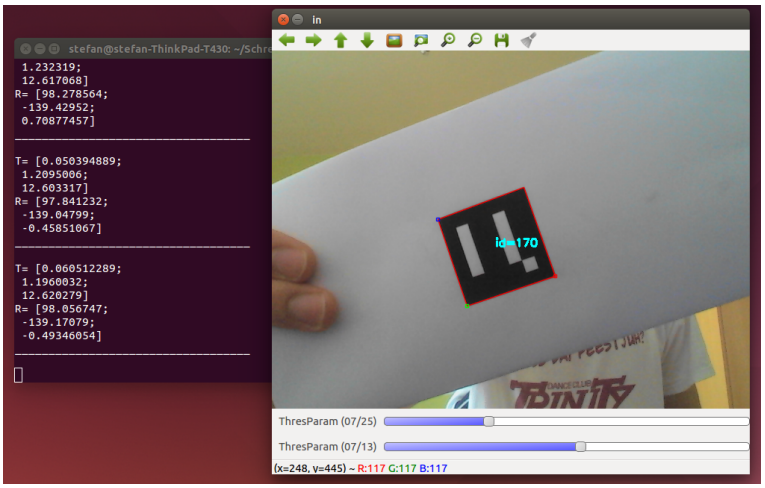

Figure 4.7: Detected Aruco marker with calculated $(R|t)^{M}_{SC}$ (in console window)

On account of the fact, that it is required to know the robots position in the room and not relatively to the camera, the four cameras have to be extrinsicly calibrated before. Placing the calibration rig in a corner of the robot environment, this point can be defined as the coordinate system center of the room. After computing the extrinsic parameters of all cameras towards that calibration rig, all four rigid body transformations $(R|t)^{Env}_{SCi}$ from each surveillance camera SC_i with $i = 1...4$ to the coordinate frame center of the robot environment can be calculated. In order to get the robots pose (respectively rather the pose of the Aruco marker attached to the robot) relatively to the robot environment frame $(R|t)^{M}_{Env}$, the following calculation is performed:

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}^{M}_{Env} = \left(\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}^{Env}_{SCi}\right)^{-1} \cdot \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}^{M}_{SCi} \tag{4.1}$$

In the end, we have a system that makes it easy to track the robots pose in real time during the experiments. That makes it possible to compare and evaluate the real ground truth trajectory with the estimations of the visual odometry algorithm of this work. It is important to note that the robot itself will not use any of these ground truth information for its self-localisation.

4.6 Experimental Results

In order to make a statement about the accuracy of the implemented localisation algorithm, some experiments were performed. This section will illustrate these experiments and their results. The quality of a visual odometry algorithm can be expressed with the relative error between the estimated pose and the actual pose (ground truth) of the robot. This comprises the translational and rotational error. The translational error ϵ_{trans} will be calculated by

$$\epsilon_{trans} = \frac{\Delta t}{t_{total}} = \frac{|\Delta t_x| + |\Delta t_y| + |\Delta t_z|}{|t_x| + |t_y| + |t_z|}, \tag{4.2}$$

where t_{total} is the total translation of ground truth and Δt is the difference between the ground truth translation and the translation that is estimated by the algorithm. The relative rotational error ϵ_{rot} is calculated in relation to t_{total} as well. Δr is describing the difference between the real and estimated rotation:

$$\epsilon_{rot} = \frac{\Delta r}{t_{total}} = \frac{|\Delta \Phi| + |\Delta \Theta| + |\Delta \Psi|}{|t_x| + |t_y| + |t_z|} \tag{4.3}$$

Before the localisation algorithm was implemented on the real robot, first it was tested on a simulation data set. The "KITTI Vision Benchmark Suite" [51] is providing a large amount of image sequences that are recorded by a car driving through different environments (cities, roads, residential areas etc.). It also supplies the calibration of the stereo camera and ground truth data of the car. So this benchmark suite can be used to test the visual odometry algorithm safely and to tune parameters. For our experiments, two different data sets have been taken into account. In the first one, the car drives about 30m, while in the other set, the car is travelling a total distance of about 200m. The results of these tests were quite good. The translational error for the first set was below 1% while the second set (longer distance) yields an error of 2.3%. The rotational error amounts 0.35°/m and 0.03°/m.

These previous simulations have proven, that the visual odometry algorithm is working in general. Then it was tested on the Robotino in our lab. The actual idea of gaining ground truth data from the surveillance system did not work out in the end. We had some hardware issues, with processing four video streams at the same time. Furthermore, the accuracy of the Aruco markers was not as good as it was expected. That is why the actual location of the robot was simply measured by hand. This worked well for the translation, however it was hard to get accurate results for the rotation.

In our experiments, some different scenarios were tested to gain information about how the algorithm is working in various situations. In all tests the Robotino had to do two tasks. First, it should simply drive to a specified position in the room. In the other

task it should move consecutively to four different points in a rectangular trajectory. The totally travelled distance for these experiments in the laboratory room was usually between 2m and 4m.

In the first test scenario, the Robotino was already facing a feature rich region, so no feature search needs to be performed. This test situation simply examines the accuracy of the VO algorithm without the adaptive feature search. The translational error was about 5% and a rotational error of roughly 1°/m was achieved. These results are quite good, although the error is higher than in the KITTI simulation.

In a second test scenario, the robot was facing an area with only few feature points. So in this case the adaptive feature search should be applied. Figure 4.8 is showing this scenario. Here the Robotino has to find the area with most features for orientation.

Figure 4.8: Robotino in the laboratory environment with areas of many (left side) and few (right side) features

There are two approaches for the adaptive feature selection like described in chapter 3.4. In the first one, the robot directly starts to move to the target location. Whenever the number of detected feature points is too low, it starts to rotate while it is driving. This is a fast and dynamic method, since no time is wasted for the feature search. The other technique starts with an initial feature search. This means that the robot stays in place and first scans the environment by rotating around its own axis. After finding the area with most features it rotates towards this area. Then it starts to move to the specified location. This static feature search is repeated after some time, to ensure that the robot is always facing the feature richest areas.

The results of the dynamic feature search are indicating, that this method is not really feasible. While it is moving, the Robotino rotates many times to find feature points.

This introduces a quite large rotational error, which also increases the translational error. In the end the robot does not move to the target area at all. The error of the position is about 60%, so this approach is quite futile. In contrast to that, the static feature search performs much better. First, the feature search itself works really well. The robot can easily rotate on the spot, without translating. After the feature search is completed, the robot turns towards the feature richest area. This search process does not introduce any translational error and only a rotational error of ca. 1°, which is quite good. However, it takes a while (about half a minute) to complete the whole initial feature search. Then the robot is moving to the specified destination. In the end, the result of this test yields a translational error of 8% and about 1.5°/m in rotation. So the accuracy is decreasing a little bit compared to the test where no adaptive feature search is performed. This is due to the fact, that an additional rotational error is introduced during the feature search, since the robot is rotating a lot.

Table 4.1 shows a comparison of the results of all tested scenarios. It can be seen that the results of the KITTI data set are a bit better, than the performance of the Robotino in our experimental setup. However, the outcome of the Robotino experiments is still reasonable for the methods used. When applying an adaptive feature search (FS), it is much more beneficial to utilise the static approach than the dynamic.

Test Scenario	Rotational Error in °/m	Translational Error in %
KITTI data set 30m	0.35	0.9
KITTI data set 200m	0.03	2.5
Robotino no adaptive FS	~ 1	5
Robotino with dynamic FS	~ 2	~ 60
Robotino with static FS	~ 1.5	8

Table 4.1: Results of the different tests

Chapter 5

Conclusion

5.1 Summary

This work is about developing and implementing an adaptive visual odometry algorithm for the self-localisation of mobile robots. After giving a foundation of this topic, the designed VO algorithm was illustrated and discussed in detail. Finally, it was shown how this algorithm got implemented on the Robotino. Some experiments of different test scenarios were performed to find out about the accuracy of the developed algorithm.

The test results have shown that the algorithm works the best on the KITTI benchmark suit. There might be various reasons, why the outcome of this data set outperforms our tests with the Robotino. First, it supplies an optimal calibration and images of a high quality. Furthermore, the totally travelled distance in these data sets is much higher compared to our experiments in the laboratory room (about 10-100 times). That is why small motion errors are having a larger effect on the tests in our lab.

Even if the achieved accuracy for the Robotino is not optimal, the implemented VO algorithm is still feasible for localising and controlling the robot. Moreover, an adaptive feature selection has been implemented. The experiments have pointed out that the static approach, of an initial feature search should be utilised. This method works really good, since it ensures that the robot is always facing the area of the room with most features in it. Even if this feature search introduces a little error compared to the the test scenario without the adaptive feature selection, it is beneficial to make use of it. When operating the robot in an unknown environment, it is essential to have feature points to orientate. The adaptive feature selection algorithm makes sure, that the robot does not lose track of these features.

All together, a working system was implemented in this work. The self-localisation is computed by a visual odometry algorithm, which is improved by an adaptive feature selection. This makes it more robust against environments with sparse feature points.

5.2 Future Work and Outlook

The experimental results have pointed out that the adaptive visual odometry algorithm works in general, but that there are still some demands on achieving a better accuracy. In order to improve the results of the localisation algorithm, there are a few possible extensions to the methods used in work. So this last section of the thesis will illustrate some approaches that may be implemented in the future to achieve a better overall performance.

One important aspect that was neglected in this work, is the appearance of moving objects in the scene. For all calculations in this thesis it was assumed that the whole scene is static. However, in reality there will often be situations where moving objects (like humans, cars, other robots etc.) are part of the image. So it would be beneficial to implement an algorithm that is able to deal with other motions in its environment. Since RANSAC is employed for pose estimation, this algorithm is already a bit robust against moving parts. Though, these parts need to be quite small compared to the rest of the scene, so RANSAC can remove them as outliers.

The other main problem with visual odometry is the drift of the calculated position. In order to get more accurate values, it is essential to reduce this drift as much as possible. Selecting only those frames where a large movement happens is just a very simple technique to do so and it will not work in all situations. A better solution is to filter the results with a Kalman or Particle filter. Based on the previous states (e.g. position, velocity) of the robot, these filters make an assumption, where the robot should be at a certain timestep. This can be compared with the calculated value of the VO algorithm, refining the final results. Moreover, these filters make it possible to utilise further sensor data and fuse them together. For example, in our case the wheel odometry of the Robotino could be taken into account as well for the localisation task.

When calculating the motion of the robot, the VO algorithm is triangulating the 3D world points of its environment. Nevertheless, these scene points are only used for pose estimation. It would be beneficial to use the knowledge about the structure of the surrounding area to build a map of the room. This information can be used for different applications: First a collision avoidance could be performed. Since the system is aware of the obstacles in the room, these objects can be circumnavigated by the robot, if they are in the way to the specified target location. This prevents the robot from bumping into any barriers that are in the (unknown) setting. Another possibility of employing the 3D scene information, is to exploit it for an improved self-localisation by reducing the drift of the visual odometry. In case of having a detailed map of the environment, it would be possible to localise the robot on this map. Having fixed points for orientation, the robots position can be determined easily. Since this position is calculated relatively to static landmarks, the results will not drift away from the actual location of the robot.

This approach is called simultaneous localisation and mapping (SLAM), which solves the loop-closure problem. While the robot explores the scene, a more and more accurate map is generated for localisation.

Finally, one additional task for the future is to make the surveillance system running. While the general construction is set up already, it needs to be capable of tracking the exact location and orientation of the Robotino. Otherwise it is impossible to evaluate the experimental results correctly in order to further improve the algorithm. A very accurate tracking system could also be used for a more precise Hand-Eye-Calibration by eliminating the uncertainties of the manual measurements.

In conclusion, there are still many interesting approaches to enhance the accuracy and robustness of the localisation algorithm. Extending the current algorithm with these methods, an improved approach for future applications can be developed. The adaptive feature search turned out to be a useful technique. Furthermore, the whole implementation got up and running on a real robot system. So in the end, this work provides a solid foundation for further research in this area.

Bibliography

[1] Festo Didactic GmbH & Co. KG. *Robotino Manual*, 2007.

[2] Alfred Kleusberg. Analytical gps navigation solution.

[3] National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center. Lidar 101: An introduction to lidar technology, data, and applications, 2012.

[4] Xiaoping Yun, E.R. Bachmann, H. Moore, and J. Calusdian. Self-contained position tracking of human movement using small inertial/magnetic sensor modules. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 2526 – 2533, april 2007.

[5] Mark W. Maimone, Yang Cheng, and Larry Matthies. Two years of visual odometry on the mars exploration rovers. *J. Field Robotics*, 24(3):169–186, 2007.

[6] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE Robot. Automat. Mag.*, 18(4):80–92, 2011.

[7] David Nister, Oleg Naroditsky, and James Bergen. Visual odometry for ground vehicle applications. *Journal of Field Robotics*, 23:2006, 2006.

[8] Jason Campbell, Rahul Sukthankar, Illah R. Nourbakhsh, and Aroon Pahwa. A robust visual odometry and precipice detection system using consumer-grade monocular vision. In *ICRA*, pages 3421–3427. IEEE, 2005.

[9] Andrew Howard. Real-time stereo visual odometry for autonomous ground vehicles. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, September 22-26, 2008, Acropolis Convention Center, Nice, France*, pages 3946–3952, 2008.

[10] Wei Mou, Han Wang, and Gerald Seet. Efficient visual odometry estimation using stereo camera. In *11th IEEE International Conference on Control & Automation, ICCA 2014, Taichung, Taiwan, June 18-20, 2014*, pages 1399–1403, 2014.

[11] Bernd Kitt, Andreas Geiger, and Henning Lategahn. Visual odometry based on stereo image sequences with ransac-based outlier rejection scheme. In *Intelligent Vehicles Symposium*, pages 486–492. IEEE, 2010.

[12] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. *ACM Trans. Graph.*, 25(3):835–846, July 2006.

[13] Christian Wöhler. *3D Computer Vision: Efficient Methods and Applications*. Springer, London, 2 edition, 2013.

[14] S. Rahmann and H. Burkhardt. *Kamerakalibrierung*.

[15] Hernan Badino. *Binocular ego-motion estimation for automotive applications*. PhD thesis, Goethe University Frankfurt am Main, 2009. http://d-nb.info/992453542.

[16] Gregory G. Slabaugh. *Computing Euler angles from a rotation matrix*, 2000.

[17] Emod Kovacs. Rotation about an arbitrary axis and reflection through an arbitrary plane. *Annales Mathematicae et Informaticae*, 2012.

[18] Y. Ma, S. Soatto, J. Kosecka, and S.S Sastry. *An Invitation to 3-D Vision*. Springer, 2004.

[19] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, November 2000.

[20] Zhengdong Zhang, Y. Matsushita, and Yi Ma. Camera calibration with lens distortion from low-rank textures. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 2321–2328, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Ananth Ranganathan Th. The levenberg-marquardt algorithm, 2004.

[22] Klaus H. Strobl and Gerd Hirzinger. Optimal hand-eye calibration. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2006, October 9-15, 2006, Beijing, China*, pages 4647–4653, 2006.

[23] Fabian Wenzel. *Robust relative pose estimation of two cameras by decomposing epipolar geometry*. PhD thesis, Hamburg University of Technology, 2007. http://d-nb.info/984801995.

[24] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[25] Epipolar geometry. `http://docs.opencv.org/master/da/de9/tutorial_py_epipolar_geometry.html`, October 2015.

[26] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[27] Silvio Savarese. Model used for image rectification example. `https://en.wikipedia.org/wiki/Image_rectification`, October 2015.

[28] Tony Lindeberg. Edge detection and ridge detection with automatic scale selection. *International Journal of Computer Vision*, 30:465–470, 1996.

[29] C G Harris and J M Pike. 3d positional integration from image sequences. In *In Proc. Alvey Vision Conference, Cambridge.England*, 1987.

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

[31] Jianbo Shi and Carlo Tomasi. Good features to track. In *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 593 – 600, 1994.

[32] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Proceedings of the 9th European Conference on Computer Vision - Volume Part I*, ECCV'06, pages 430–443, Berlin, Heidelberg, 2006. Springer-Verlag.

[33] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[34] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008.

[35] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: binary robust independent elementary features. In *Proceedings of the 11th European conference on Computer vision: Part IV*, ECCV'10, pages 778–792, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2564–2571, Washington, DC, USA, 2011. IEEE Computer Society.

[37] Cmglee. Visual representation of an image pyramid with 5 levels. `https://en.wikipedia.org/wiki/Pyramid_(image_processing)`, October 2015.

[38] J. Rabin, J. Delon, and Y. Gousseau. A statistical approach to the matching of local features. *SIAM J. Img. Sci.*, 2(3):931–958, September 2009.

[39] Marius Muja and David G. Lowe. Fast matching of binary features. In *Proceedings of the 2012 Ninth Conference on Computer and Robot Vision*, CRV '12, pages 404–410, Washington, DC, USA, 2012. IEEE Computer Society.

[40] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[41] Richard Hartley and Peter Sturm. Triangulation, 1996.

[42] Hatem Said Alismail, Brett Browning, and M Bernardine Dias. Evaluating pose estimation methods for stereo visual odometry on robots. In *the 11th International Conference on Intelligent Autonomous Systems (IAS-11)*, 2011.

[43] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[44] Robert M. Haralick, Chung-Nan Lee, Karsten Ottenberg, and Michael Nölle. Review and analysis of solutions of the three point perspective pose estimation problem. *Int. J. Comput. Vision*, 13(3):331–356, December 1994.

[45] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[46] Dr. Sturm. Lecture: Visual navigation for flying robots. `http://vision.in.tum.de/teaching/ss2013/visnav2013`, October 2015.

[47] Jonathan Kelly and Gaurav S. Sukhatme. An experimental study of aerial stereo visual odometry. In *Symposium on Intelligent Autonomous Vehicles*, Toulouse, France, Sep 2007.

[48] Janne Heikkila and Olli Silven. A four-step camera calibration procedure with implicit image correction. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, CVPR '97, pages 1106–, Washington, DC, USA, 1997. IEEE Computer Society.

[49] Scott Krig. *Computer Vision Metrics: Survey, Taxonomy, and Analysis*. Apress, Berkely, CA, USA, 1st edition, 2014.

[50] S. Garrido-Jurado, R. Muñoz Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recogn.*, 47(6):2280–2292, June 2014.

[51] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and PatternRecognition (CVPR)*, 2012.