Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Institut für Informatik
Lehrstuhl VII
Robotik und Telematik
Prof. Dr. Klaus Schilling

Distributed Spacecraft 3D Pose and Position Estimation based on Camera Images

Masterarbeit im Fach Informatik
vorgelegt von

Anna Aumann

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Institut für Informatik
Lehrstuhl VII
Robotik und Telematik
Prof. Dr. Klaus Schilling

Distributed Spacecraft 3D Pose and Position Estimation based on Camera Images

Masterarbeit im Fach Informatik
vorgelegt von

Anna Aumann
geboren am 05.11.1991 in Worms

Angefertigt am
Lehrstuhl für Robotik und Telematik
Bayerische Julius–Maximilians–Universität Würzburg

Betreuer:
Prof. Dr. A. Nüchter
Dipl.-Inform. F. Kempf

Abgabe der Arbeit:
31.03.2017

Erklärung

Ich versichere, die vorliegende Masterarbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt zu haben.

Würzburg, den 31.03.2017

(Anna Aumann)

Zusammenfassung

Das Interesse an Raumfahrtszenarien, in denen zwei Satelliten auf engem Raum kooperieren müssen, wächst. Beim sogenannten On-Orbit-Servicing, werden Satelliten wieder aufgetankt, repariert oder defekte Satelliten aus ihrem Orbit entfernt. Ziel ist die Reduzierung von Weltraumschrott. Besonders im Zusammenhang mit diesen Szenarien werden bildverarbeitende Methoden für Navigation und Automatisierung der Satelliten erforscht. Auch diese Arbeit basiert auf einem On Orbit Servicing (OOS) Szenario. Ein Satellit ist ausgestattet mit einem abnehmbaren robotischen Arm. Zwei mögliche Situationen sind denkbar: entweder der Satellit selbst oder der Roboterarm sind defekt. In beiden Fällen ist die Abnahme des Arms erwünscht - zum Austausch oder zur Wiederverwertung. Der robotische Arm besteht unter anderem auch aus zylinderförmigen Teilen. Das Ziel dieser Arbeit ist es daher, einen Algorithmus zu entwerfen, der die 3D Pose eines zylindrischen Objekts anhand von Kameraaufnahmen schätzen kann.

Neue Algorithmen für optische Navigation wie eben Algorithmen für Positionsschätzung, werden zunächst auf Computern getestet, die in den meisten Fällen eine höhere Rechenkapazität besitzen als ein Satellit. Später werden sie auf für Bildverarbeitung spezialisierter Hardware implementiert, z.B. einem Field Programmable Gate Array (FPGA) oder Digital Signal Processor (DSP). Diese führen genau den einen Zweck aus - ein Ansatz, der der üblichen Herangehensweise an den Entwurf der Borddatenverarbeitung eines Satelliten entspricht. Eine gegensätzliche Idee wird vom Projekt phYsical distributEd conTrol in spacE (YETE) an der Universität Würzburg verfolgt: die Verwendung mehrerer gewöhnlicher Allzweckprozessoren, die kabellos kommunizieren.

Um die Verwendung spezialisierter Hardware vermeiden zu können, wird der entworfene Algorithmus nach Möglichkeit zur verteilten Anwendung analysiert. Die Arbeit beginnt mit einer Einführung in häufig verwendete bildverarbeitende Algorithmen in Raumfahrtmissionen. Mithilfe der freien Computer Vision Bibliothek OpenCV wird der Algorithmus zunächst implementiert. Im Anschluss werden die Teile des Algorithmus, die die meiste Laufzeit benötigen, als verteilte Anwendung mithilfe des Echtzeitbetriebssystems RODOS implementiert.

Abstract

In space applications the importance of imaging and image processing increases steadily. Especially in the context of OOS scenarios vision-based navigation for near-range operations is researched. This thesis is based on such an OOS scenario: a satellite is equipped with a detachable robotic manipulator. Two possible situations may occur: either the satellite or the manipulator break. In both cases it is desired, to remove the manipulator - either to replace it or to reuse it. The manipulator's design includes cylindrical parts. The aim of this master thesis therefore is to design and implement an algorithm, which is able to estimate the 3D pose of a cylindrical object of known dimensions given input from a camera system.

New algorithms for vision-based navigation, such as algorithms for pose estimation are first tested on machines with high computational power. Later they are often implemented on specialized hardware for image processing such as FPGAs or DSPs. Those are specifically programmed for one special purpose, namely the computer vision task, matching the common approach of on board data handling - a specialized processor belonging to an individual sensor or actuator. By contrast, a recent approach proposes the usage of a computing cluster of general purpose processors, which are wirelessly linked. This is currently researched by project YETE of the University of Würzburg.

In order to refrain from the use of specialized hardware, the algorithm is analysed for possibilities of distributed execution. The thesis begins with a introduction to frequently used image processing algorithms in space missions. A reference implementation is created using the open source computer vision library OpenCV. Parts of the algorithm which have been identified as the algorithm's bottlenecks are then distributed using the RODOS real time operating system.

Contents

Acronyms

OOS On Orbit Servicing

FPGA Field Programmable Gate Array

DSP Digital Signal Processor

YETE phYsical distributEd conTrol in spacE

ISS International Space Station

OBDH On Board Data Handling

RGB Red-Green-Blue colour space

HSV Hue-Saturation-Value colour space

GPU Graphics Processing Unit

ROI Region Of Interest

CSN Camera Sensor Network

MER Mars Exploration Rover

DIMES Descent Image Motion Estimation System

SVS Advanced Space Vision System

DARPA Defense Advanced Research Projects Agency

WAC Wide Angle Camera

HRC High Resolution Camera

OEDMS Orbital Express Demonstration Manipulator System

LIDAR Light Detection and Ranging

VERTIGO Visual Estimation and Relative Tracking for Inspection of Generic Objects

NASA National Aeronautics and Space Administration

PanCam Panoramic Camera

RODOS Realtime Onboard Dependable Operating System

PCA Principal Component Analysis

RPC Remote Procedure Call

CORBA Common Object Request Broker Architecture

API Application Programming Interface

Chapter 1

Introduction

1.1 Aim of this Thesis

The importance of computer vision increases in space technology. Sun sensors and star trackers have long been used for attitude determination of spacecraft. Emerging space applications are for example On Orbit Servicing (OOS) and active debris removal, both aiming to reduce the amount of space debris, either by removing it from space or by extending the life span of satellites. An active satellite, here referred to as chaser, has to approach a target in orbit. The target, either a damaged or fuel-less satellite, part of a rocket or natural debris, is mostly non-cooperative, meaning the chaser has to succeed to the task of estimating the target's pose. An example of an OOS scenario, as imagined by Airbus Defence & Space, is seen in figure 1.1.

In the scenario of this thesis the relative pose of a cylindrical object has to be estimated. This object is part of a robotic manipulator arm, which is mounted on the target satellite (see for example the chaser in figure 1.1). To distinguish it from other parts of the manipulator and the target satellite. The cylindrical part is coloured in a distinct shade. The input of the pose estimation consists of a continuous stream of images.

One possibility to classify algorithms for vision-based pose estimation, is the reliance on fiducial markers. A fiducial marker is a known pattern which has to be attached to the target. Pattern based navigation has for example been used for docking to the International Space Station (ISS) (see fig. 1.2), but is in general inconvenient

Fig. 1.1: An OOS scenario as given in this thesis. One of the satellites is equipped with a robotic manipulator. (from `http://www.esa.int`)

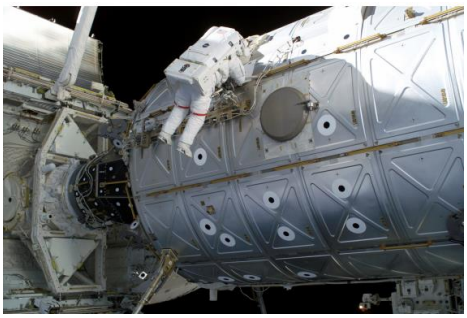


Fig. 1.2: Fiducial markers are attached to the ISS in order to support the rendesvouz operation (from `http://www.nasa.gov`)

because the patterns have to be mounted on the target. Relinquishing fiducial markers however impedes the detection of the object, which pose has to be estimated. Instead of a known pattern, a model of the object or alternatively features are used, which have to be trained beforehand. In almost all vision-based approaches, though, the algorithms are based on the same foundation of image processing methods. This includes colour conversion (for example Red-Green-Blue colour space (RGB) to Hue-Saturation-Value colour space (HSV) or greyscale) and convolution (for sharpening or blurring) as well as more complex algorithms like blob, edge or corner detection.

Image processing methods which are performed on each pixel put themselves forward to be executed in a parallel or distributed fashion. The second idea behind this thesis is to investigate the possibilities of distributing the image processing parts of the pose estimation. The idea concurs with the YETE project of the university of Wüerzburg:

The design of a spacecraft's On Board Data Handling (OBDH) subsystem currently observes the following conservative principle: sensors and actuators are each connected to their respective preprocessors forming subsystems and only a few central processing units connecting those subsystems. All processors have to conform to the power requirements, meaning they have to be able to provide the power needed during peaks, while at the same time being idle most of the time. Besides, the failure of a processing unit implies the loss of a whole sensor or actuator.

Project YETE therefore proposes a new decentralized approach. A distributed cluster of general purpose processors, which are wirelessly linked, is meant to replace the standard OBDH architecture in order to achieve highly reliable operations. This is not only relevant for single spacecraft, but also for satellite formations, where resources from another satellite in the formation might be used.

1.2 Related Work

1.2.1 Computer Vision in Space Applications

One of the early examples of vision-based navigation are planetary rovers. The research for autonomous navigation already started in the 70s, using on-board stereo cameras and scanning laser range-finders as well as off-board computers. In the 1997

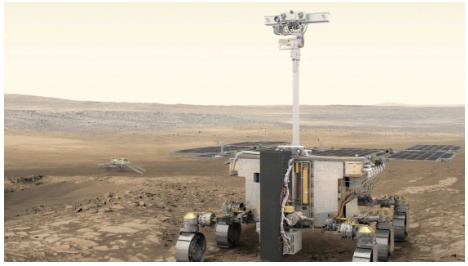


Fig. 1.3: The Rover of the European ExoMars mission (from `http://exploration.esa.int`)

NASA Mars Pathfinder mission, the rover Sojourner was equipped with a multispectral stereo camera pair. The images produced, however, were sent to earth for processing, as Sojourner itself only possessed an on-board computer clocked at 2 MHz. Later, the Mars Exploration Rovers (MERs) Spirit and Opportunity (2004) were equipped with processors with 20 MHz as well as with three sets of stereo camera pairs. Using their input for autonomous obstacle detection, the rovers could traverse at velocities of 1cm/sec.

The lander of the MER mission also was equipped with a camera. In the landing process it executed the Descent Image Motion Estimation System (DIMES) in order to estimate its horizontal velocity. DIMES required three images. The algorithm consisted of image binning (to reduce the image size), the determination of overlapping fields for each pair of images, the selection and tracking of features and the flattening and rectification of the images. The total runtime of this algorithm was 14 sec, in which time the lander traversed more than half the distance to the ground, namely over 1000 m. [1]

A more recent example of planetary rovers with vision based navigation is the European ExoMars mission (see figure 1.3). The ExoMars rover is planned to land on the surface of mars in 2020[1][2]. It is equipped with a Panoramic Camera (PanCam), which consists of two stereo Wide Angle Cameras (WACs) and a High Resolution Camera (HRC)[3].

In 2005 the interest in computer vision in space applications has led to the organization of a workshop accompanying the International Conference on Intelligent Robots and Systems, which then took place in Alberta, Canada. [4]

The Canadians also were also responsible for the Advanced Space Vision System (SVS), which has been designed primarily for the assembly of the International Space Station (ISS). It is dependent on fiducial markers which had to be placed on the ISS. The system was first tested on Spaceshuttle STS-52 (1992). Later on for docking and rendezvous the TriDAR 3D system has been employed on Spaceshuttles STS-128 and STS-129. Instead of fiducial markers this system involves an active 3D sensor, a thermal imager and model-based tracking software. [5]

The first usage of computer vision in a on orbit rendezvous scenario has been demonstrated by the Orbital Express mission by the Defense Advanced Research

[1]The launch has been delayed several times (among others the launch was scheduled in 2013, 2018, ...).

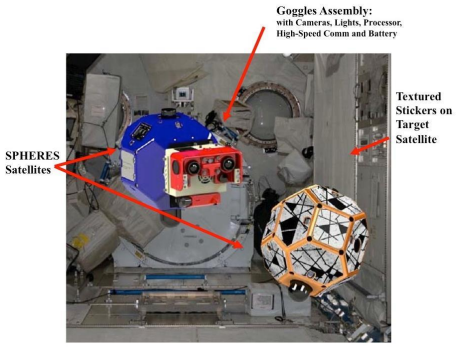


Fig. 1.4: The spherical Sphere Satellites with the VERTIGO goggles attached (from `https://www.nasa.gov/`)

Projects Agency (DARPA). The Orbital Express mission consisted of the chaser satellite ASTRO which was equipped with a robotic arm. The robotic arm was tasked with grappling the target satellite NEXTsat, using the on board Orbital Express Demonstration Manipulator System (OEDMS) vision system. This vision system consists of a camera, a frame grabber and a pose estimation algorithm. [6]

A more recent example represents the Japanese Hayabusa mission. The Hayabusa was tasked with approaching and touching down on a Asteroid and returning to earth with samples. The rendezvous and touch town process was supported by an optical navigation camera, a Light Detection and Ranging (LIDAR) sensor, laser range finders and fan beam sensors [7].

In order to test algorithms for control of small satellite, on board of the ISS the Spheres satellites are available. They have been extended in 2012 with the Visual Estimation and Relative Tracking for Inspection of Generic Objects (VERTIGO) goggles enabling scientists to test new approaches in vision based mapping and localization (see figure 1.4). The goggles come with a 1.2 GHz Via Nano processor and 128 GB of flash storage, making them much more capable as the devices targetted in this thesis. The images captured by the goggles can be received on a additional laptop as a stream of OpenCV images [8].

1.2.2 Pose Estimation in Spacecraft Proximity Operations

A lot of recent research deals with pose estimation in space scenarios. Most of the publications deal with the pose estimation of target spacecraft for autonomous docking operations.

Lichter and Dubowsky[9] propose the usage of a Kalman filter to estimate the pose of an unknown space object based on range images. As surrogate measurement for the filter the principal geometrical axes of the object are computed after the range images of different range sensors such as laser scanners and stereo cameras have been kinematically fused. The processing of the range images is not explained further. To estimate the orientation expressed as the objects principal inertial axes and the translation separate Kalman filters are utilized. The space object is assumed to be rigid and not affected by any external torque. Synthetically created images were used

for testing, no explicit studies were made regarding the computational requirements of the algorithm.

Kelsey et al.[10] present a model-based pose refinement algorithm for vision based relative pose estimation and tracking. Model-based means, a priori knowledge about the object in the form of features or a CAD model is used. Their algorithm consists of three steps. At first an estimation of the pose is obtained e.g. by active appearance models, robust Principal Component Analysis (PCA) or simultaneous pose and correspondence determination. Then the model is rendered at the estimated pose and a iterative reweighted least squares approach is used to refine the pose by minimizing the error between detected edges in the image and the edges of the rendered model. At last an extended Kalman filter is used for pose tracking using a known motion model. The rendering of the model is a computationally intense task and best achieved using Graphics Processing Units (GPUs).

Du et al.[11] use knowledge about the satellite's architecture to estimate its pose. Their algorithm is refined for large communication satellites, which typically have rectangular shaped antennae. As the field of view of a monocular camera is insufficient to capture the complete structure, two collaborative monocular cameras are employed. Each of the images is filtered to reduce noise, Canny edge detection then produces edge images where lines are extracted using Hough transform. As a next step the sides of the rectangles are determined. As a lot of lines are contained in the image due to its complex scene, reference points have to be chosen manually. As a result this algorithm is not totally suited for autonomous satellites. The junctions of the rectangle sides are extracted as the rectangle vertices. The information of both cameras is then fused and fed to the pose estimation algorithm. The algorithm was only tested with synthetic images, assuming different levels of noise.

The scenario is similar in the paper of Hu et al.[12], also using two cameras which have an overlapping field of view, though. Not a rectangular structure, but a thruster nozzle is used as known architectural feature. However the algorithm follows the same steps, after a Canny edge detection a circular Hough transform is performed to detect circular features. The circular feature with the greatest radius is extracted in both images and an ellipse is fit to the circles. The origins of the ellipses are then used to calculate the pose of the thruster nozzle by triangulation.

Another model-based approach is presented by Liu and Hu[13], which is especially interesting as it deals with solid-of-revolutions shaped spacecraft[2]. The algorithm is one of the few, which also were tested with non-synthetic images. Their method to extract ellipses from an edge image is described in [14]. At first they use image processing methods to decline all edge segments which are not ellipses. Than ellipses are fit to the remaining arc segments at first using a direct least square method (direct LS). If the average distance of the points of the arc segment to this ellipse is higher than a threshold the RANSAC algorithm is applied iteratively to find a better matching ellipse. Arc segments belonging to the same ellipse are then merged. After the ellipses are extracted a CAD model is used to find the meaningful ellipses and match them to arcs of the model.

[2] A cylinder is also a solid of revolution

1.2.3 Distributed Applications

The term *distributed* computing is sometimes confused with *parallel* computing, as parallel computing may mean the distribution of threads to processing cores in order to be processed in parallel (those cores then however belong to the same machine). In a large scale distributed applications may be executed on a single machine with multiple cores, by using a multithreading approach where each originally distributed part of the application is succeeded by one thread respectively. In general distribution refers to architectures, consisting of more than one processing unit, where tasks are divided between those units.

The idea of clusters of general purpose processors in Space Technology was researched during project YETE of University of Würzburg. The idea behind is to make On Board Data Handling (OBDH) more reliable, as a failing processing unit does not imply the loss of a sensor or actuator, as is the case in the conservative OBDH approach [15].

This concept also conforms to the idea of networked satellite formations, as satellites of one formation might share their computational capabilities [16].

In computer vision, distributed algorithms are often driven by the usage of Camera Sensor Network (CSN), which is a set of resource-constrained camera-equipped sensor nodes that can communicate over a wireless network [17]. They are for example used in video surveillance tasks. One of the challenges is, that existing computer vision algorithms assume centralized computation. CSNs do not totally conform to the YETE idea, as each camera of the network is equipped with a processing unit (together forming a smart cameras). Besides, those networks do usually not consist of only a few cameras, but of hundreds to thousands to millions[3].

In [18] several distributed algorithms for Computer Vision are mentioned. However, not all of them are tightly connected to Computer Vision but more to the problems arising with the use of many cameras, such as topology estimation where the vision graph of the network has to be determined.[4].

1.3 Approach

There are several tasks involved in this thesis. The first part is merely an algorithmic problem. As explained previously, the pose of a cylindrical object has to be estimated. However, a cylinder is mostly a feature-poor object, especially when its circular sides are occluded. Therefore, feature based object detection algorithms do not succeed to the task of recognizing a cylinder, if only the parallel lines of its body are seen. A line itself is also a feature which can be extracted after an edge detection algorithm was applied. Additionally, the relation between a 3D object and its 2D projection is known and can be exploited, if certain properties are known. This problem is also known as the *inverse perspective problem* and solving the problem for a cylinder with known radius is the first task of the thesis.

[3] Imagine for example a traffic surveillance system.

[4] The vision graph describes which cameras have an overlapping field of view

Before any lines can be extracted the input image has to be processed. As the colour of the cylinder is previously known, image segmentation based on blob detection is employable. This allows to reduce the input of the edge detection algorithm to the rectangle around the detected blob. After the edges have been detected, they are transformed to lines. Therefore at first connected edge pixels are resolved to chains, which are then simplified to either match a straight line or an arc (of the cylinder side). The image moments of the detected blob also are used to select the correct lines corresponding to the sides of the cylinder.

Both parts, the image processing and the estimation of the cylinder, are combined. The estimation may serve as input to a filter, allowing to refine the estimated pose of the cylinder. The application is implemented using the freely available computer vision library OpenCV. It is tested on different platforms, including a standard laptop and also a UDOO QUAD single board computer. It has a quad-core ARM Cortex-A9 processor and is able to run Linux.

The application is furthermore analysed in order to determine run time bottle necks. These application parts are going to be distributed. In order to abstract from hardware, RODOS is used as distribution middleware. RODOS is a minimalistic real-time operating system, which supports many processors. RODOS facilitates communication between distributed devices, as it provides implementation of standard communication channels (such as UDP, USART, I2C).

1.4 Outline

Chapter 1 contained the introduction of the topic and classified the problem in the scientific world.

Chapter 2 introduces the fundamental knowledge needed to know in order to understand the main part of the thesis. It includes an explanation of the projective geometry used to solve the inverse perspective problem of the cylinder. Additionally it provides an overview on the vast topic of computer vision and explains some problems to be encountered in distributed applications.

Chapter 3 then explains the pose estimation algorithm. It includes a description of the scenario and explains the inverse perspective problem of the cylinder.

Chapter 4 gives an introduction to the software tools, which were used to implement the application and to create the synthetic test images.

Chapter 5 describes the implementations of the algorithm and their testing. The tests are divided into different categories: the pose estimation algorithm itself is tested using synthetic video material, where the pose of the cylinder at each point in time is exactly known. Also, images were recorded of a cylindrical object (the part of the arm), using a robotic manipulator (which bears the camera). The rate, at which the application is able to estimate the pose, is measured for different levels of distribution.

Chapter 6 concludes this thesis and evaluates its contribution.

Chapter 2

Fundamentals

2.1 Projective Geometry

2.1.1 2D, 3D and homogeneous coordinates

The following subsections are a short summary of the chapter *Geometric primitives and transformation* in the book of Szeliski(2011) [19].

Projective geometry is used to describe the transformation between a point in the 3D world coordinate system Σ_W into the 2D image coordinate system Σ_I. In general, a coordinate system of a n-dimensional vector space consists of n linearly independent vectors (the axes) and a position vector giving the origin $\mathbf{O}$. In 3D the three axes are denoted as the unit vectors $\hat{\mathbf{e}}_x$, $\hat{\mathbf{e}}_y$ and $\hat{\mathbf{e}}_z$. A point in a three dimensional coordinate frame is therefore either described as a combination of the unit vector $\mathbf{v} = x \cdot \hat{\mathbf{e}}_x + y \cdot \hat{\mathbf{e}}_y + z \cdot \hat{\mathbf{e}}_y$ with $x, y, z \in \mathbb{R}$ or directly as a vector $\mathbf{v} = \begin{pmatrix} x & y & z \end{pmatrix}$. The image coordinate system Σ_I originates in the image center $\mathbf{c} = \begin{pmatrix} c_x & c_y & f \end{pmatrix}$, where f is the focal length of the camera, the x-axis is directed to the right, the y-axis is directed downwards. Lengths in Σ_I are measured in pixel units.

2D coordinates are also described using *homogeneous coordinates* $\tilde{\mathbf{x}} = \begin{pmatrix} \tilde{x} & \tilde{y} & \tilde{z} \end{pmatrix} \in \mathbb{P}^2$, where $\mathbb{P}^2 = \mathbb{R}^3 - \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$ is the 2D projective space. Homogeneous coordinates are only defined up to a scale. The conversion between a homogeneous vector $\tilde{\mathbf{x}}$ back into a 2D point $\mathbf{x}$ is done as follows:

$$\tilde{\mathbf{x}} = \begin{pmatrix} \tilde{x} & \tilde{y} & \tilde{z} \end{pmatrix} = \tilde{z} \begin{pmatrix} x & y & 1 \end{pmatrix} = \tilde{z}\bar{\mathbf{x}} \tag{2.1}$$

The vector $\bar{\mathbf{x}}$ is called the *augmented vector*. The planes in $\mathbb{P}^2$ defined as the set of points, which share the same third element (e.g. $z = 1$) are referred to as projective or perspective planes.

Other than points, also lines can be represented as homogeneous coordinates $\tilde{\mathbf{l}} = \begin{pmatrix} a & b & c \end{pmatrix}$. The general line equation is then

$$\bar{\mathbf{x}}\tilde{\mathbf{l}} = ax + by + c = 0 \tag{2.2}$$

.

There are more forms of line representation:

- High school students are more familiar with the line equation $y = mx + b$, where m is the slope of the line and b is the intersection with the y-axis. This

representation cannot describe vertical lines.

- A line is also described by the normal $\hat{\mathbf{n}}$ perpendicular to the line and its distance d to the origin. Instead of using the vector $\hat{\mathbf{n}}$, also the angle θ between $\hat{\mathbf{n}}$ and the x-axis can be used to represent the line as $\begin{pmatrix}\theta & d\end{pmatrix}$. The relation between both forms is given as, $a = \cos\theta$, $b = \sin\theta$ and $c = d$.

- Another possibility is to use a position vector $\mathbf{p} = \begin{pmatrix}c & d\end{pmatrix}^T$ and a normalized direction vector $\mathbf{q} = \begin{pmatrix}g & h\end{pmatrix}^T$. The direction vector is related to the normal vector by $\hat{\mathbf{n}} = \begin{pmatrix}-h & g\end{pmatrix}$ and c may then be calculated by inserting point $\mathbf{p}$ into the general equation. This representation is further referred to as point representation.

The intersection between two lines using homogeneous coordinates is calculated using the crossproduct:

$$\tilde{\mathbf{x}} = \tilde{\mathbf{l_1}} \times \tilde{\mathbf{l_2}} \tag{2.3}$$

In 3D the equivalent of lines are planes. So, using homogeneous coordinates of $\mathbb{P}^3$ the equation

$$\bar{\mathbf{x}}\tilde{\mathbf{m}} = ax + by + cz + d = 0 \tag{2.4}$$

with $\tilde{\mathbf{m}} = \begin{pmatrix}a & b & c & d\end{pmatrix}$ describes a plane rather than a line. To describe 3D lines, two points on the lines $\begin{pmatrix}\mathbf{p} & \mathbf{q}\end{pmatrix}$ are needed. The points $\mathbf{x}$ on the line are then described by

$$\mathbf{x} = \mathbf{p} + \lambda(\mathbf{q} - \mathbf{p}) \tag{2.5}$$

The vector $\mathbf{q}-\mathbf{p}$ is a direction vector and may be normalized. The resulting direction vector $\mathbf{b}$ is therefore

$$\mathbf{b} = \frac{\mathbf{q} - \mathbf{p}}{\|\mathbf{q} - \mathbf{p}\|} = \begin{pmatrix}b_1\\ b_2\\ b_3\end{pmatrix} \tag{2.6}$$

where the vector elements are called the *direction cosines* of the lines. Lines, which share the same direction cosines, are parallel.

2.1.2 Coordinate Transforms

The sets of two-dimensional and three-dimensional coordinate transformation are very similar. A coordinate transformation converts coordinates of one coordinate frame into coordinates of another coordinate frame. The coordinates frames are related to each other by a transformation matrix. Expressed in homogeneous coordinates, the matrix of a 3D transformation has size 4×4. The five coordinate transformations as displayed in figure 2.1 can be ordered hierarchically. The transformations of lower order preserve more properties than the transformations of higher order.

The highest order transform is the projective transformation, also known as *3D perspective transform*. The transformation matrix $\tilde{\mathbf{H}}$ is an arbitrary 4×4 homogeneous matrix. The projective transform only preserves straight lines, meaning that a straight line remains straight after the transformation.

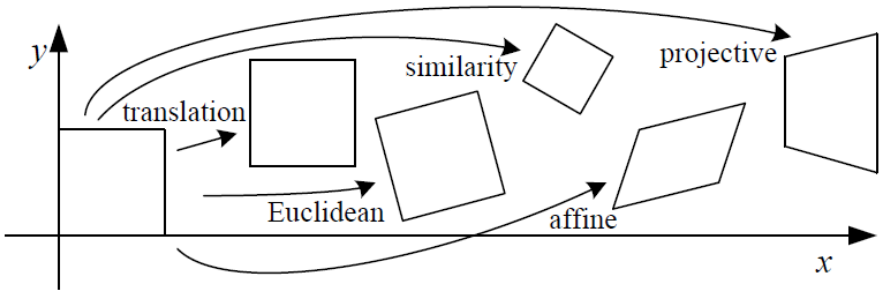


Fig. 2.1: Basic set of 2D planar transformations, from [19]

The affine transformation also preserves parallelism. It is a 3×4 arbitrary matrix $\mathbf{A}$ applied to non-homogeneous coordinates. Similarity transformations additionally preserve the angles between lines and planes. Its transformation matrix is build as a matrix

$$\mathbf{S} = \begin{bmatrix} s\mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \end{bmatrix} \tag{2.7}$$

where s is a scaling factor, $\mathbf{R}$ is a rotation matrix and $\mathbf{t}$ is a translation vector. In a rigid or Euclidean transformation, the scaling factor is not used ($s = 1$) and lengths are preserved, the matrix therefore is

$$\mathbf{E} = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \end{bmatrix} \tag{2.8}$$

The most simple transformation is a translation, where the rotation matrix is a unit matrix $\mathbf{E}$.

$$\mathbf{T} = \begin{bmatrix} \mathbf{E}_{3\times3} & \mathbf{t}_{3\times1} \end{bmatrix} \tag{2.9}$$

2.1.3 Transformation between World and Image space

The transformation between coordinates of the world coordinate system and the image coordinate system can also be described using a perspective transformation matrix. The rotation matrix and translation vector between the coordinate system of the camera-centred coordinate system Σ_C and Σ_W are the extrinsic parameters of the camera. The so called calibration matrix $\mathbf{K}$ depends on the intrinsic parameter of the used camera and maps points of Σ_C to 2D pixel coordinates. By calibration, $\mathbf{K}$ can be found as an upper-triangular form:

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Here, f_x and f_y are the independent focal lengths for the sensor x and y dimensions respectively. The optical center expressed in pixel coordinates is $\begin{pmatrix} c_x & c_y \end{pmatrix}$ and s denotes any *skew*, which appears if the sensor axes are not mounted perfectly perpendicular to the optical axis. For many applications, it is sufficient, to assume a common focal length f, no skew and to set the origin at the center of the image.

The camera matrix $\tilde{\mathbf{P}}$ is now obtained by putting together the extrinsic and intrinsic camera parameters:

$$\tilde{\mathbf{P}} = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{\mathbf{K}}\mathbf{E} \tag{2.11}$$

where $\mathbf{E}$ is a Euclidean transformation and $\tilde{\mathbf{K}}$ is the full-rank calibration matrix. A point $\bar{\mathbf{p}}_w = \begin{pmatrix} x_w & y_w & z_w & 1 \end{pmatrix}$ is mapped to pixel coordinates $\mathbf{x}_i = \begin{pmatrix} x_i & y_i & 1 & d \end{pmatrix}$ with disparity d as

$$\mathbf{x}_i \sim \tilde{\mathbf{P}}\bar{\mathbf{p}}_w \tag{2.12}$$

The resulting vector has to be divided by its *third* element to obtain the normalized form $\mathbf{x}_i$. Using this transformation model means that effects like radial distortion are neglected. They are not included in this thesis, as it is assumed that the calibration and undistorting of images is done before the images are fed to the algorithm.

2.1.4 Inverse projective geometry

To reconstruct 3D information from a 2D image, the problem of inverse projection has to be solved. Usually this is not possible if only one image is available. One possible method to reconstruct depth information is stereo vision. The same scene is photographed with two cameras with known transformation. Then the respective pixels or features on each image are matched to calculate a disparity. As the distance to the camera is proportional to this disparity, a depth value for this pixel position can be deduced.

In monocular vision, with only one camera is involved, the knowledge of the observed objects can be used, to reconstruct information. Possible properties are parallel lines, collinear points with known inter-point distances or intersecting lines in a known plane. A analysis of the relations between the 2D projection and the 3D description of primitives such as points and lines, is given by Haralick(1989) [20]. The relationship between the parameters of the two-point representation of a 3D line and the point representation of a 2D line for example is found by inserting the line equation for the 3D point into the 2D line equation.

Assuming $\mathbf{c} = \begin{pmatrix} c_x & c_y \end{pmatrix} = \begin{pmatrix} 0 & 0 \end{pmatrix}$, the pixel coordinates $\mathbf{p} = \begin{pmatrix} u & v \end{pmatrix}$ are calculated as

$$u = f\frac{x}{z} \quad v = f\frac{y}{z} \tag{2.13}$$

If the 3D line L is given as the set

$$L = \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \middle| \text{for some } \lambda, \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \lambda \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \right\} \tag{2.14}$$

then the following relationship between the unknown parameters $a_1, a_2, a_3, b_1, b_2, b_3$ and the observed parameters c, d, g, h of the 2D line is found:

$$h \cdot f \cdot a_1 - g \cdot f \cdot a_2 + (d \cdot g - c \cdot h))a_3 = 0 \tag{2.15}$$

$$h \cdot f \cdot b_1 - g \cdot f \cdot b_2 + (d \cdot g - c \cdot h))b_3 = 0 \tag{2.16}$$

Using this relationship, if the projections of two parallel lines are observed, the direction cosines can be calculated as:

$$\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \frac{\begin{pmatrix} h_1 f \\ -g_1 f \\ d_1 g_1 - c_1 h_1 \end{pmatrix} \times \begin{pmatrix} h_2 f \\ -g_2 f \\ d_2 g_2 - c_2 h_2 \end{pmatrix}}{\left\| \begin{pmatrix} h_1 f \\ -g_1 f \\ d_1 g_1 - c_1 h_1 \end{pmatrix} \times \begin{pmatrix} h_2 f \\ -g_2 f \\ d_2 g_2 - c_2 h_2 \end{pmatrix} \right\|} \tag{2.17}$$

where c_1, d_1, g_1, h_1 are the parameters of the first 2D line and c_2, d_2, g_2, h_2 are the parameters of the second line.

2.2 Computer Vision

2.2.1 Image Processing or Computer Vision?

The terms *Image Processing* and *Computer Vision* are often mixed up. Computer vision describes a field of research and applications, which aims to reconstruct the properties of a scene and its objects depicted in one or more images. It uses methods of image processing and image analysis, e.g. to separate background and foreground of a scene, to identify objects based on known features or to reconstruct three dimensional information based on two dimensional inputs. The block diagram in figure 2.2 relates 2D images to properties considering shape and appearance. The number in the boxes refer to the chapters in the book by Szeliski [19], which is also recommended as source for further reading.

Generally, image processing refers to the transformation of the image itself. The output of an image processing algorithm is usually an image again, but with changed properties. Examples of image processing are colour conversions(e.g. the transform of a colour image into a grey scale image), geometric transformations as scaling or rotations, and also filtering. In contrast to image processing, image analysis methods for segmentation, feature detection or motion estimation do not output a transformed image, but rather abstract information. If the information is afunction of each pixel, the result might also be represented as an image. For example, a segmentation, which separates the foreground and background of an image, might produce a binary image, where 0 is assigned to each background pixel and 1 is assigned to each foreground pixel.

2.2.2 Digital Images

A digital image may be represented by a pixel matrix, where the term pixel is abbreviated for *picture element*. It is defined by the space $[0, W) \times [0, H) \subseteq \mathbb{N}^2$, where W is the image width and H is the image height. The origin is located in the top left corner. The coordinate x_i refers to the respective column, y_i refers to the row. This means pixel $p = \begin{pmatrix} 5 & 12 \end{pmatrix}$ is placed in the sixth column and 13th row of the

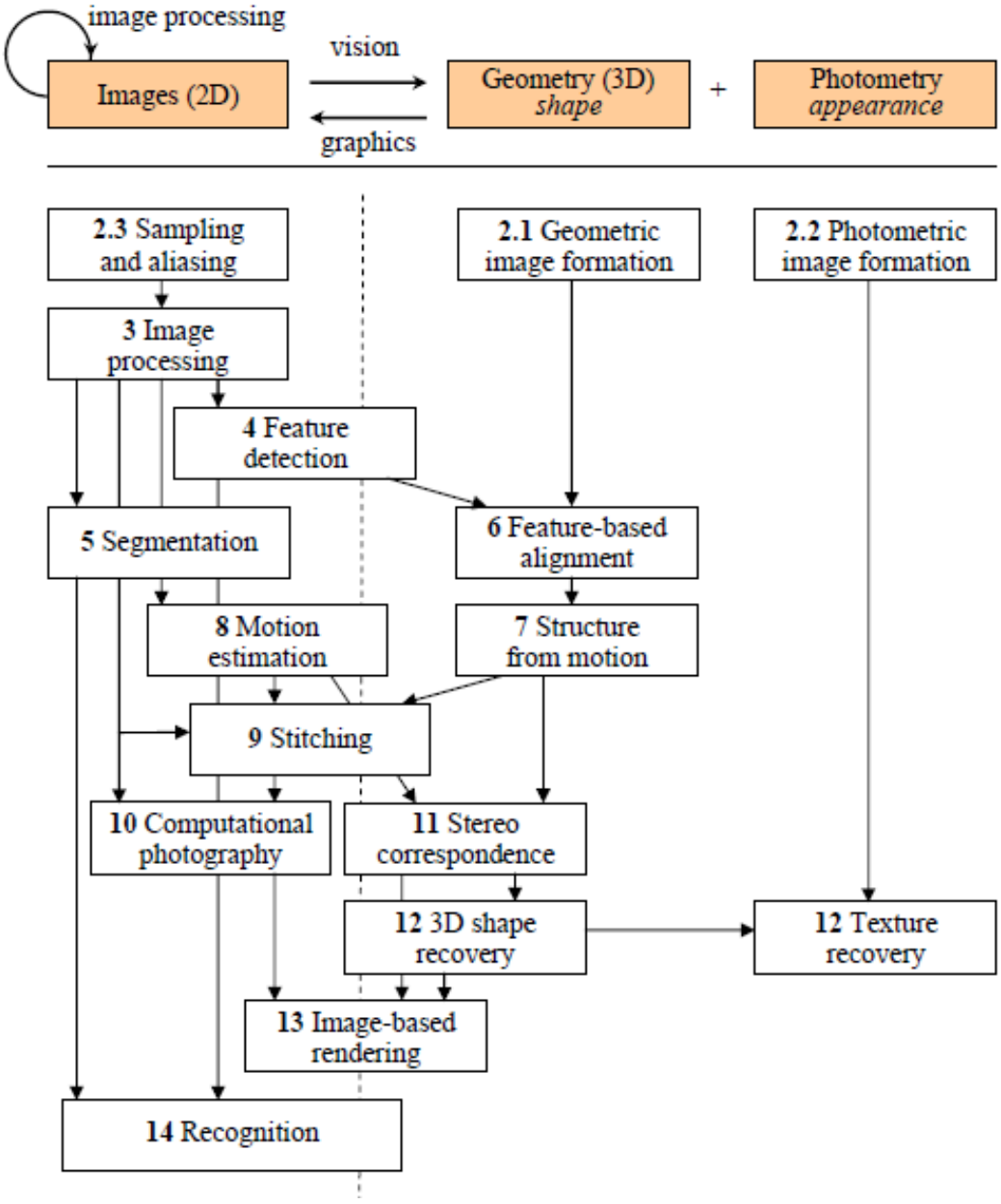


Fig. 2.2: A block diagram of the relationship between images, geometry, and photometry, (from [19])

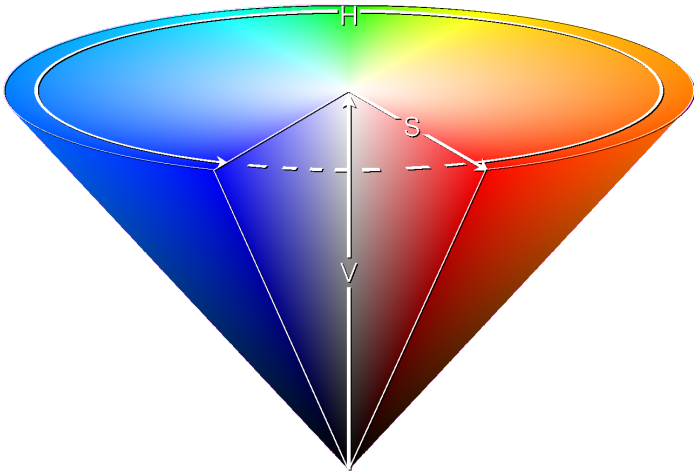


Fig. 2.3: Hue H, Saturation S and Value V (from (3ucky(3all - CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=943857`)

image. The function $I : [0, W) \times [0, H) \to \mathbb{K}$ where $\mathbb{K}$ is a set of numbers, maps a value to each element of the image matrix. If the image encodes intensity values, then $\mathbb{K} = [0, 1]$ may be chosen, where 0 refers to the darkest intensity and 1 refers to the lightest intensity. In practice, images, which represent visual properties, mostly contain integer values. For example, a gray scale image might contain intensity values between 0 and 255. Images which encode other information - such as distances or heat, may contain floating values.

Colour images are multi-channel images. This implies, that each pixel corresponds to multiple values. $\mathbb{K}$ may then be defined as a vector space. In RGB images three channels are used. The RGB colour space is often visualized as a unit cube of a coordinate frame, where red, green and blue make up the axes. As RGB colours are awkwardly to use by intuition, the Hue-Saturation-Value colour space (HSV) as visualized in figure 2.3 is sometimes more useful. The colour itself is defined by the *hue*, whereas saturation and brightness (value) are defined separately. In computer vision applications, it is therefore easier to segment colours by hue.

2.2.3 Operators in Image Processing

In image processing operations, pixels are mapped to different values. Image transforms that manipulate each pixel independently of its neighbours are often called point operators. These operators affect brightness and contrast or are used to correct or convert colours. On discrete images a pixel transform is described as $g(i, j) = h(f(i, j))$, where $x = (i, j)$ is the pixel position. The function $f(i, j)$ reveals the image value at that position and h represents the function which is used for the transformation. A common transformation function is the multiplication and addition with

a constant, which is expressed as:

$$g(x) = a \cdot f(x) + b \tag{2.18}$$

where $a > 0$ is called the gain and b is called the bias parameter, which both may also be spatially variable. Increasing the gain changes the contrast of the image. Increasing or decreasing the bias influences the brightness.

Image transforms where pixel manipulations depend on their neighbour pixels are neighbourhood operators or local operators. Two well known local operators are correlation and convolution which are both linear filters and very similar in their definition. Correlation is denoted as $g = f \otimes h$ and defined as

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l) \tag{2.19}$$

where $h(k,l)$ is the kernel or mask and the indices k, l depend on the kernel size. In convolutions the sign of the offsets in f is reversed:

$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) \tag{2.20}$$

It is denoted as $g = f * h$. Filters are often used to blur (smooth) or sharpen images, but are also preprocessing steps in order to find interesting image points such as corners or before edge extraction. A few examples of linear filters are given below:

- Easiest to implement is the moving average or box filter, which averages the pixel in a window of size $K \times K$. If $K = 3$ the filter kernel looks like

$$\frac{1}{9}\begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix} \tag{2.21}$$

- A Gaussian filter kernel is computed as

$$G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.22}$$

- The Laplacian Operator is defined as the second derivative of a two dimensional image

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 y}{\partial y^2} \tag{2.23}$$

- The combined Laplacian of Gaussian (LoG) filter is the equivalent of first applying a Gaussian to an image and then taking its Laplacian

$$\nabla^2 G(x,y,\sigma) = \left(\frac{x^2+y^2}{\sigma^4} - \frac{2}{\sigma^2}\right) G(x,y,\sigma) \tag{2.24}$$

There are also non-linear filters, meaning their response to a sum of two signals is

not the same as the sum of the individual responses ([19]). An example of a non-linear filter is the median filter, which chooses the median of the neighbourhood as new value for the pixel (i, j).

Binary images are commonly processed with non-linear morphological operations, morphological as they change the shape of the binary objects. The conversion of intensity images to binary images, where only two values are present, may be obtained by a thresholding operation:

$$\theta(f,t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else} \end{cases} \tag{2.25}$$

The following morphological operations using a thresholding operator, a 3×3 structuring element s, the integer-valued count of active pixel $c = f \otimes s$ and the total number of pixels S in the considered window are commonly used. :

- $dilate(f, s) = \theta(c, 1)$
- $erode(f, s) = \theta(c, S)$
- $majority(f, s) = \theta(c, S)$
- $open(f, s) = dilate(erode(f, s), s)$
- $close(f, s) = erode(dilate(f, s), s)$

Morphological operations are helpful to remove unwanted artefacts, for example very short edge segments after edge enhancement.

2.2.4 Edge Detection

Edges in an images often correspond to contours of an object or borders between different regions in an image. Therefore edges are usually used in order to find objects or to segment an image. As an edge pixel represents a discontinuity in an image, it is distinguishable from non-edge pixels. The methods, which exert this tasks, belong to the field of edge detection or edge enhancement. The result of an edge detector is often an intensity image, where high intensities represent sharp edges, or a binary image, where each edge pixel is white. The edge image is sometimes also referred to as edge map.

An edge according to the *Dictionary of Computer Vision and Image Processing* is 'a sharp variation of the gradient function. Represented by its position, the magnitude of the intensity gradient, and the direction of the maximum intensity variation'[21]. A definition, which is less mathematical, is given by the book *Algorithms for Image Processing and Computer Vision*: 'An edge is the boundary between an object and the background, and indicates the boundary between overlapping objects'[22]. In the German Book *Computer Vision* the objects are generalized to 'homogeneous [meaning all pixel have similar values] regions'[23]. The book *Concise Computer Vision* is referring to a *step edge model*, which defines edges by 'changes in local derivatives'[24].

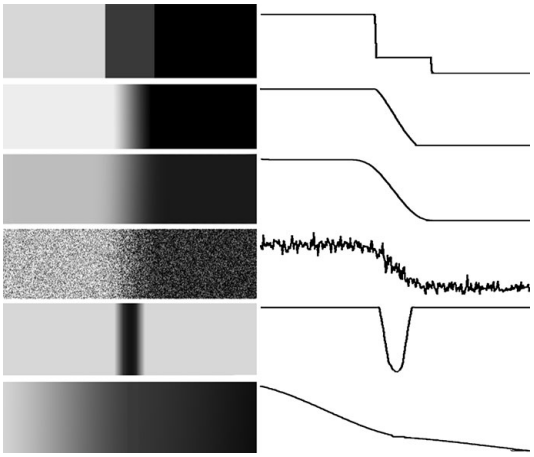


Fig. 2.4: Different forms of edges found in intensity images. On the left are synthetic input images, on the right are the corresponding intensity profiles, from [24]

Figure 2.4 out of the same book shows, that edges appear in various forms. The ideal step edge is a discrete border between regions of different intensities. In reality, ideal step edges seldom occur. One reason is the discretization of the image: an edge does not always appear at the margin of a pixel. More often, the intensity values on the line perpendicular to the edge build a ramp function, or due to discretisation rather a step function. Therefore edge detection methods which are based on gradient functions lead to multiple responses by a single edge.

The gradient or first derivative of an image at pixel position x, y is defined as

$$\nabla I = \mathbf{grad} I = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]^T \tag{2.26}$$

The magnitude of the gradient is the norm of the gradient vector, the direction is given as

$$G_{dir} = \arctan 2(\frac{\partial I}{\partial y}, \frac{\partial I}{\partial x}) \tag{2.27}$$

The first derivatives in x and y direction can be approximated by

$$I_y(x, y) = \frac{I(x, y+1) - I(x, y-1)}{2} \tag{2.28}$$

and

$$I_x(x, y) = \frac{I(x+1, y) - I(x-1, y)}{2} \tag{2.29}$$

The corresponding filter kernels which have to be applied to the image, to obtain the gradient in each pixel is

$$\frac{1}{2}\begin{vmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{vmatrix} \tag{2.30}$$

for the horizontal gradient (in x direction) as well as

$$\frac{1}{9}\begin{vmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} \tag{2.31}$$

for the vertical gradient (in y direction). Another approximation of the gradient is given by Sobel[25]. He used the filter kernels

$$\begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix} \tag{2.32}$$

and

$$\begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix} \tag{2.33}$$

Applying either of these filters to the image yields a gradient magnitude map, where the maxima are possible edge positions. They are therefore basic edge detectors. However, neither of them is optimal. The criteria of an optimal edge detection was established by Canny, who invented the *Canny Edge Detector*, which today is a popular edge detection method.[26]. He summarized the criteria as:

- edges that are in the images should not be missed
- edge points should be well localized
- no multiple responses to a single edge

The Canny algorithm is applied to a (Gaussian) smoothed image. The first derivative of a Gaussian is then used to obtain a gradient image. The first derivative of a Gaussian was chosen accordingly to the criteria formulated as mathematical constraints. The next step is the *non maxima suppression*. At each position, the gradient magnitude is compared to the gradients of the positions given by the gradient directions. This means: if the direction is pointing downwards or upwards, the magnitude has to be greater than the gradient magnitude of pixel above as well as the pixel below. If the direction is to the left or right, the magnitude has to be greater than the magnitudes of pixels left and right, otherwise the magnitude at this position is discarded. The same applies for diagonal directions. As there are only 8 possible directions, the calculated gradients are rounded to multiples of $\pi/4$.

Hysteresis thresholding is the last step of the Canny algorithm. Two thresholds are chosen, a pixel counts as an edge pixel if the magnitude is higher than a threshold T_{high}. A pixel in its neighbourhood, which magnitude is lower than this threshold, only counts as an edge pixel, if its magnitude is still higher than a second threshold T_{low}.

2.3 Distributed Algorithms

As their name suggests, distributed algorithms are designed to be executed on a distributed system. Such a system consists of a network of different computing nodes which usually do not share common memory but communicate using messages [27]. Often the computing nodes have varying capabilities and therefore are responsible for different kinds of actions.

Certain problems accompany distribution: a distributed system needs to be fault tolerant, as messages between nodes might get erroneous or even lost. The failure of one or more computing nodes has to be handled as well, e.g. through the introduction of redundancies and the reasonable redistribution of tasks. Is one single computing node responsible for an important task, its availability has to be monitored: does it provide enough computing power, does the communication impose a severe bottle neck? Another problem is load division, which describes the decision of how to distribute loads to possibly heterogeneous nodes.

At the same time each problem turns into an advantage, if handled well. Distributed systems are more scalable, fault tolerant and offer greater concurrency than a centralized system.

There are multiple motivations to design or use a distributed system and therefore also distributed algorithms. With the rise of the world wide web as a giant distributed system, client-server applications have become more common. A internet user may access a service through a web interface, which communicates with a server, which may then again communicate with another server, which may have access to a necessary database or offer enough computational power to perform complex calculations (e.g. cloud computing). The motivation in this thesis, though, is to exploit a distributed system in order to parallel the task of an algorithm in order to improve its execution rate.

Of the distributed system to be used, only the available computing nodes are known. The architecture and communication method has to be chosen carefully and requires an introduction into the available possibilities. This introduction is given in the next sections.

2.3.1 Common Architectures for Distributed Systems

A distributed architecture describes how tasks are distributed to the computational nodes and which nodes control this distribution. In web applications the client-server model is widely used. A simple client-server architecture consists of a single server and multiple clients. The server offers a service, which can be requested by the clients. The number of clients is not fixed and the server does not need to know the client. Only the address of the server has to be public or known to the clients. A client-server model may be extended, for example if the server itself requests functionality from another server.

In systems, where the number of servers and their services are not known or the system is about to be extended in the future, a broker architecture may be used. In this architectural style, server and clients do not communicate directly, but through

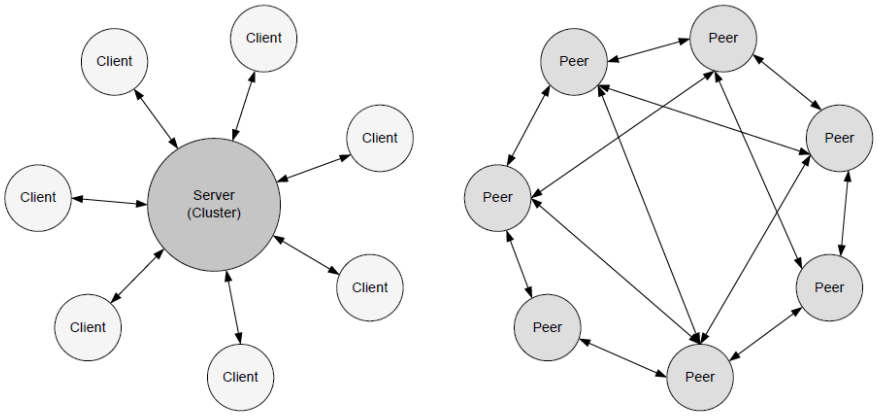


Fig. 2.5: Client-Server versus Peer-to-Peer Architecture, from [27]

a kind of *middle man*, the broker. Its task is to coordinate communication and to forward the requests, sent by clients, to the correct service. An international standard for a broker implementation is given by Common Object Request Broker Architecture (CORBA).

The third architecture to be presented here, is the peer-to-peer architecture. In contrast to both of the other architecture, the roles of peers are not hierarchically fixed. Every peer may use a service by or offer a service to the other peers in the network. This may involve more complex communication channels, as peers communicate with each other, as shown in figure 2.5. In order to find other peers in the system, indexes provided by a central node may be used or flooded-requests algorithms, which distribute service requests by broadcast to their neighbours until a matching peer is found.

2.3.2 Communication Methods

Two similar communication methods are Remote Procedure Call (RPC) and *message-based communication*. RPC is a basic mechanism for the client-server model. The term was introduced by Bruce Jay Nelson in 1981. The process is explained in a paper by Nelson [28] as follows:

> When a remote procedure is invoked, the calling environment is suspended, the parameters are passed across the network to the environment where the procedure is to execute (which we will refer to as the callee), and the desired procedure is executed there. When the procedure finishes and produces its results, the results are passed backed to the calling environment, where execution resumes as if returning from a simple single-machine call.

Advancements of RPC, called asynchronous RPC, exist which do not block the calling environment, while the method is executed but only as long until the request is accepted by the callee. A problem occurring with RPC is that the sender and

receiver may have different data representations.The process to correctly pack the request and parameters for the callee and vice versa is called *marshaling*.

RPC assumes that the callee is always able to respond to a request immediately. In contrast, in message-based communication queues are used, where messages are collected and the receiver decides when to answer to them. The *publish-subscribe* pattern is a message-based pattern. The messages are not directed towards a sender, but rather interested receivers subscribe to available messages.

Chapter 3

Pose Estimation of a Cylindrical Object in an OOS Scenario

3.1 Scenario

In an in-orbit manipulation or OOS scenario two spacecraft participate. The target of the manipulation is likely to be non-cooperative, meaning there is no possibility to control the spacecraft's attitude. Additionally no communication with the target is possible, e.g. to receive its self-determined pose. The servicer or chaser spacecraft approaches the target and servicing may start, if target and chaser are close enough. In the following scenario, the chaser has approached its target up to a close distance (less than 5 m). Both spacecraft are equipped with robotic manipulators. These often include cylindrical parts as seen in figure 3.1.

The chaser arm has to grasp a cylindrically shaped arm of the manipulator mounted on the target satellite, therefore the relative pose of this object has to be determined. It is assumed, that the dimensions of the cylindrical part are known.

The relative pose of the part is defined as its 3D pose consisting of an orientation vector corresponding to the cylinder's main axis and a point on this axis corresponding to the center of the cylinder with respect to the chaser's body frame coordinate system. At first, the relative pose with respect to the camera frame is determined. Knowing the transformation matrix between camera and chaser frame, the relative pose w.r.t. the chaser frame may then be computed. As a cylinder is a rotational symmetric shape, the pose estimation only indicates a possible point, which is suitable

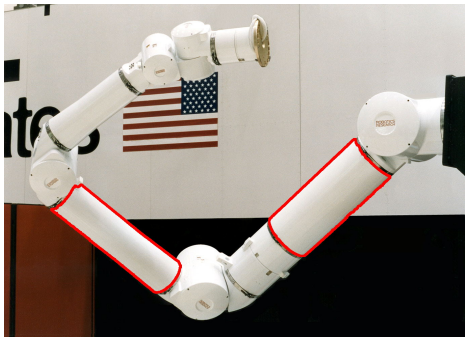


Fig. 3.1: A typical robotic manipulator arm with cylindrical parts(outlined in red) (image from `http://www.robotics-research.com/`)

to grasp. If the manipulator includes more than one cylinder, the estimation of all their poses may be used, to decide on the manipulators current configuration.

In order to regard the constraints of this master thesis, the scenario is simplified. The following assumptions are made:

- the colour of the cylindrical part is known and is unambiguous in the scenario
- no noise is created due to data transmission
- the illumination is balanced without bright spots or widespread shadowing
- the relative linear and angular velocities between target and chaser are relatively small with negligible acceleration
- the target is non-cooperative, but does at most tumbling motions

These assumptions lead to several advantages. The known colour allows a hue-based segmentation of the image and determination of a Region Of Interest (ROI) around the cylinder. Severe noise and illumination issues exacerbates the complexity of the given task as they introduce disturbances to any computer vision algorithm lowering their robustness. However, this is actually not given in space contexts and a severe simplification of the environment. The small changes in velocity prevents motion blur.

3.2 Perspective Projections of Cylinders

A cylinder is a closed solid that has two parallel circular bases connected by a curved surface[1]. The main axis of the cylinder is the line joining the center of each base. A cylinder is defined by its height h (the perpendicular distance between the bases), and the radius r of its bases. In the following sections a *right* cylinder is assumed, meaning the main axis is at a right angle to the base.

A reference frame w.r.t. the cylinder may be defined as follows: the origin is at the center of one base. The z-axis is collinear with the main axis, pointing from the origin to the center of the other base. As a cylinder is symmetrically around its main axis, the x- and y- axis may be chosen arbitrarily, perpendicular to the z-axis forming a right handed coordinate system as seen in figure 3.2. The points w.r.t. the cylinder reference frame belonging to the cylinder, are then described as:

$$C(r,h) = \{x, y, z \in \mathbb{R} | x^2 + y^2 \leq r, 0 \leq z \leq h\} \tag{3.1}$$

The contour of the cylinder surface on a perspective plane (such as the image plane) appears as two straight lines, except in the case where one of the bases is showing directly parallel to the image plane. The straight lines are either parallel or intersect in the vanishing point which may lie outside of the image borders. Only one of the bases is visible at most, appearing as a conic.

[1] `http://www.mathopenref.com/cylinder.html`

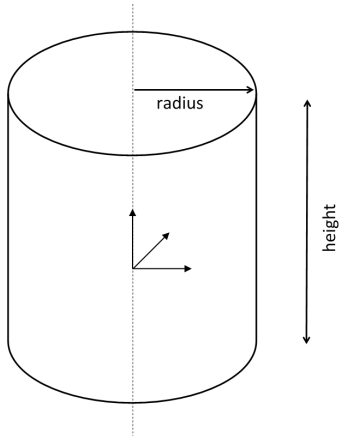


Fig. 3.2: A Cylinder parametrized by radius and height

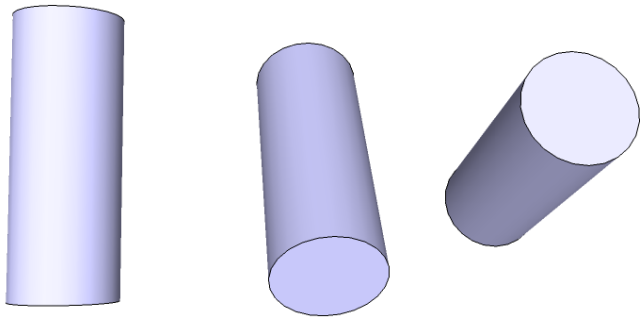


Fig. 3.3: Possible appearances of a cylinder after perspective projections

Conics are the curves obtained by intersecting a plane and a cone. These may be circles, ellipses, parabolas or hyperbolas. In the perspective plane a general conic with homogeneous coordinates is described as

$$\mathbf{a}^T\mathbf{x} = ax^2 + bxy + cy^2 + dxz + eyz + fz^2 = 0 \tag{3.2}$$

with $\mathbf{a} = (a, b, c, d, e, f)^T$ and $\mathbf{x} = (x^2, xy, y^2, xz, yz, z^2)^T$, or in matrix notation as

$$\begin{pmatrix} x & y & z \end{pmatrix} \begin{pmatrix} a & \frac{b}{2} & \frac{d}{2} \\ \frac{b}{2} & c & \frac{e}{2} \\ \frac{d}{2} & \frac{e}{2} & f \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = 0 \tag{3.3}$$

In case of an ellipse $b^2 - 4ac < 0$ holds. Ellipses are defined by their semi-major axis a, their semi-minor axis b, their center point (x_c, y_c) as well as the angle between the positive horizontal axis to the ellipse's major axis ρ.

3.3 Pose Estimation Algorithm for Cylindrical Object

3.3.1 Short Description

The pose estimation algorithm given a coloured input image consists of three parts. The sub goal is to determine either the both lines of the cylinder surface or the ellipse of one of its bases in the 2D image space. They are then converted into the 3D camera space using the known parameters radius r and height h of the cylinder. The last step is to calculate the main axis and the point on the main axis in between both cylinder bases. The line determination is supported by knowledge of the colour of the cylindrical segment as well as a a priori - estimation based on the center of mass and orientation of the coloured image parts. The process of estimating an ellipse is explained in a following section. However, it emerged quickly, that it is impractical due to its computational inaccuracy and the fact that the cylinder bases are often occluded as well.

As the colour of the cylinder is known, at first a ROI is defined with respect to its HSV colour values. HSV colour values allow for easier colour based segmentation, as colours may be determined by the *hue* value only independent of brightness or saturation. It is assumed, that the cylinder is the only object with the given colour, so the ROI is determined as the minimum rectangle which encloses the coloured object. In the worst case the ROI does not reduce the problem space, e.g. when the cylinder's projection is with the main axis parallel to the image plane and rotated around 45°. However, the colour segmentation may be used to mask the image, so that only pixels which are part of the cylinder are non-black.

The extracted ROI is afterwards converted to an image containing only the contour of the cylinder. For this purpose the Canny edge detector may be used. As a next step the connected line segments are extracted. Line segments which are shorter than a threshold are disregarded.

In order to disconnect straight lines and arcs, a recursive line segment cutting algorithm is used. It is one of the oldest methods of line simplification and has already been proposed by Ramer in 1972[29]. The line segment is recursively divided at the point which is furthest away from the line joining both end points of the segment. Two thresholds may be given: one threshold to decide the minimum line length in pixels, and one threshold to decide the minimum distance to the line of the point to be cut. Again, too short segments after the cut are neglected.

Into each remaining line segment a straight line has to be fitted. On that account a linear regression algorithm is chosen, which was extended in order to be applicable for vertical lines. The algorithm is described in the next section. The line estimates, which have a high pixel support (e.g. the squared error of distances is less than 2 pixels) are then compared to the principal orientation of the coloured part. Every line with an angular displacement towards the orientation greater than a threshold (e.g. 35°) is disregarded. Such lines might belong to objects occluding part of the cylinder. Another possibility to determine the straight lines given in the picture is the application of a Hough transform. This was decided against because of its need of

an accumulator array, whose size is dependent on the accuracy of the line estimates.

The most reasonable pair of lines is chosen as candidates. Using a point parametrization of the 2D lines, the direction cosines of the cylinder's z-axis are calculated according to equation 2.17. Knowing, that corresponding points on both sides of the cylinder lie on a circle with given radius in the plane given by the normals to the z-axis, an anchor point for the z-axis is determined.

3.3.2 Fitting Straight Lines

A linear regression is used to estimate a straight line into each segment. Usually, a Hough transform might be used to find straight lines. In a Hough transform for each edge pixel, line estimates parametrized using the normal representation with ρ and d are accumulated in an array. The maxima then correspond to found lines. This approach is robust against occlusion, as the line segments do not have to be connected. However, the accumulator array has high memory requirements. In the existing case the line segments are connected, though, and due to the previously performed line simplification are thought to be either straight lines or arcs.

In general, a simple linear regression is a least squares method and performed as follows[30]: At first the means $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ are determined, where n is the number of elements used for estimation of the line. The slope m is then calculated as:

$$m = \frac{\sum_{i=1}^{n} y_i x_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2} \tag{3.4}$$

The offset b is easily determined as $b = \bar{y} - m\bar{x}$. In the case of a vertical line, which is described as $x = c$, where c is a constant, linear regression does not lead to a reasonable result. In that case the constant c is estimated as $c = \bar{x}$. The relation between ρ and b is given as $b = \frac{\sin\rho}{\cos\rho} = \tan\rho$. Then d is determined with the normal line equation.

3.3.3 Fitting Ellipses

A popular method of fitting ellipses is direct least square fitting introduced by Fitzgibbon et al.[31]. In comparison to other methods, their proposal is ellipse specific and may be solved analytically. The problem to be solved is constrained by the equation $4ac - b^2 = 1$ which is a scaled version of the constraint for an ellipse as given in section 3.2. In matrix form $\mathbf{a}^T\mathbf{C}\mathbf{a} = 1$ it may be expressed as

$$\mathbf{a}^T \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{a} = 1 \tag{3.5}$$

The ellipse fitting problem is then:

$$\text{Minimize } E = \|\mathbf{Da}\|^2, \text{ subject to the constraint } \mathbf{a}^T\mathbf{a} = 1 \tag{3.6}$$

The matrix $\mathbf{D}$ is called design matrix and defined as the $n \times 6$ matrix $[\mathbf{x}_1\mathbf{x}_2 \dots \mathbf{x}_n]$. The problem is solved by building the scatter matrix $\mathbf{S} = \mathbf{D}^T\mathbf{D}$ and finding the eigenvalues to the matrix $\mathbf{S}^{-1}\mathbf{C}$.

A version of the direct least square algorithm was implemented using the C++ library OpenCV. The results were compared to the *Matlab* implementation provided by [31], showing that the OpenCV implementation of finding eigenvalues yields different outcomes, which did not correspond to useful results. Therefore, the idea of using ellipse fitting in the algorithm was discarded.

3.3.4 Estimating the pose given two line candidates

After line candidates are determined, which are assumed to be the perspective projections of the cylinder's sides , the image processing part of the algorithm is completed. The next step is the determination of the straight line corresponding to the z-axis of the cylinder. Knowing that both line candidates are the projections of parallel lines, the direction cosines $\mathbf{b} = \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix}^T$ corresponding to the cylinder's main axis are calculated as described by Haralick[20]. One of the line candidates is parametrized as

$$l_1 = \{\mathbf{x} | \mathbf{x} = \begin{pmatrix} c_1 \\ d_1 \end{pmatrix} + \lambda \begin{pmatrix} g_1 \\ h_1 \end{pmatrix}\} = \{\begin{pmatrix} x \\ y \end{pmatrix} \mid \cos\rho_1 x + \sin\rho_1 y = \delta_1\} \tag{3.7}$$

and the other line as

$$l_2 = \{\mathbf{x} | \mathbf{x} = \begin{pmatrix} c_2 \\ d_2 \end{pmatrix} + \lambda \begin{pmatrix} g_2 \\ h_2 \end{pmatrix}\} = \{\begin{pmatrix} x \\ y \end{pmatrix} \mid \cos\rho_2 x + \sin\rho_2 y = \delta_2\} \tag{3.8}$$

Then the direction cosines is calculated using equation 2.17. To find an anchor point on the line through the z-axis, two opposing points on the projected line candidates. At first an arbitrary point on the first line is chosen.

$$\mathbf{p} = \begin{pmatrix} u_p \\ v_p \end{pmatrix} = \begin{pmatrix} l_1 \cos\rho_1 \\ l_1 \sin\rho_1 \end{pmatrix} \tag{3.9}$$

In order to determine the corresponding point on the second line, the straight line which has the same angular distance to each of the other lines has to be found. It is parametrized as:

$$l_0 = \{\mathbf{x} | \mathbf{x} = \begin{pmatrix} c_0 \\ d_0 \end{pmatrix} + \lambda \begin{pmatrix} g_0 \\ h_0 \end{pmatrix}\} = \{\begin{pmatrix} x \\ y \end{pmatrix} \mid \cos\rho_0 x + \sin\rho_0 y = \delta_0\} \tag{3.10}$$

The normal angle ρ_0 may be determined as:

$$\rho_0 = \text{atan2}(\sin\rho_2 - \sin\rho_1, \cos\rho_1 - \cos\rho_2) \tag{3.11}$$

The direction cosines of the line are then chosen as $g_0 = -\sin\rho_0$ and $h_0 = \cos\rho_0$.

The corresponding point lies on the second line.

$$\mathbf{q} = \begin{pmatrix} c_2 \\ d_2 \end{pmatrix} + j \cdot \begin{pmatrix} g_2 \\ h_2 \end{pmatrix} \tag{3.12}$$

The angle between the mid line and the line connecting the corresponding points is a right angle. Using the vector representation of the lines, it yields that

$$\begin{pmatrix} u_p \\ v_p \end{pmatrix} + i \cdot \begin{pmatrix} -g_0 \\ h_0 \end{pmatrix} = \begin{pmatrix} c_2 \\ d_2 \end{pmatrix} + j \cdot \begin{pmatrix} g_2 \\ h_2 \end{pmatrix} \tag{3.13}$$

The two equations are then rearranged to determine either i or j, in order to find the corresponding point $\mathbf{q}$.

$$j = \frac{g_0(u_p - c_2) + h_0(v_p - d_2)}{g_0 g_2 + h_0 h_2} \tag{3.14}$$

The point, of which $\mathbf{p}$ is the projection, shall be denounced as $\mathbf{P}$ and analogous point $\mathbf{Q}$ to the projected point $\mathbf{q}$. Each of the three dimensional points is on a circle with radius r and center $\mathbf{M}$. The three dimensional circle equation is:

$$\mathbf{X} = r\cos\phi\hat{\mathbf{e}}_1 + r\sin\phi\hat{\mathbf{e}}_2 + \mathbf{M} \tag{3.15}$$

The vectors $\hat{\mathbf{e}}_1$ and $\hat{\mathbf{e}}_2$ are unit vectors in the plane perpendicular two the direction cosine vector $\mathbf{b}$ and perpendicular to each other. The vector between $\mathbf{Q}$ and $\mathbf{P}$ is computed using the circle equations.

$$\begin{aligned} \mathbf{P} - \mathbf{Q} &= (r\cos\phi_p\hat{\mathbf{e}}_1 + r\sin\phi_p\hat{\mathbf{e}}_2 + \mathbf{M}) - (r\cos\phi_q\hat{\mathbf{e}}_1 + r\sin\phi_q\hat{\mathbf{e}}_2 + \mathbf{M}) \\ &= r(\cos\phi_p - \cos\phi_q)\hat{\mathbf{e}}_1 + r(\sin\phi_p - \sin\phi_q)\hat{\mathbf{e}}_2 \end{aligned} \tag{3.16}$$

Using identities for sine and cosine differences

$$\alpha - \sin\beta = 2\cos\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right) \tag{3.17}$$

and

$$\alpha - \cos\beta = 2\sin\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right) \tag{3.18}$$

and denoting the variable $\phi' = \frac{\phi_p+\phi_q}{2}$ the equation simplifies to:

$$\mathbf{P} - \mathbf{Q} = 2r(\cos\phi'\hat{\mathbf{e}}_2 - \sin\phi'\hat{\mathbf{e}}_1) \tag{3.19}$$

Now the projection equations are used. They yield:

$$\begin{pmatrix} \frac{1}{f}\left(Z_p u_p - Z_q u_p + h_0 Z_q i\right) \\ \frac{1}{f}\left(Z_p v_p - Z_q v_p - g_0 Z_q i\right) \\ Z_p - Z_q \end{pmatrix} = 2r(\cos\phi'\hat{\mathbf{e}}_2 - \sin\phi'\hat{\mathbf{e}}_1) \tag{3.20}$$

The angle ϕ' is calculated as:

$$a = (h_0 (\hat{\mathbf{e}}_{2,z} v_p - f\hat{\mathbf{e}}_{2,y}) + g_0 (\hat{\mathbf{e}}_{2,z} u_p - f\hat{\mathbf{e}}_{2,x})) \tag{3.21}$$

$$b = (h_0 (f\hat{\mathbf{e}}_{1,y} - \hat{\mathbf{e}}_{1,z} v_p) + g_0 (f\hat{\mathbf{e}}_{1,x} - \hat{\mathbf{e}}_{1,z} u_p)) \tag{3.22}$$

$$\phi' = \text{atan2}(-a, b) \tag{3.23}$$

Solving for Z_q yields two possible equations, which have to be used depending on h_0 or g_0 unequal to zero.

$$Z_q = \frac{2r}{h_0 i} (\cos\phi' (\hat{\mathbf{e}}_{2,z} u_p - f\hat{\mathbf{e}}_{2,x}) + \sin\phi' (f\hat{\mathbf{e}}_{1,x} - \hat{\mathbf{e}}_{1,z} u_p)) \tag{3.24}$$

$$Z_q = \frac{2r}{g_0 i} (\cos\phi' (\hat{\mathbf{e}}_{2,z} v_p - f\hat{\mathbf{e}}_{2,y}) + \sin\phi' (f\hat{\mathbf{e}}_{1,y} - \hat{\mathbf{e}}_{1,z} v_p)) \tag{3.25}$$

$$\tag{3.26}$$

Z_p is then calculated easily and a point on the line is given by $0.5(\mathbf{P} + \mathbf{Q})$. As an estimation for the cylinder position, the point on the line with closest distance to its mean, may be chosen.

Chapter 4

Tools

This chapter presents the tools used in this master thesis. The term *tool* here refers to software tools as well as programming libraries. They are introduced in the order of how they appeared in the working process. The robotic simulator V-REP for example was used to create synthetic test data as input of the algorithm. The open source computer vision library OpenCV by contrast was used to implement a prototype of the application, which simultaneously serves as a reference for tests. Eventually, the real time operating system RODOS provided the necessary functionality to design a distributed version of the application.

4.1 V-REP

The *Virtual Robot Experimentation Platform* V-REP is a robot simulation framework, which is freely available for download, if used for educational purposes[1]. It was chosen, as it is possible to simulate a camera and its input, while being able to freely manipulate the scene to be recorded.

In V-REP a scene is similar to a project in a development environment. A V-REP scene consists of several scene objects. These are hierarchically ordered in a tree-like structure called scene hierarchy and every object may be extended by a script, which defines its behaviour[32]. Next to embedded scripts written in the *Lua* script language there are various other ways to program control algorithms for V-REP simulations, for example using plug-ins, a remote Application Programming Interface (API)-client or even a ROS node.

The standard user interface of V-REP as seen in figure 4.1 consists of several parts. The figure was extended visually with a red overlay and numbers which are used to explain the functionality. The 3D rendered scene (number 1) is displayed in the right half of the screen. Additionally, it may be split into different pages showing other views (e.g. top view, side view, a coordinate system, ...) by using the page selector (number 2). The next important part is the scene hierarchy (number 3). It allows changing an object's name, manipulating the hierarchy by drag and drop and to access the scripts. Complex objects are added to the scene by using the model browser (number 4). The models are sorted into folders. The ant hexapod robot seen in the rendering for example was chosen from the model browser. The panel on the left side (number 5) gives access to several dialogues. The first button opens the

[1] `http://www.coppeliarobotics.com/downloads.html`

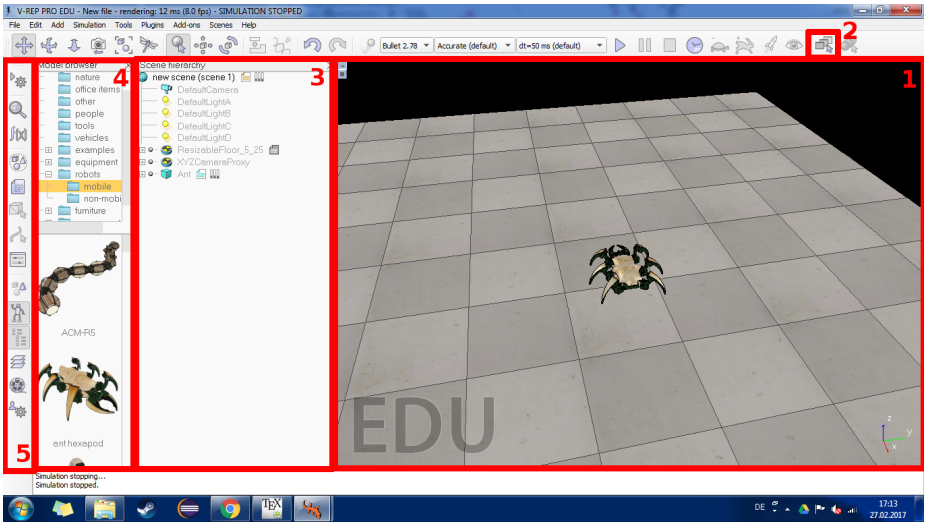


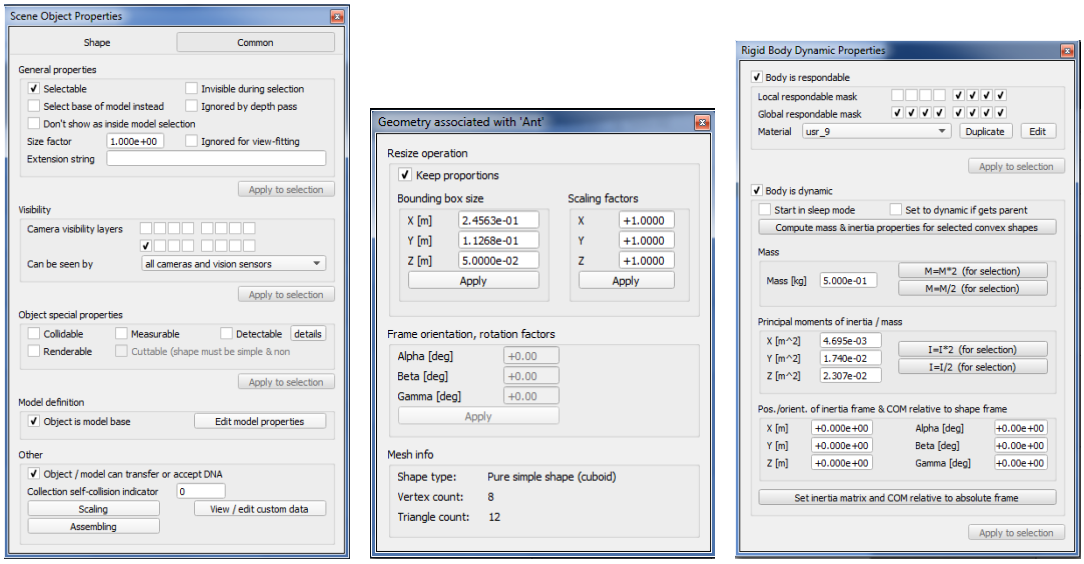
Fig. 4.1: The standard UI of V-REP

simulation settings, where among others the simulation time step may be changed. Some of the functions are also accessible by a button in the top panel. The second button opens the scene object properties of the selected object (see figure 4.2(a)). This dialogue may also be opened by double clicking on the object in the scene hierarchy. In the object properties dialogue the geometry (see figure 4.2(b)) and the dynamic responses of the object (see figure 4.2(c)) can be changed. The calculations modules are opened by clicking the third button in the side panel. Here for example the gravity setting is adjusted. The button after that allows to form collections of scene objects. The fifth button opens a overview of existing scripts. Button 6 and 7 are only available, when an object or path is selected and allow editing of them. The eighth buttons opens the OpenGl-based custom UIs view. Selection of multiple paths by type is available after clicking button 9. The next two buttons hide the model browser and scene hierarchy. Button 12 allows choosing the visible layers. A video recorder is available through the next button and the last buttons opens the user settings

A scene object like a simple shape or a model is by default subject to dynamics. If that is not desired, it may be deactivated in the body dynamics properties. In order to extend the functionality of an object a child script may be added in the right click menu. A non-threaded Lua child script is segmented in several parts. Those are depicted in listing 4.1. The initialization part is only executed once, when the scene object is inserted into the simulation scene (which might be at the the beginning of the simulation).

The *vision sensor*[2] is the scene object, which is needed to record synthetic test data. It has a built-in filtering function which among other things allows to add noise to the scene. In order to be detected by a vision sensor, the object needs to be renderable. There are two different types of vision sensors, *orthographic projection-type* sensors have a rectangular field of view, while *perspective projection-type* sensors

[2] `http://www.coppeliarobotics.com/helpFiles/index.html`



(a) The scene object properties dialogue (b) The geometry properties dialogue (c) The dynamic properties dialogue

Fig. 4.2: Some of the Property dialogues of V-REP

Listing 4.1: The standard V-REP child script content

```
if (sim_call_type==sim_childscriptcall_initialization) then
    -- here any objects needed by the script
    --should be initialized, e.g. parameters
end

if (sim_call_type==sim_childscriptcall_actuation) then
    -- here any actuation work should be added
end

if (sim_call_type==sim_childscriptcall_sensing) then
        -- here any sensing object is doing its work
end

if (sim_call_type==sim_childscriptcall_cleanup) then
    -- Put some restoration code here
end
```

have a trapezoidal field of view. The last type of sensor is more suited for a camera-like sensor. Four properties may be set in perspective mode as shown in figure 4.3. The near and far clipping plane define the range in which objects are detected by the sensor. The perspective angle defines the maximum opening angle of the detection volume. If the horizontal resolution is greater than the vertical resolution, then it is the horizontal angle and vice versa. As the vision sensor is not a real camera, the

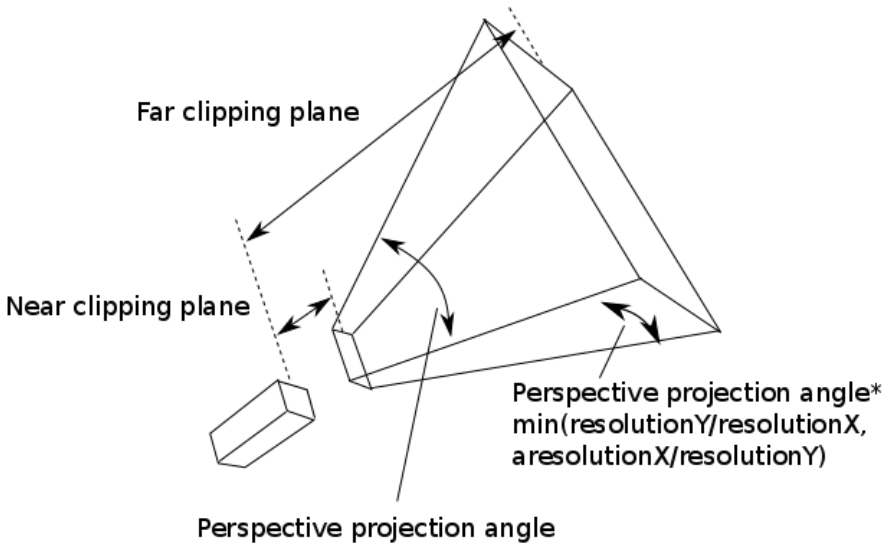


Fig. 4.3: The properties of a V-REP vision sensor in perspective projection mode, from the V-REP manual

focal length is not a property of the vision sensor.

4.2 OpenCV

OpenCV is an open source computer vision library. It has been introduced by Intel in June, 2000. Currently, OpenCV is maintained by Itseez, official releases occur every six month. A lot of contemporary computer vision algorithms are added yearly through *Google Summer of Code* projects. OpenCV is an optimized library and supports GPU acceleration and parallelism. In order to turn off optimization and enforce sequential execution, at the beginning of an OpenCV program the functions `cv::setNumThreads(0)` and `cv::setUseOptimized(false)` have to be called.

In this thesis, OpenCV was used in version 3.1 and may be downloaded on its website[3]. The documentation is also available online[4]. The library is divided into several modules. By far not all of them are needed to understand the reference implementation. However, extracts of the four basic modules *core*, *imgproc*, *imgcodecs* and *highgui* are explained in this section.

The easiest way to compile an OpenCV application is to use CMake and a C++ compiler. The CMakeLists.txt file should look similar to listing 4.2.

4.2.1 The Core Module

The core module provides a data structure for images and other utility classes. An image is interpreted as a dense n-dimensional array and stored in an object of the

[3] `http://opencv.org/downloads.html`
[4] `http://docs.opencv.org/3.1.0/`

Listing 4.2: CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
project( DisplayImage )
find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )
add_executable( DisplayImage DisplayImage.cpp )
target_link_libraries( DisplayImage ${OpenCV_LIBS} )
```

`Mat` class[5]. Image data is laid out in an one-dimensional array, where the image is stored row-by-row. A multi-channel image, for example a colour image, is stored plane-by-plane.

The data to be stored in a `Mat` is defined by its type. The type is given as constant, `CV_8UC1` for example describes an 8 bit single channel image, an image of type `CV_32FC2` has two channels and contains 32-bit floating point numbers. Copying a matrix or extracting a ROI is executed in $O(1)$ time, as only the matrix headers are copied and the data is accessed by reference. In order to perform a deep copy, `Mat cv::clone()` has to be called.

Data in a matrix is accessed either element-by-element or through row pointers. The second way is more efficient, if whole rows of the matrix need to be processed. A rectangular submatrix is extracted using the bracket operator. The argument given is of type `Rect` which is also a class of the core module. A `Rect` is defined by its top left corner, height and width.

Different type of point classes are available. The simple `Point` class represents a two dimensional vector of integers and is conveniently used e.g. for pixel coordinates. Points in a 3D environment are represented by the `Point3f` or `Point3d` classes, which hold 3D vectors of floats or doubles.

4.2.2 The Image Processing Module

The Image Processing Module (imgproc) includes classes and functions for image filtering, image transformation, feature and object detection and more.

The morphological operations blur and dilate are implemented in this module. Both functions require a kernel, which defines the shape of the pixel neighbourhood of which the minimum, respectively maximum is taken. To create the kernel, for example the function `getStructuringElement(...)` may be used. For dilation the formula is:

$$\mathrm{dst}(x, y) = \max_{(x',y'):\mathrm{kernel}(x',y')\neq 0} \mathrm{src}(x + x', y + y')$$

Blurring is also a filtering operation. OpenCV offers among others, Gaussian and median blur functions. Besides input and output image the function `GaussianBlur (...)` requires the Gaussian kernel standard deviation in both x- and y-direction. If both are set to zero, they are computed using the given size.

[5] see `http://docs.opencv.org/3.1.0/d3/d63/classcv_1_1Mat.html`

In order to convert an image to another colour space the function `cvtColor(...)` is used. The code argument is an element of the enum `ColorConversionCodes`. In order to convert between colour RGB and grayscale image for example, the code `COLOR_BRG2GRAY` is used, `COLOR_BGR2HSV` converts a RGB image to the HSV colour space.

The thresholding function `threshold(...)` does exactly what its name indicates. There are several threshold types which are depicted in figure 4.4.

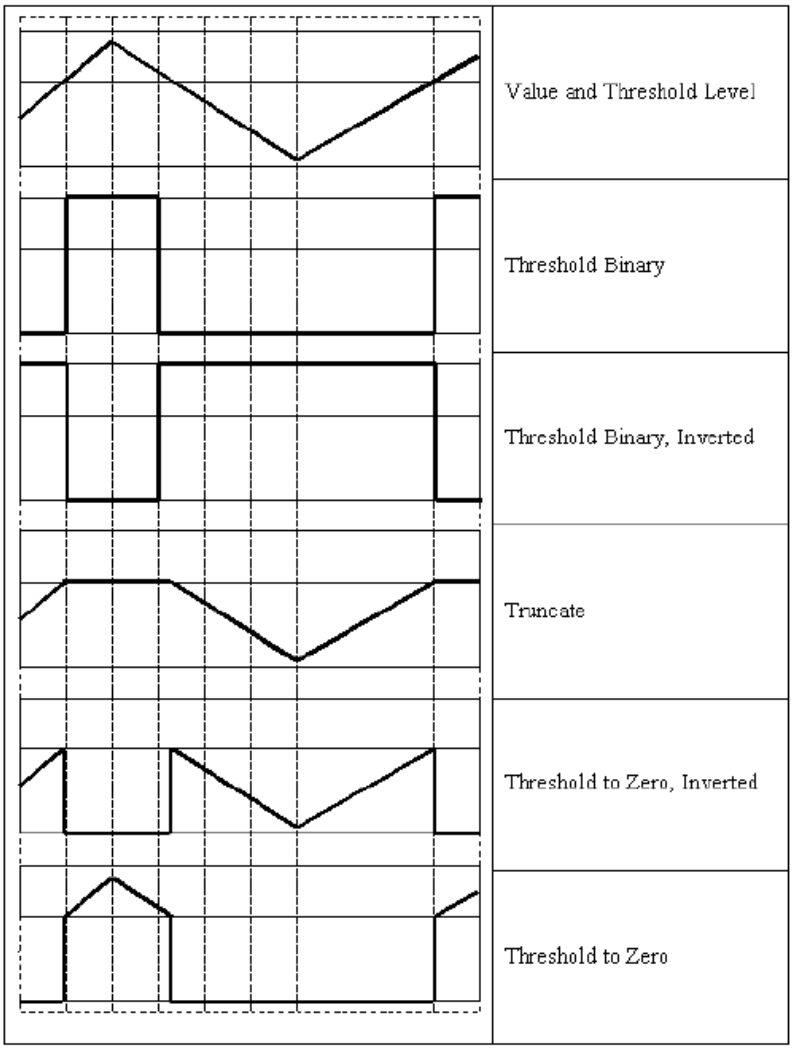


Fig. 4.4: The different thresholding types implemented in OpenCV. From second top to bottom: `THRESH_BINARY, THRESH_BINARY_INV, THRESH_TRUNC, THRESH_TOZERO_INV, THRESH_TOZERO`, (image source: OpenCV documentation)

As edges are image features, the implementation of the `Canny(...)` function in the documentation is found under the label of feature detection. The function requires an 8-bit input image. The edges are returned as a 8-bit image as well, which has the same size as the input image. The implementation uses the standard algorithm explained in the chapter before. Instead of a default L_1 norm the more accurate L_2 norm may be used by setting a flag.

In order to detect line segments the output of the Canny algorithm may then be fed to a line detector. OpenCV supplies a factory function to create a custom line detector(listing 4.3).

Listing 4.3: The OpenCV factory function to create a line segment detector

```
Ptr<LineSegmentDetector> cv::createLineSegmentDetector  (
        int     _refine = LSD_REFINE_STD,
        double  _scale = 0.8,
        double  _sigma_scale = 0.6,
        double  _quant = 2.0,
        double  _ang_th = 22.5,
        double  _log_eps = 0,
        double  _density_th = 0.7,
        int     _n_bins = 1024
)
```

The refinement argument determines the level of line refinement and is either none, standard or advanced. Standard refinement breaks arcs into smaller line approximations, advanced refinements increases precision, etc.

4.2.3 Image reading and High GUI

Images of various formats are read to image matrices using the `imread(...)` method. By default they are read as colour images with channels stored in Blue, Green, Red order. In order to look at results, it is possible to display images using the graphical user interface options of OpenCV. A window is created using `namedWindow(...)`, the image then is sent to the window by `imshow(...)`. The function `waitKey(int delay = 0)` has to be called afterwards as it is the only function that processes GUI events, otherwise the image will not show. If called with argument zero, the function will wait infinitely, until a Key is pressed. A short example of this is given in listing 4.4.

Listing 4.4: Loading and displaying an image

```
...
namedWindow("EXAMPLE");
Mat img = imread("example.png";
imshow("EXAMPLE",img);
waitKey();
...
```

4.3 RODOS

RODOS is a real-time operating system developed by the German space agency DLR which is grounded on the principle of simplicity. The framework is available online[6].

[6] `http://www.montenegros.de/sergio/rodos/rodos-de.html`

RODOS runs on different embedded platforms and on top of Linux, allowing fast development. The RODOS core is implemented in object oriented C++ and the API is easy to understand.

In order to compile an application a bash shell is used. At first the RODOS environment variables have to be set by calling `$ source make/rodosenvs` in the RODOS core directory. At the first time, the library has to be compiled for the given target system. In case of Linux one has to call `$ linux-lib` in an arbitrary folder. The command `$ linux-executable <filename> [<filename> <filename>` compiles a Linux binary. The executable is named *tst* by default.

Only a few basic elements are needed to create complex applications. The RODOS middleware takes care of the communication between tasks within an application. A task is implemented by creating a class inheriting from the `RODOS::Thread` class and instantiating static objects of it. Every thread brings its own context and stack. A scheduler is responsible for thread execution control and context switch.

Threads may execute their task periodically or react to events. A periodic thread suspends itself after execution and is resumed by the scheduler at the desired point in time. The system time is counted with nanosecond precision and starts at boot time. To simplify the access of time related functions the following macros exist:

- `NOW()` returns the current time in nanoseconds
- `SECONDS_NOW()` returns the current time in seconds
- `AT(time)` suspends the caller until the time point is reached
- `TIME_LOOP(firstExecution,period){...}` is used to control periodic execution

Time units are also given as macros: `NANOSECONDS , MICROSECONDS , MILLISECONDS , SECONDS , MINUTES, HOURS , DAYS , WEEKS , END_OF_TIME`. The last constant is the highest time possible and lies about 293 years in the future which should be sufficient for most applications.

The example in listing 4.5 shows an example of a simple thread, which increments a counter every 2 seconds. The thread execution starts one second after boot. The thread main task is implemented in its `run()` method. The time loop macro suspends the thread automatically after each loop.

Listing 4.5: A simple RODOS thread

```
1  static int cnt = 0;
2
3  class SimpleThread : Thread{
4  public:
5     SimpleThread() : Thread("SimpleThread"){}
6     void run(){
7       TIME_LOOP(1*SECONDS,2*SECONDS){
8           cnt++;
9       }
10
11    }
```

```
12
13 } simplethread;
```

Important threads are given a priority, which ensures that they are not interrupted by less important tasks. If a task needs to be executed without interruption, for example when accessing a variable, a `Semaphore` may be used. A semaphore is a data structure used to control the access to limited resources. If a semaphore is occupied, the process is blocked until the occupying process releases the semaphore.

The RODOS middleware implements the publisher-subscriber protocol and relies on topics. A topic is a pair of a data-type and an integer identifier and may be understood as a single communication channel. Messages are published to the topic and the middleware distributes them locally and, using gateways, also to other computing nodes in the network. In order to receive the message, a subscriber has to be instantiated. Each subscriber is connected to exactly one topic. Simple subscribers may forward the received message to a `Fifo`[7] or `CommBuffer`[8], which is then accessed by a thread. More complex subscribers are implemented by inheriting from the `RODOS::Subscriber` class.

Listing 4.6 is an example for the different kinds of subscribers. All topics are of integer type. The first subscriber enqueues received messages to a Fifo. The second subscriber prints the message directly.

Listing 4.6: Different kinds of subscribers

```
1  Fifo<int, 10> fifo;
2  static Subscriber fifoSubscriber(counter1, fifo, "
       fifoSubscriber");
3
4  class Printer : public Subscriber{
5  public:
6      Printer() : Subscriber(counter2, "printer") { }
7      long put(const long topicId, const long len, const void*
           data, const NetMsgInfo& netMsgInfo) {
8                  PRINTF( "Received:␣%d", *(int*)data);
9                  return 1;
10         }
11 } printer;
```

In order to distribute messages beyond the boundaries of the computing node, a `RODOS::Gateway` is used. The gateway requires an interface to the desired network over which the messages are to be distributed. Interfaces are implemented for several hardware/software protocols. Listing 4.7 is an example of creating a gateway which uses UDP.

Listing 4.7: Creating a gateway using a UDP connection

```
1  static UDPInOut udp(-5001); // negative number for broadcast
2  static LinkinterfaceUDP linkinterfaceUDP(&udp);
```

[7] Also called a queue, a FiFo is a First in First out data structure.

[8] In contrast to a queue, the buffer only holds the latest received message.

```
3 static Gateway gateway1(&linkinterfaceUDP);
```

Chapter 5

Implementations of the Pose Estimation Algorithm

5.1 Reference Implementation using OpenCV

5.1.1 Implementation

At first the pose estimation algorithm explained in the previous chapter has been implemented using the OpenCV library, which has been given an introduction to in chapter 4. There are several advantages to this approach:

- There is no need to re-implement the basic image processing procedures, instead the focus lies on the estimation algorithm itself
- It simplifies the access to a video stream of a USB web cam or to read images from hard disk
- OpenCV is widely used, in commercial products as well as in research, and therefore a reasonable and well tested point of reference

The three parts of the algorithm were implemented in three functions to allow individual testing. The first function converts the input image into a contour image of the cylinder. The image is afterwards cropped to the cylinder's bounding rectangle, further called the ROI. The ROI of the previous iteration may be used to reduce the input image size.

The second function implements the line extraction. The contour image produced by the first function has binary content. Connected non-zero pixels are extracted as edges. Edges are actually vectors of pixel coordinates. However, an `Edge` class wrapping those vectors simplifies splitting of edges. Also a `Line` class has been implemented in order to keep track of found lines. A line is created with either two points or its polar parameters. Additionally, also an edge may be used to estimate the line based on linear regression. The two line candidates are chosen considering the lines' length, straightness and orientation towards each other.

Both candidates are finally fed to the third function and used to compute an estimated position and orientation. The formulas used are explained in the previous chapter. The accuracy of the orientation estimation depends only on the correct identification of the cylinder sides. The estimation of the position is largely depending on two additional factors: the orientation of the cylinder and its occlusion.

5.1.1.1 The Utility Classes and Functions

Utility classes and functions have been implemented under the namespace `uwi`[1]. This includes a class for lines and for edges, but also functions connected to perspective computations. To implement simple classes for points etc. has been refrained from, as the OpenCV core modules include templated classes for points of all dimensions and types of numbers.

Pixel coordinates are represented as objects of the `cv::Point` class. Computations in two- and three-dimensional space are performed with double points (`cv::Point2d` and `cv::Point3d`). The edge and line classes also use OpenCV objects for their attributes.

An `uwi::Edge` is a wrapper object. It holds a pointer to a vector of `cv::Point` objects, which is managed outside of the class. The edge is initialized with the pointer and a start- and end-index. The advantage of using a dedicated edge class lies in the easier splitting into segments. Instead of copying the vector, segments may be created by instantiating multiple edge objects with varying indices.

Algorithm 5.1: The Edge Extraction Algorithm

input : An edge image Im, where edge pixels have value 1, and a minimal edge size $minSize$
output: A set $edges$ of connected edge segments

```
V ⟵ ∅;
for Pixel p ∈ Im do
    if p ∉ V and value(p) = 1 then
        E ⟵ {p};
        tmpP ⟵ p;
        while tmpP has neighbours do
            tmpP ⟵ nextNeighbour(tmpP);
            V ⟵ V ∪ {tmpP};
            E ⟵ E ∪ {tmpP};
        end
        if size(E) ≥ minSize then
            edges ⟵ edges ∪ E;
        end
    end
end
```

In order to turn a edge map into a collection of edge objects, at first the pixel vectors are extracted. The algorithm in listing 5.1 walks through the image row by row until it finds an edge pixel. It then follows the neighbours of the pixel and adds them to a temporal edge until a pixel does not have further neighbours. If the size of the temporal edge is greater than a threshold, the edge is added to the set of edges to be returned. Then the search for the next edge pixel begins, ignoring all pixels which

[1]Uwi is meant to be an abbreviation for Uni Würzburg Image and has been chosen arbitrarily

are already part of an edge. The edge segments are then fed to the Ramer-Douglas algorithm as given in algorithm 5.2:

Algorithm 5.2: The Edge Simplification Algorithm based on Ramer-Douglas-Peucker

input : A connected edge segment e, the minimal cutting size $minCut$ and maximal distance $maxDistance$
output: A set E of connected edge segments, at the beginning $E \longleftarrow \emptyset$

find the line l through the first and last edge element; find the edge element $elem$ with greatest distance to l; **if** *distance(elem,l)* $\geq maxDistance$ **then**
  if *size(*$e(1, elem)$*)* $\geq minCut$ **then**
    call algorithm recursively for $e(1, elem)$;
  end
  if *size(*$e(elem, end)$*)* $\geq minCut$ **then**
    call algorithm recursively for $e(elem, end)$;
  end
else
  $E \longleftarrow E \cup \{e\}$;
end

A `uwi::Line` represents a two-dimensional line. Internally, it is represented through its polar coordinates. A line's support is given by the amount of pixels which is used to create a line. Lines may be compared using `bool compareLines(Line&, Line&)` or tested for equality using `bool equalLines(Line&,Line&)`. Comparing considers not only the line parameters, but also their support. The second function is a *dirty* function, as it adjusts the lines' support. If two lines are equal, their parameters are re-computed using the original parameters weighted by support. In order to account for discretization errors, the parameters only have to be equal up to a small difference.

5.1.1.2 The Function `createEdgeImage`

The algorithm 5.3 summarizes the function. The main input is the RGB colour image and the ROI of the last pose estimation.

If no ROI is given the complete input image is used for the subsequent operations, otherwise the ROI is increased by a fourth of its original size. The factor was chosen to account for a enlargement of the region due to movement of the cylinder. A fourth seemed large enough to include the complete cylinder. The value may be adjusted if the maximal movement of the cylinder is known. In order to find the cylinder, only pixels with a hue value in the given range are considered. Two morphological operations are executed. The erosion removes all pixels, that have inactive neighbours (these pixels are considered noise). The dilation then adds wrongly removed pixels at the cylinder border.

Remaining in the image is now a set of pixels in the given hue range, which should be the projection of the cylinder, we were looking for. The bounding rectangle of the blob serves as new ROI. To obtain the mean, orientation and eccentricity, the image

Algorithm 5.3: Creating and Analysing an Edge Image given a Colour Image

input : A coloured image *input*,
the minimal and maximal hue values to be considered *hueRange*,
the ROI *roi* of the previous iteration

output: the binary image containing edges *edgeImg* (in this algorithm treated as set of active pixels),
the *mean* of the active pixels,
their orientation as angle θ and eccentricity *ecc*

```
edgeImg ⟵ ∅;
if roi smaller than size(input) then
    enlarge roi by a fourth of its size;
end
for Pixel p ∈ input(roi) do
    if hue(p) ∈ hueRange then
        edgeImg ⟵ edgeImg ∪ {hue(p)};
    end
end
erode(edgeImg);
dilate(edgeImg);
mean ⟵ mean(edgeImg);
ecc ⟵ eccentricity(edgeImg);
theta ⟵ orientation(edgeImg);
roi ⟵ boundingBox(edgeImg);
edgeImg ⟵ Canny(edgeImg);
```

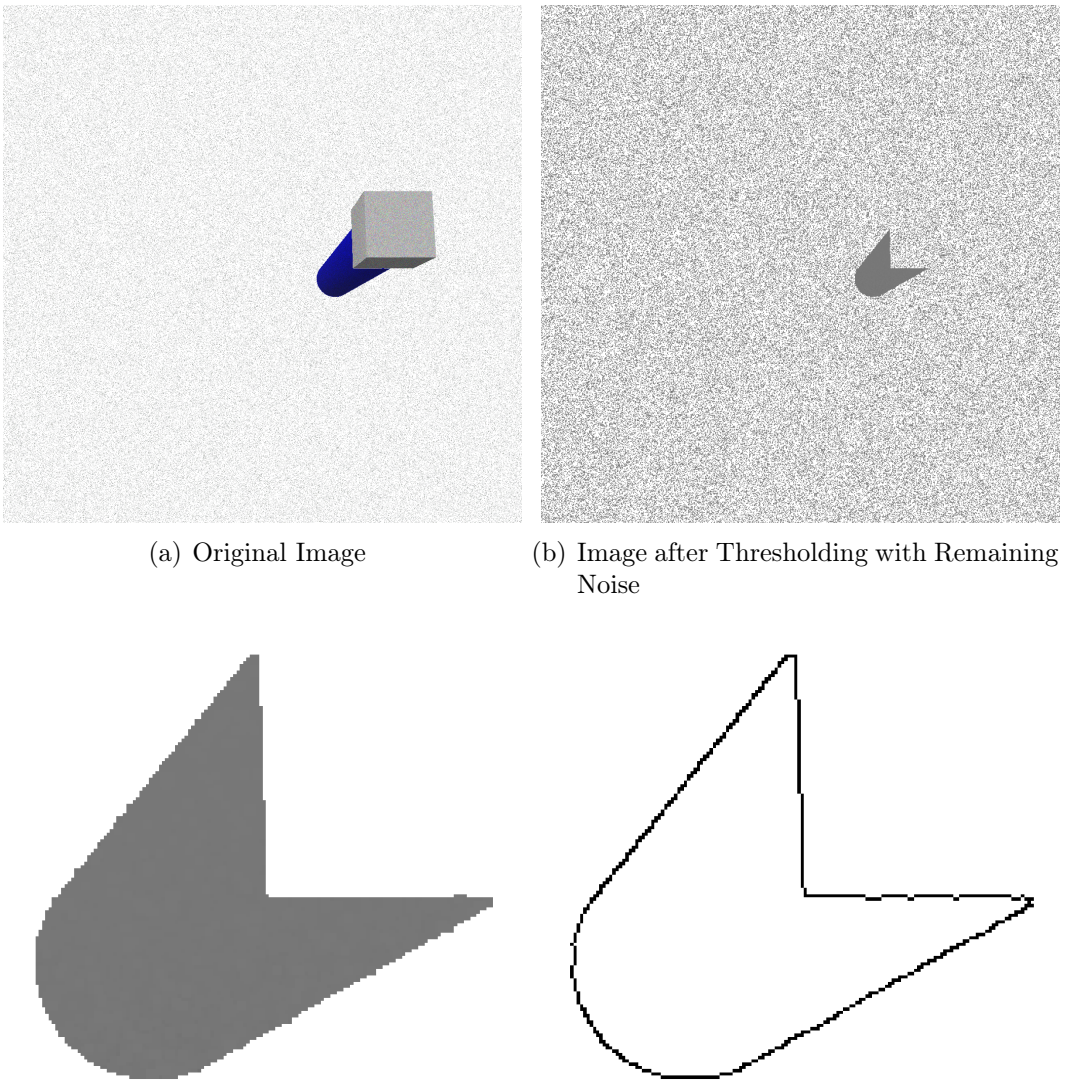
moments up to the second order are computed. The moment M_{ij} is calculated as

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y) \tag{5.1}$$

.

The mean corresponds to the moments of first order, while the orientation angle θ is computed as

$$\theta = \frac{\operatorname{atan2}\left(2 \cdot M_{11}, (M_{20} - M_{02})\right)}{2} \tag{5.2}$$



(a) Original Image

(b) Image after Thresholding with Remaining Noise

(c) The Image ROI

(d) The Resulting Edge Image

Fig. 5.1: The intermediate steps of `createEdgeImage`

and the eccentricity as

$$\begin{aligned} tmp &= \frac{\sqrt{4 \cdot M_{11}^2 + (M_{20} - M_{02}^2}}{2} \\ \lambda_1 &= \frac{M_{20} + M_{02}}{2} + tmp \\ \lambda_2 &= \frac{M_{20} + M_{02}}{2} - tmp \\ ecc &= \frac{\min \lambda_1, \lambda_2}{\max \lambda_1, \lambda_2} \end{aligned} \tag{5.3}$$

5.1.1.3 The Function `findLineCandidates`

Algorithm 5.4 summarizes the function. Input is the edge image, returned by the previous function. The edge segments in the edge image are extracted as explained in algorithm 5.1. The minimum size of connected pixels to be considered as an edge was set to 30.

Algorithm 5.4: Finding Line Candidates in a Edge Image

input : the binary image containing edges $edgeImg$,
the minimum length of a line to be considered λ_{min},
the maximal error of a line to be considered straight ϵ_{max}
output: the two line candidates l_1 and l_2

$L \longleftarrow \emptyset$;
$E \longleftarrow$ extractEdges($edgeImg$,λ_{min});
for $e \in E$ **do**
  $E \longleftarrow E - \{e\}$;
  $E \longleftarrow E \cup$ simplifyEdge(e,λ_{min});
end
for $e \in E$ **do**
  $l \longleftarrow$ createLineFrom(e);
  $\epsilon \longleftarrow$ squaredError(l,e);
  if $\epsilon < \epsilon_{max}$ ***or*** *(line is vertical/horizontal* ***and*** $\epsilon < 2 \cdot \epsilon_{max}$ **then**
    $L \longleftarrow L \cup \{l\}$;
  end
end
combine ambiguous lines in L;
$l_1 \longleftarrow$ line in L with highest support;
$l_2 \longleftarrow$ line with least angular difference to l_1;

Each segment is fed to the line simplifcation algorithm 5.2. Into every resulting edge segment a line is fitted into. For each line, the mean squared error to all edge pixels is calculated. The line is only considered a candidate, if the error is small enough (less than one pixel). An exception is given by the case of vertical or horizontal lines. Pixels of such lines tend to jitter leading to a greater error. Ambiguous lines are combined. In the end the line candidates are sorted by support. If only two line candidates are left or the support of the second candidate is significantly greater than the third line's support, the first and second candidate are returned. Else the line candidates are taken, which have less difference in orientation.

5.1.1.4 The Function `estimateCylinder`

The last function uses the line candidates to compute the cylinder's estimated orientation and position. It is described in algorithm 5.5. The algorithm requires the following inputs:

- Lines l_1 and l_2, the line candidates returned by the previous function

Algorithm 5.5: Pose Estimation of the Cylinder using the given Line Candidates

input : the two line candidates l_1 and l_2,
the *mean* of the cylinder projection,
the focal length f of the camera, which has been used,
the radius r of the cylinder

output: the cosines b corresponding to the cylinder's main axis,
a point M on the axis

$l_0 \longleftarrow$ lineBetween(l_1,l_2);
$p \longleftarrow$ point on line l_1;
$b \longleftarrow$ cosines3DfromParallelLines(l_1,l_2,f);
Calculate q as corresponding point to p;
Use equations 3.21 to 3.25 to determine point M;
Move M towards the mean of the cylinder;

- the computed mean used as indication for the cylinders midpoint
- radius r of the cylinder

Both, the line candidates as well as the mean, have to be shifted as they have originally been determined with respect to the ROI. The function follows the algorithm explained in section 3.3.4. As a first step, the line in between the line candidates is calculated. This line is the projection of the cylinder's main axis in the image. All lines are required in their point representation, in order to use equation 2.17. An arbitrary point on one of the cylinder's sides is chosen as basis for the next steps. To estimate the cylinder's midpoint, the point on the cylinder axis is chosen, which is closest to the mean. This point may differ greatly from the real midpoint in case of occlusions or be shifted in direction of the base of the cylinder, which is closer to the camera.

5.1.2 On Synthetic Images

The test of the OpenCV implementation had to yield two different results: at first the accuracy of the algorithm had to be evaluated, secondly its runtime had to be measured. To perform the tests, several scenarios were created with the *V-REP* robotic simulator. For the camera, a pixel resolution of 1280x720 px was assumed, which complies with the resolution of the *Odroid* webcam which was used to record the experimental data.

Before a consecutive series of images is fed to the algorithm, some single cases are examined in detail:

- a cylinder with main axis parallel to the image plane and oriented vertically
- the same cylinder as above, but rotated around the z-axis of the camera coordinate system
- the same cylinder in two arbitrary orientations,

- a manipulator model with coloured cylindrical part
- an example of an occluded cylinder, where the algorithm does not work
- a real cylindrically shaped object

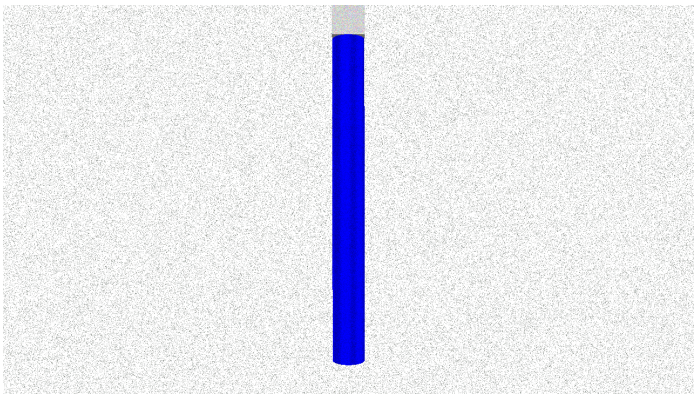
The results are shown in the following figures. For each example, the original image is displayed first. The red rectangle in the middle figure corresponds to the ROI. Each detected segment is coloured in a different colour. If the pose estimation was successful, a final image is seen, where the estimation is projected into the original image.

Figures 5.2 and 5.3 show the cylinders parallel to the image plane. In both cases, the estimation appears correct. The benefit of using a ROI is greater in the first example, where the cylinder almost completely fills in the bounding rectangle. However, also in the second example the problem space is obviously reduced.

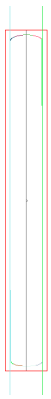
Figure 5.4 is an example of how the estimated center is shifted towards the closer end of the cylinder. In general, a compensation might be added. Figure 5.5 in contrast shows how occlusion additionally influences the estimated center. The occlusion is just enough to shift the projected cylinder's mean towards the original cylinder's center. Therefore compensation has been discarded, instead it is recommended to feed the estimation to a pose filter, which is able to account for sudden changes.

Figure 5.6 proves, that the algorithm may not only be executed on simple cylinders, but shows part of a manipulator. The chosen manipulator model is part of V-REPs model browser. Only its colour has been changed to a blue shade.

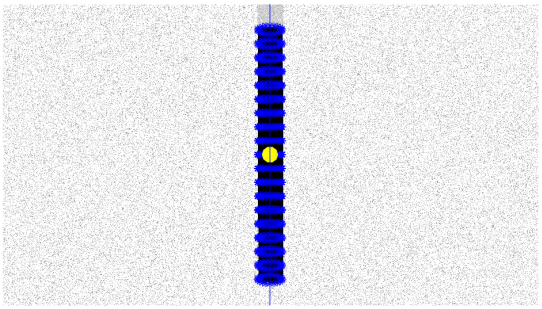
A case of failure is displayed in figure 5.7. The cylinder is far away and furthermore partially occluded. The algorithm is therefore not able to identify the line candidates.



(a) Original Image

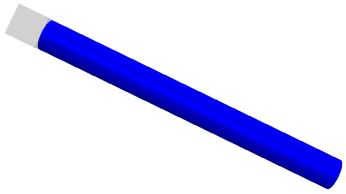


(b) Detected Lines, Segments and ROI (red)

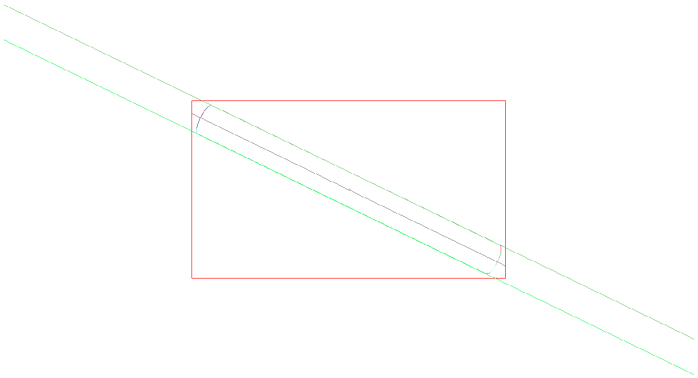


(c) The estimated Cylinder (blue stars) and its Centre (yellow dot)

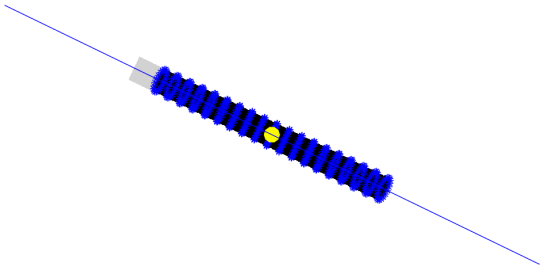
Fig. 5.2: Test Results for Example 1: Vertical Cylinder parallel to the Image Plane



(a) Original Image

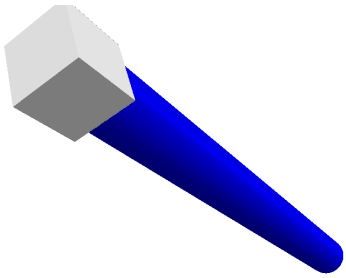


(b) Detected Lines, Segments and ROI (red)

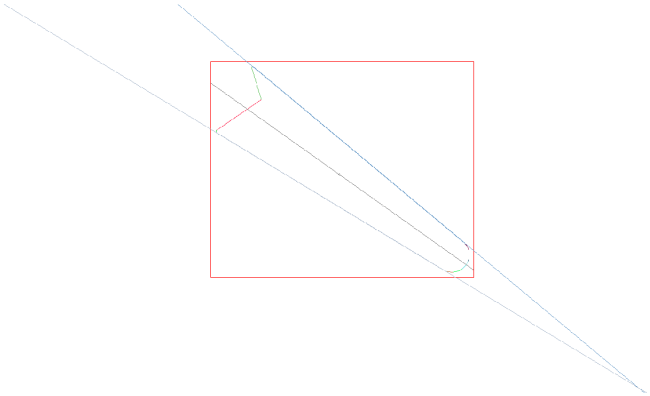


(c) The estimated Cylinder (blue stars) and its Centre (yellow dot)

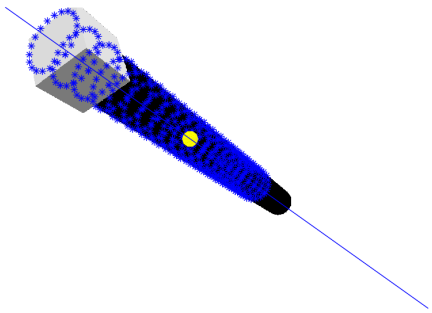
Fig. 5.3: Test Results for Example 2: Cylinder parallel to the Image Plane rotated around the z-Axis



(a) Original Image

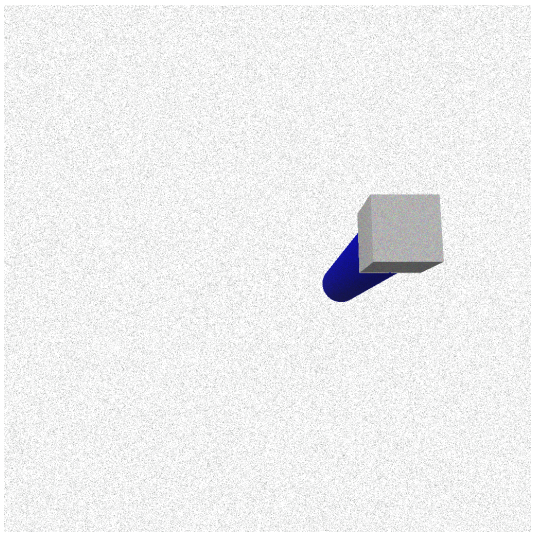


(b) Detected Lines, Segments and ROI (red)

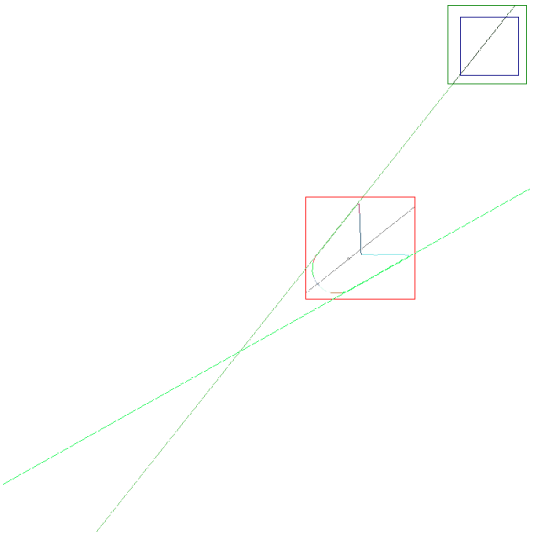


(c) The estimated Cylinder (blue stars) and its Centre (yellow dot)

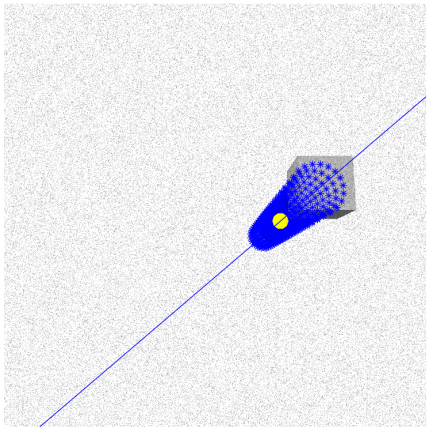
Fig. 5.4: Test Results for Example 3a: Cylinder with arbitrary Pose



(a) Original Image

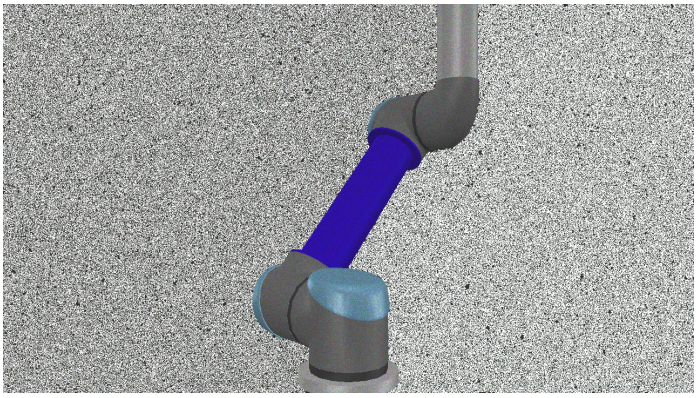


(b) Detected Lines, Segments and ROI (red)

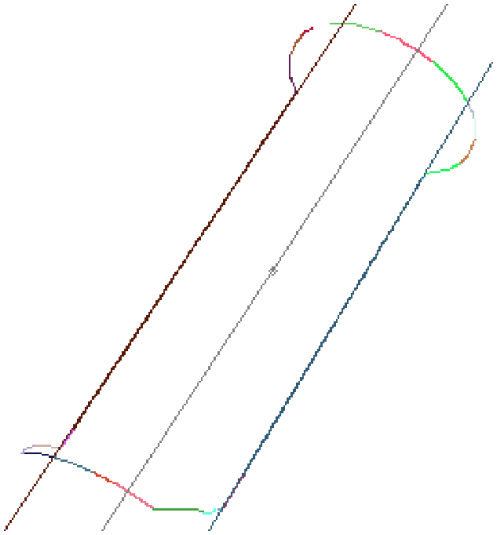


(c) The estimated Cylinder (blue stars) and its Centre (yellow dot)

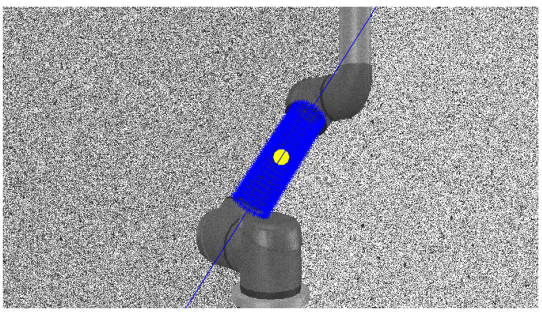
Fig. 5.5: Test Results for Example 3b: Cylinder with arbitrary Pose



(a) Original Image



(b) Detected Lines, Segments and ROI (red)



(c) The estimated Cylinder (blue stars) and its Centre (yellow dot)

Fig. 5.6: Test Results for Example 4: Manipulator Arm

(a) Original Image (b) Detected Segments and ROI (red)

Fig. 5.7: Test Results for Example 5: Occluded Cylinder with no Detection

The next test cases were executed on a consecutive series of images each. They are supposed to reveal the algorithm's accuracy. The result is presented in two groups of three diagrams each. The first group shows the components of the three dimensional position vector. The second group shows the components of the estimated main axis orientation. Both groups include a mean squared error. In case of the position, it has been computed as:

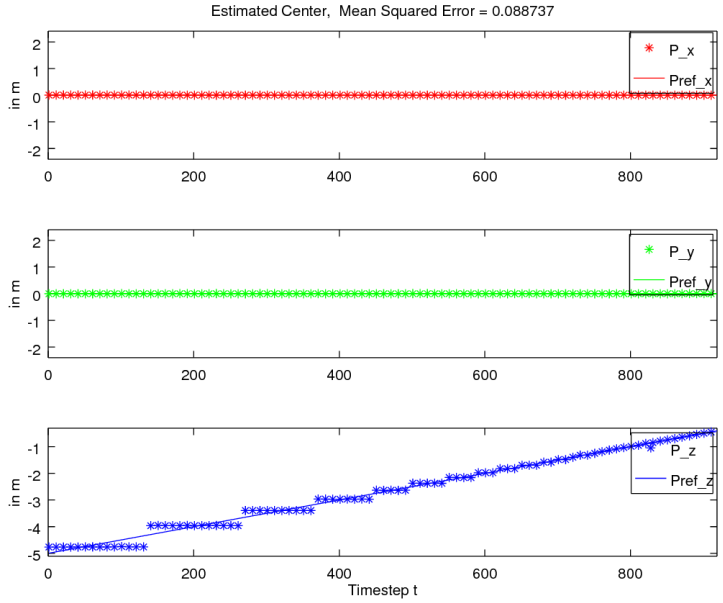
$$E = \frac{\sum_i^n \sqrt{\sum_{j=0}^{2} (P_{i,j} - P_{ref,i,j})^2}}{n} \tag{5.4}$$

where n is the number of points and $P_{i,j}$ is the j-th component of point P_i. Note, that a more negative z-component of the position means the cylinder is further away.

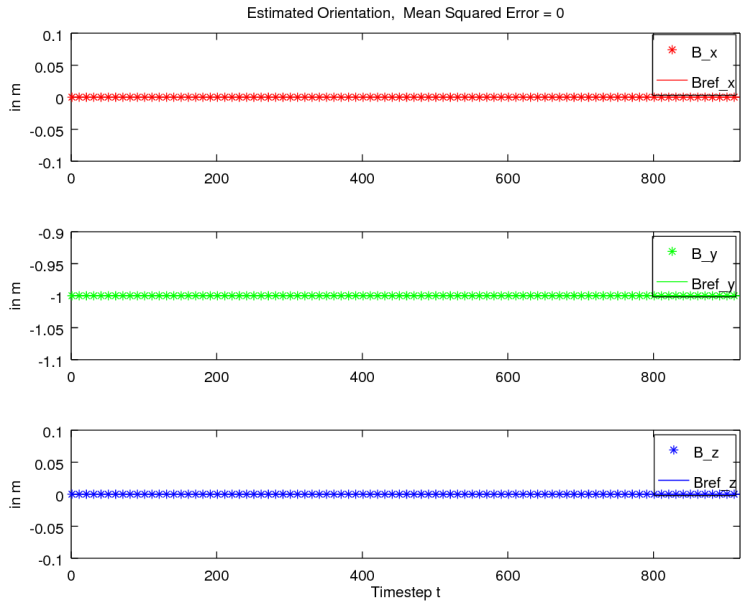
In the first scenario, the cylinder approaches with constant velocity of $0.01\,\mathrm{m\,s^{-1}}$ along the z-axis and fixed orientation, starting at a point of 5 m distance. In figure 5.8(a) it becomes apparent, that the distance (z-component) resembles a stair function. However, it is supposed increase linearly as the red line (ground truth) implies. The errors are a result of the limited image resolution. If the cylinder's main axis complies with the vertical or horizontal image axis, these errors are the highest.

The second scenario (figure 5.9), where the cylinder still approaches with constant velocity but its main axis is tilted by 45°, yields smoother results. The mean squared error amounts to 0.004 403 m. The orientation estimation is unsteady. It improves visibly, if the cylinder has approached to less than 2 m.

In the next scenario displayed in figure 5.10, the cylinders stays at a fixed position, but is rotated around the z-axis with fixed angular velocity of $0.01\,^\circ\,\mathrm{s^{-1}}$. In this case, position as well as orientation estimation are very accurate with errors in the range of millimeters. The influence of the shifted mean as seen in figure 5.4(c) becomes apparent during a combined rotation and translation motion in a distance of less than 1 meter. (figure 5.11). The positioning error is increased and now in the range of a centimeter. The difference between estimated position and reference increases with the ongoing change in orientation.

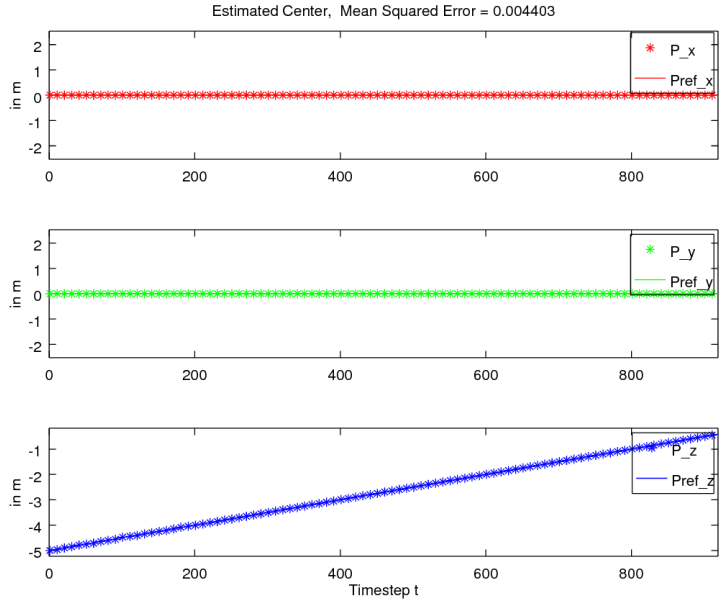


(a) Estimated Position and Reference

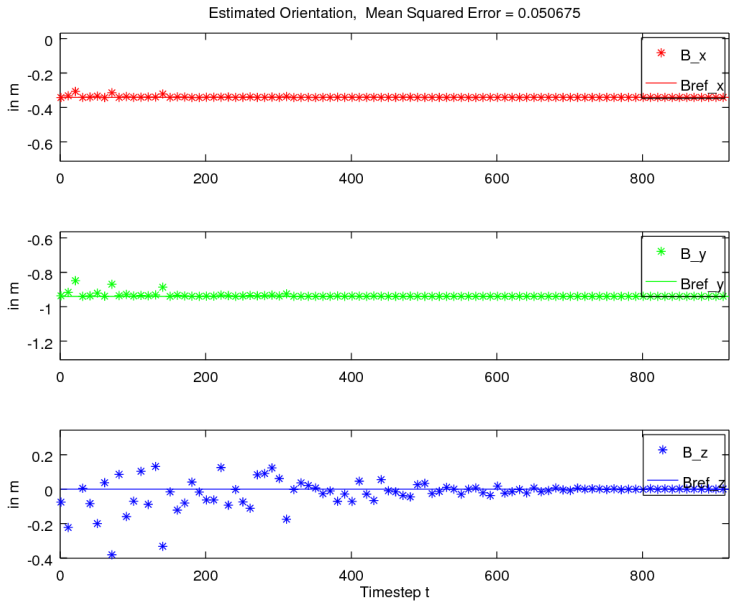


(b) Estimated Orientation and Reference

Fig. 5.8: Test Series 1: Vertical Cylinder with Constant Velocity

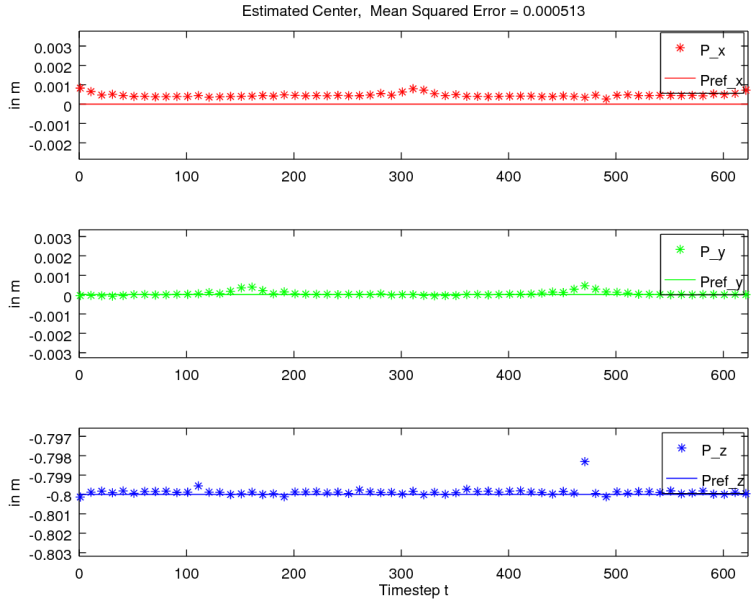


(a) Estimated Position and Reference

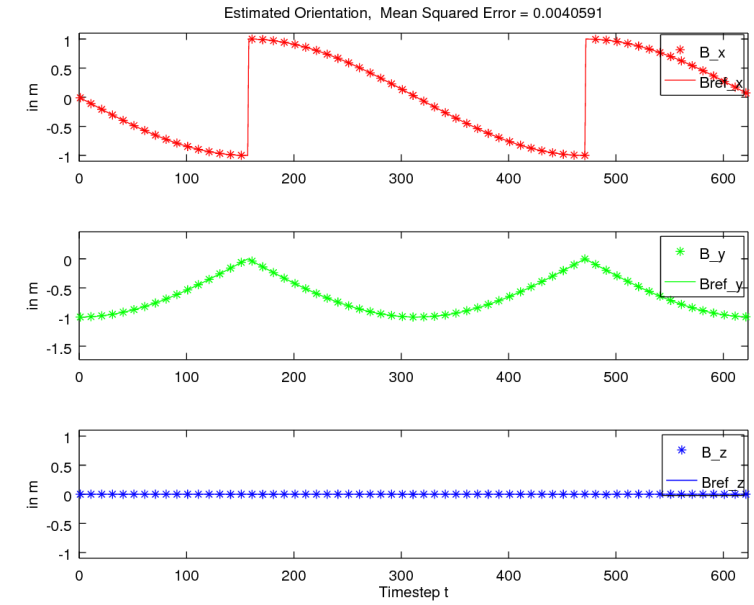


(b) Estimated Orientation and Reference

Fig. 5.9: Test Series 2: Tilted Cylinder with Constant Velocity

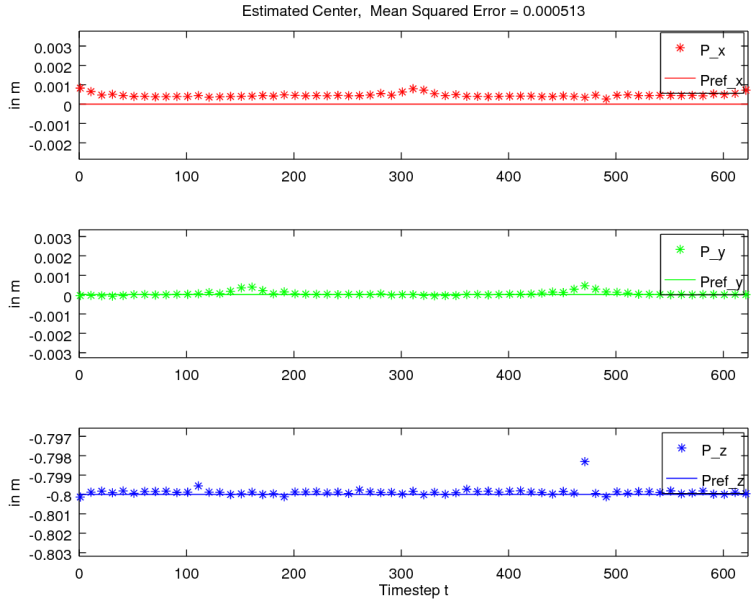


(a) Estimated Position and Reference

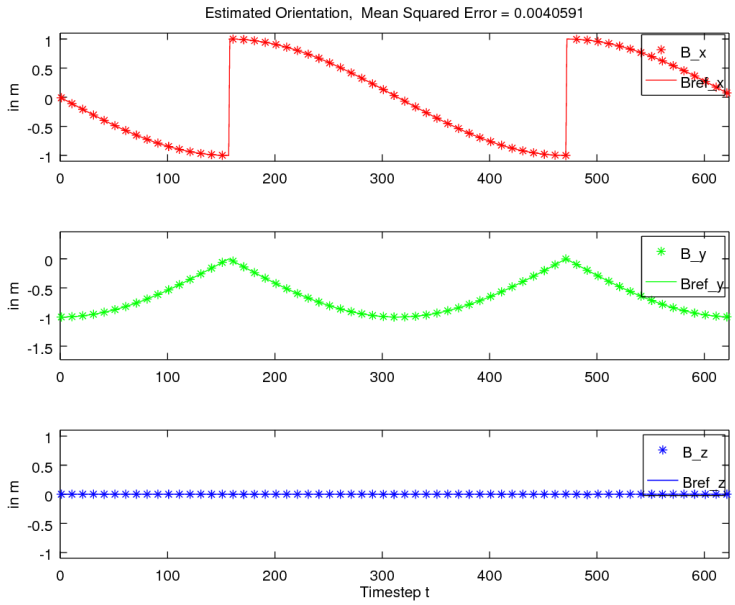


(b) Estimated Orientation and Reference

Fig. 5.10: Test Series 3: Rotation around Z-Axis



(a) Estimated Position and Reference



(b) Estimated Orientation and Reference

Fig. 5.11: Test Series 4: Combined Rotation and Translation

5.1.3 On Camera Images

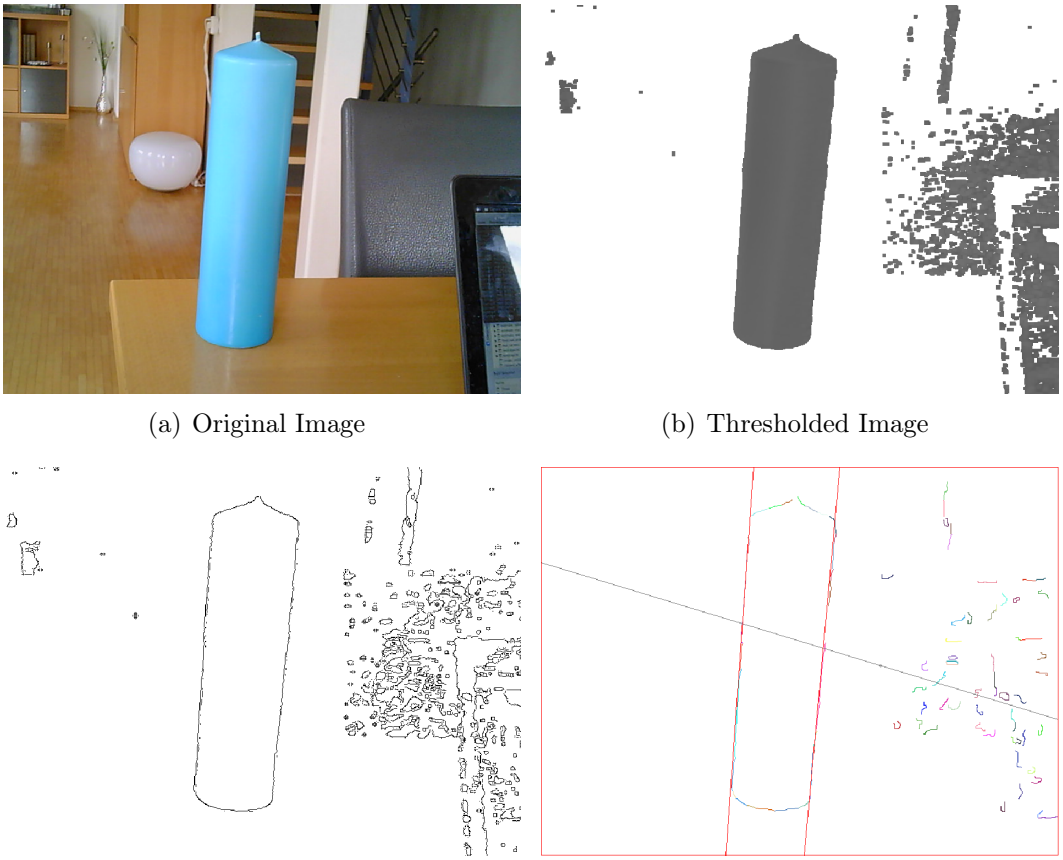
As soon as the algorithm was tested on real world images, it became clear, that the algorithm with the parameters chosen so far would not work. In the real images (for example as in figure 5.12(a)) two situations changed: not only the segmentation by hue did not prove to be perfectly successful (see figure 5.12(b)), but also the influence of lighting effects affected the Canny algorithm. The detected segments were a lot noisier than before (see figure 5.12(c)). Therefore the following parameter had to be adjusted: the minimal length of line segments to be considered was set to 40 pixels in order to eradicate false segments due to noise. The maximal error of lines also was increased to 3 pixels.

The `equalLines` function had to be adjusted as well. Two lines are now considered equal, if they differ not more than 0.1 rad in orientation, and 15 pixel in normal length. As seen in figure 5.12(d) the pre-estimated orientation cannot further be used to decide on line candidates. As a result, the algorithm is not working properly anymore in situations, where the cylinder sides are very short. Additionally, the lighting situation and scenery does not correspond to an OOS scenario. Still, the test shows how a variation of parameters allows to use the algorithm in different scenarios.

A series of images was recorded, using a robotic manipulator, which a camera was attached to. The cylindrically shaped object, which is a candle of 0.0395 m radius and 0.27 m length (excluding the pointy wick end), was placed on top of a metal table. However, the table colour resulted in similar hue values as the candle itself, for what reason a wooden plate was placed beneath the candle. In a lot of the recorded images, parts of the metallic table still interfere with the algorithm. The test results therefore were not optimal. Further tests with a cylindrical object of a different colour are proposed, but were not possible due to time constraints. The experiments were already delayed, as they originally were to be performed using a 3D printed cylinder, however the cylinder was not ready in time.

Three scenarios were considered. In the first and second scenario the manipulator starts above the candle. The distance between camera and metallic table was estimated by calibration with a checker board to account for 1.15 m. The focal length of the web cam, which was used, is 1215 pixel units. The first scenario is a pure translation, where the camera approaches the candle. The distance between start and end point accounts for 0.685 m. The manipulator movements were programmed to be linear movements. Figure 5.13 does not contain a reference, as it was not possible, to extract reference points and orientation. Due to the influence of the metallic table, the x- and y- values of the position estimation are prone to errors. Obviously, the values in general are fluctuating. In order to compare the results, the mean of the first fifty images and the mean of the last twenty images was calculated, omitting outliers(values which differed more than 0.1 m from the median). The z-value in the beginning is 1.09 m, differing by 0.02 m of reference distance (including the candle radius). In the end of the experiment, the z-value equals to 0.408 m. The difference between both values corresponds to 0.684 m and therefore to the known reference distance.

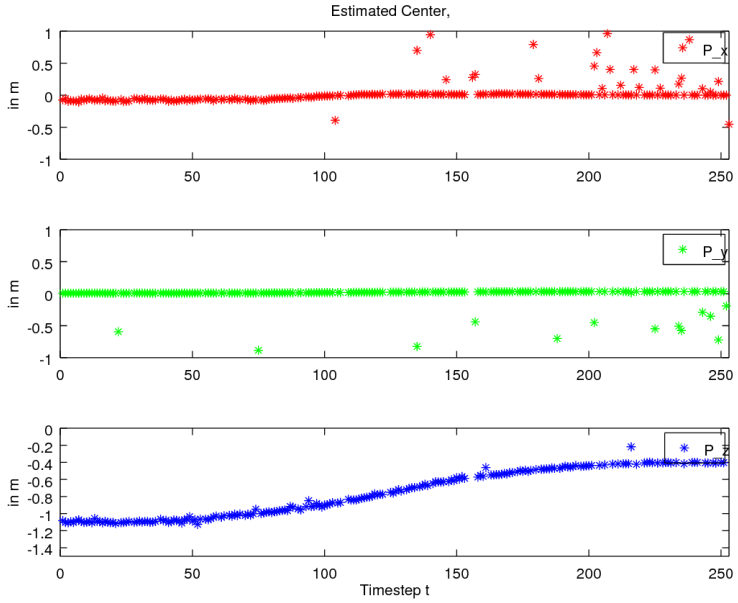
Figure 5.14 shows two examples of faulty line candidates. Both are the results of a failure in the edge detection. One of the cylinder sides is too fragmented to assign



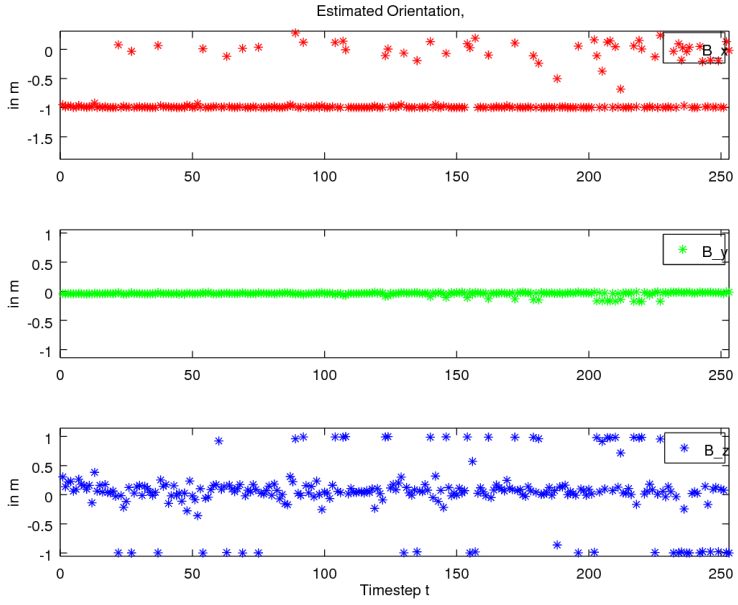
(a) Original Image (b) Thresholded Image

(c) Edge Map (d) Detected Segments and ROI (red)

Fig. 5.12: Test Results for Example 6: A cylindrically shaped object in an arbitrary environment

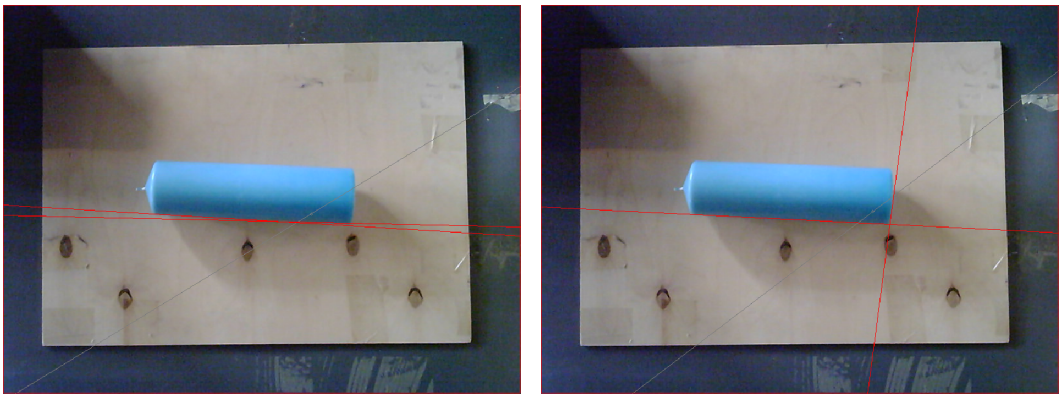


(a) Estimated Position



(b) Estimated Orientation

Fig. 5.13: Test Series 5: Real Scenario (Translation)



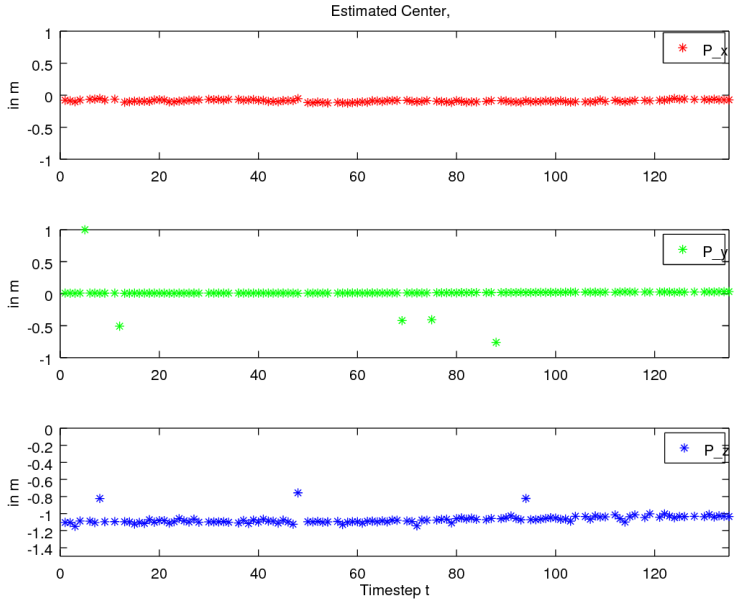
(a) Failures of this kind result in z-value outliers

(b) Failures of this kind result in orientation outliers

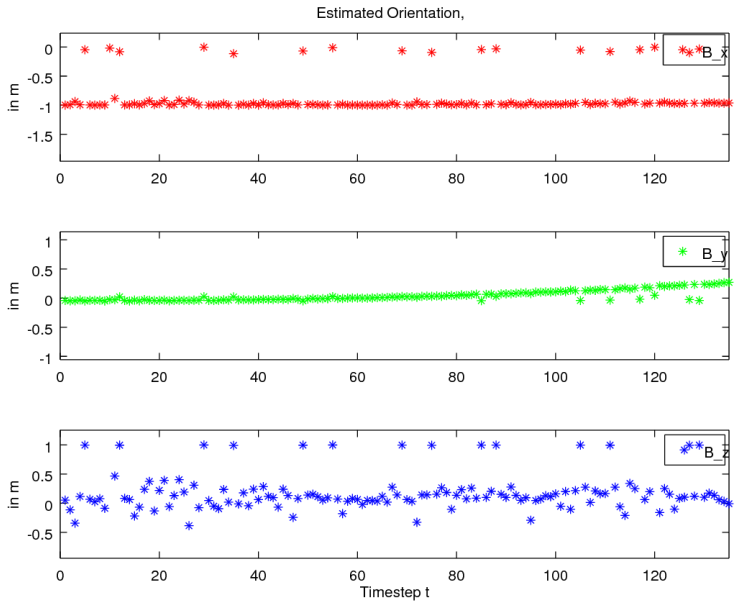
Fig. 5.14: Possible faulty line candidates

a line. In the first case, instead two line candidates are chosen which belong to the same cylinder side due to erroneous line fitting. The resulting position estimation is too far away. In the diagram these outliers are not depicted, as they would distort the actual diagram. In the second case, the orientation was wrongly measured. As seen in figure 5.13(b), this happens quite often (b_z equal to 1 or -1).

Figure 5.15 shows the results of the second scenario, a rotation in the x-y-plane. Here, also the fluctuating values of the orientation z-value are apparent. Even more in the results of the third scenario - a combined rotation and translation with results shown in figure 5.16. The amount of faulty estimation is so high, that it is not possible to decide, if an estimation is valid. Choosing instead single estimations in between, the resulting detected cylinders as seen in figure 5.17 look promising.

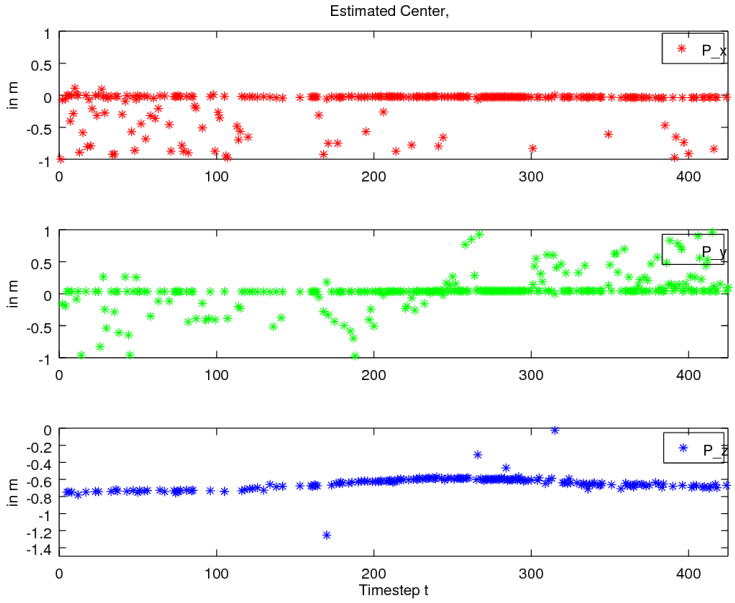


(a) Estimated Position

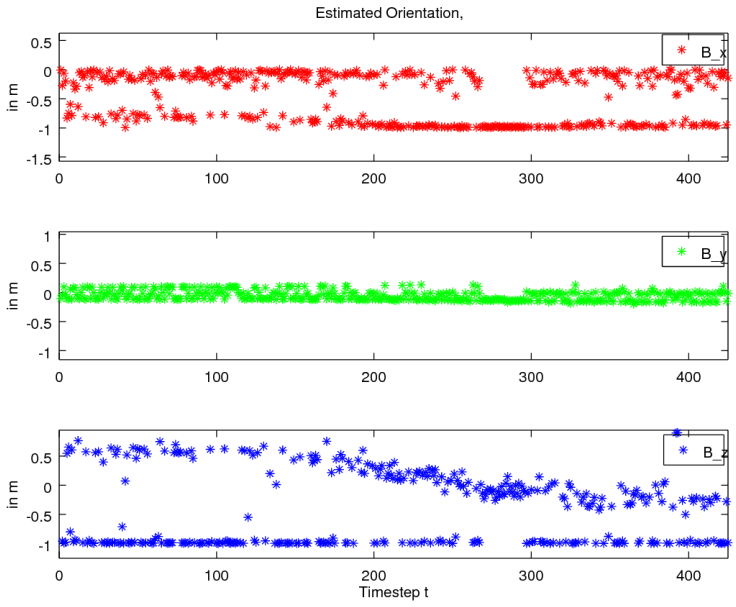


(b) Estimated Orientation

Fig. 5.15: Test Series 6: Real Scenario (Rotation)



(a) Estimated Position



(b) Estimated Orientation

Fig. 5.16: Test Series 7: Real Scenario (Rotation and Translation)

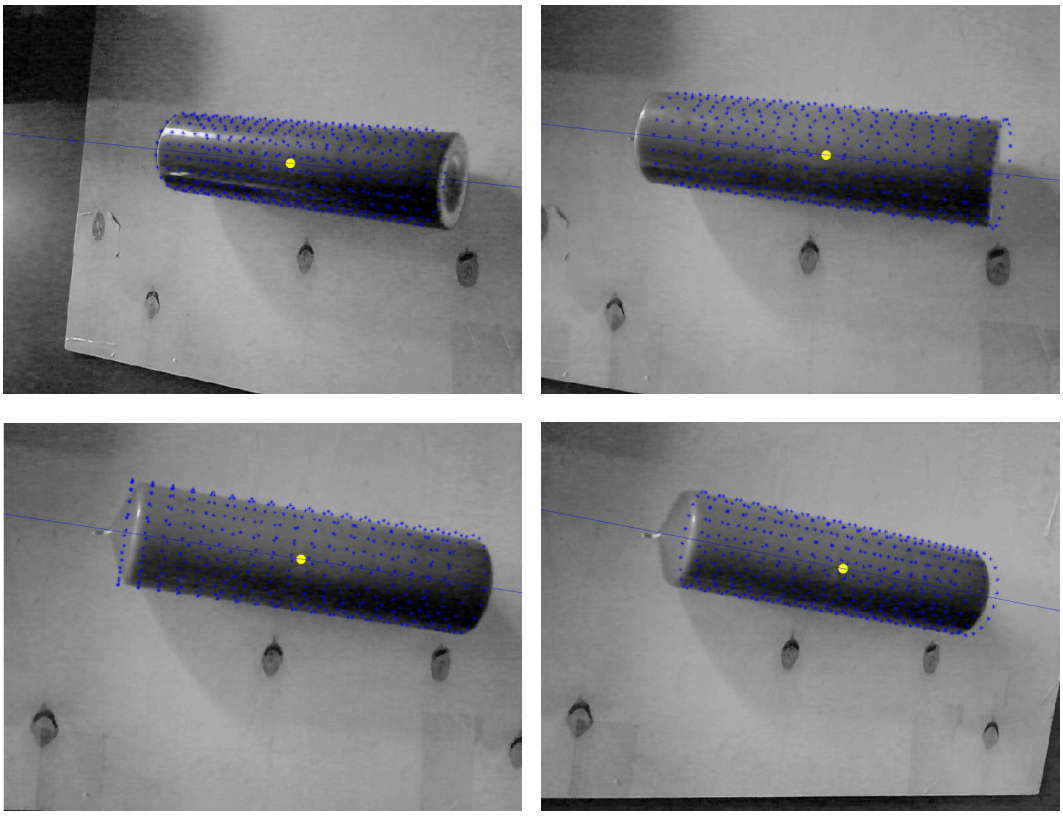


Fig. 5.17: Single Detected Cylinders of Test Series 7

5.1.4 Run Time Tests

The algorithm run time is the next property to be considered. All test scenarios have been benchmarked in order to determine, which part of the algorithm consumes the most time.

OpenCV is optimized, to work in parallel if possible. Figure 5.18 shows, how disabling optimization and parallelization influences the total runtime. The blue bars show the average runtime for the processing of a single frame using the default OpenCV optimization. The red bars behind show the average runtime when optimization is turned off. In the worst case, disabling optimization may almost double the average run time. The run time differences between scenarios are a result of the possible ROIs. The smallest average run time therefore corresponds to the translation scenario of the vertical cylinder, where the ROI is ideal.

The application was then executed on two different processors: at first a standard laptop was used, which includes an Intel Core^{TM} i5 processor with 4 cores running at 1.7 GHz each. The second machine is a UDOO single board computer with a ARM Cortex-A9 CPU, also having 4 cores but running at 1.0 GHz each. Figure 5.20 shows the increase of run-time running the non-optimized OpenCV application on the UDOO board. In both cases, the runs on the laptop needed only about 17°/$_{\circ}$ of the run-time on the UDOO board. In both runs it becomes apparent that the image processing is the highest run-time consumer. Figure 5.19 shows the result of the Intel core. The `createEdgeImage` function takes the major part. On the contrary, the final

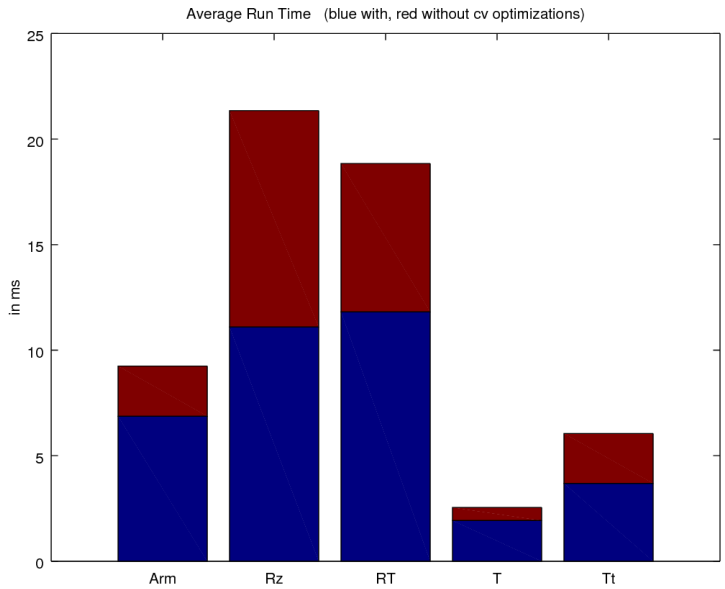


Fig. 5.18: A Run Time comparison showing the influence of OpenCV optimization and parallelization. The labels on the x-Axis denote the synthetic test scenario, which was used to gather the results. *Arm* stands for the manipulator scenario, *Rz* for the rotation around the z-axis, *RT* for the combined rotation and translation, and *T* for a translation. The small *t* denotes the tilted translation scenario.

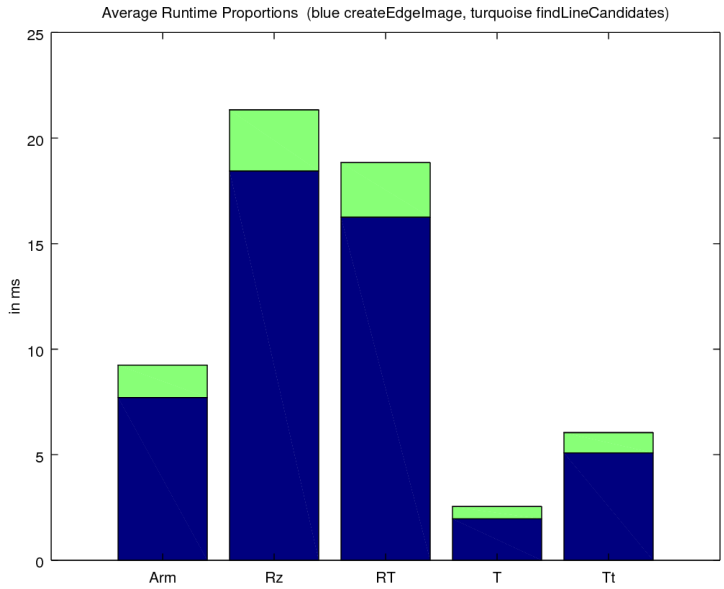


Fig. 5.19: A run time comparison showing which part of the algorithm accounts for what proportion of average run time. The bar sections are absolute. The blue part corresponds to `createEdgeImage`, the turquoise part to `findLineCandidates` . The run time of `estimateCylinder` is too small in relation in order to be displayed properly.

pose estimation using the determined line candidates is so fast in comparison, that it is not visible in the bar diagram.

5.1.5 Performance Analysis

The algorithm was tested using the OpenCV reference implementation . The first two tests cases examined properties of the algorithm itself, the third test case investigated the implementation specific runtime. On synthetic images the pose estimation is working correctly up to some flaws for which in the context of this thesis no solution has been sought for. The discretization error might be overcome by using a swivel-mounted camera as sensor. In case of a steady vertical or horizontal orientation, the camera may then be moved to a less error-prone position. To account for the shifted sensor and deviations in general, the usage of a filter (e.g. a Kalman filter) is recommended. Predictions of the filter also suggest the possibility to bypass estimation failures due to occlusion.

The test scenarios with camera images were suboptimally chosen: the scenario did not fully correspond to the chosen prerequisites and the experimental environment imposed further problems. The additional image regions with similar hue also prevented image reduction to a region of interest, severely increasing the run time (e.g. in test scenario 6 to almost 100 ms).

Considering the runtime requirements, as assumed the image processing part con-

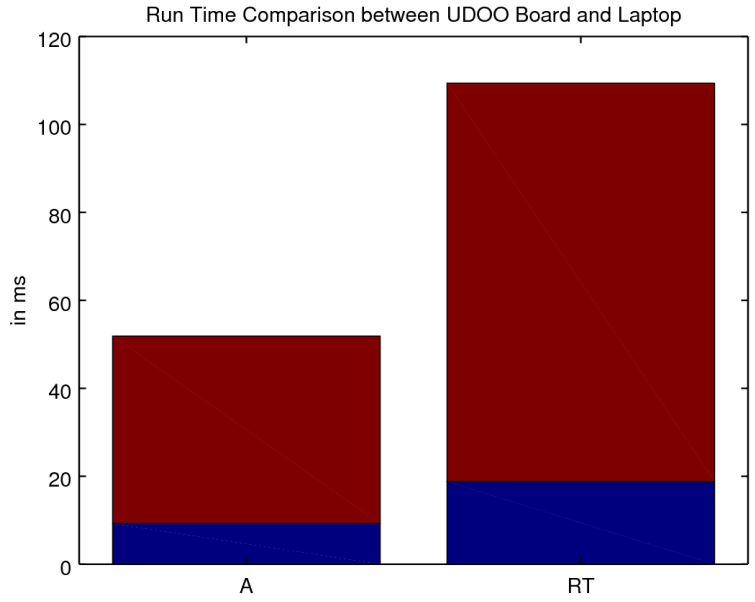


Fig. 5.20: A run time comparison showing the slow down when using the UDOO board. The blue part corresponds to the average runtime per frame on the laptop, the red on the UDOO.

stitutes the most time-consuming activity and is therefore the first indication to look for optimization opportunities. Both, the colour conversion as well as the segmentation, are point operators and may therefore also be executed in a parallel or in a distributed way. The edge detection could also be distributed. However, the Canny operator is complex and it is assumed, that an implementation, which does not exploit all processor capabilities, becomes much slower than the optimized OpenCV implementation of the Canny algorithm. The slow-down using the UDOO board, which still has a capable processor, show how distribution may be needed, to be able to execute the algorithm in real time. However, the effect of using the OpenCV optimization shows that distribution is likely to improve the run-time.

5.2 Distributed Implementation with RODOS

5.2.1 Architecture

The distributed implementation follows one interest. The lack of single processor computing power shall be compensated by distributing parts of the algorithm to available general purpose processors, in order to speed up the pose estimation. As the distribution of huge amounts of image data also consumes time, a trade-off has to be found, where the distributed execution is faster than the execution on a single core processor.

Exemplary, only the parts identified in the previous section are outsourced and executed in a distributed way. The remaining application is adopted as is. The goal

of this section is therefore to design a program architecture, which fulfils the following requirements:

- Identification of available computing nodes
- Slicing and reassembling of the image
- Distribution to the identified computing nodes
- Correct execution of the task itself

To some of these requirements the solution is not obvious and influenced by the given circumstances and desired result. It was tried to find a good understanding between a realistic scenario and a sufficiently comprehensible implementation. In the scenario described in the introduction of this thesis, the thought of a distributed system in space goes as far as to include not only the computing nodes of a single system but also of nearby, maybe idle spacecraft. As a result the network of nodes is versatile and not all nodes are known. Additionally, nodes may become unavailable if they leave the network or are occupied with other tasks.

A useful slicing of the image data may also depend on the capabilities of the nodes. Nodes with higher computational power process more data than small micro controllers. Some may not even have the resources to store the data needed. Either the distributing node or the recipient of a request has to decide, if it is qualified enough to adopt the proposed task. The assumption, that all data returns in the same order as sent out, is also invalid due to differing capabilities of available nodes. The distributor is therefore in charge to manage the reassembly of the image, before it is processed further.

In order to consider the versatile availability of nodes, the distributed application is implemented based on a broker architecture. Theoretically the role of the broker may be filled in by any free node running RODOS. Subsequently the broker task is adopted by only one node, though. The broker does not have to keep track of available nodes in the network. Instead a lose request-answer-protocol is employed: the broker sends requests to the network and distributes data to any node, which is answering that request.

The decision of in what size and to whom the image data is distributed, may be arbitrarily complicated. One possibility is to wait, until the entire image has been received by the broker. The broker then sends a single request, awaits the answers and slices the image data according to those. The broker has to decide, how much data each of the potential worker receives. A simpler concept with more communication overhead is to break down the image to messages of a fixed size and send a request for each message. The message may then be sent to the node whichever answers first. The advantage of the latter concept is, that faster nodes automatically get more messages than slow nodes. In order to keep the implementation simple, the second concept is used in the design.

The messages have an identifier, so the distributor is able to keep track of the messages it has sent and received. In case one of the messages gets lost, the distributor may resent the request. Summarized, the application consists of a distributor node

and an arbitrarily number of workers. For the distributor, it does not matter where a worker node resides. The distributor node receives the image byte-wise and slices it into messages of fixed-size. Requests are sent for every message. An available worker answers to the request. The distributor sends the message to the sender of the first answer, it receives. The worker is now responsible for performing its required task and sending back the processed bytes. The distributor collects the messages and sends the image data back in the correct order.

The subsequent sections explain the implementation of the nodes in detail and present the test results. The goal of this chapter is to determine how the processing rate changes in comparison to the OpenCV implementation. Also, the speed-up of using multiple computing nodes is measured. The distributed application is first run on a multi-core processor where every core has the same capabilities. Then also nodes with lower computing power, which in general were not able to execute the algorithm alone, are included into the network.

5.2.2 Implementation

The implementation of the distributed application consists of several parts. The distributor and the worker nodes are implemented as RODOS applications. In order to compare the runtime, an OpenCV application is used which first computes the hue and thresholding locally and then sends the image to the distributor. The time until the complete image is returned is measured. Sockets are used to communicate between distributor and the OpenCV application through UDP. In general, using TCP instead is recommended, as the latter is a streaming protocol which takes care of correct order and lost messages, while UDP is a datagram protocol.

Workers and distributor share a common header file, which defines the topics for messages and requests. Gateways are used to distribute the topics between computing nodes. Nodes receive their messages using UDP. The following constants are defined in the header file:

- `MESSAGE_SIZE` is the size of UDP messages received by the OpenCV application
- `MAX_DATA` is the maximum data size, to be distributed between RODOS nodes
- `OFFSET` is the number of bytes reserved at the beginning of UDP messages
- `MIN_THRESH` and `MAX_THRESH` are the thresholds used by the worker nodes

Messages to worker nodes are distributed over the `messages` topic, using a `Message` datatype. It consists of a header and a data array of size `MAX_DATA`. The header contains a stamp, the IDs of sender and receiver as well as the actual length of data, sent with this message. The messages are returned using the `returnedMessages` topic. Requests are distributed over the `requests` topic. A `Request` is similar to a `Message` but without the length and data attribute. A broadcast request has sender id 0.

The OpenCV application copies the image data into byte arrays of fixed length. The first two bytes are reserved for an identifier in order to provide an internal order. The next two bytes are reserved for the actual data length. The image data has to be aligned in packets of three bytes to account for the B G R components of a pixel.

The distributor application includes four active objects. The subscriber `UdpReceiver` is responsible for the communication with the OpenCV application. It slices the received data packets into chunks of maximum data size and adds the message header. The messages are then enqueued to a fifo. The `RequestSender` thread regularly checks the fifo and sends a `Request` for each message. The request is stamped using the current time in nanoseconds. The subscriber `RequestHandler` processes the answers of the worker nodes. If an answer is received, it dequeues a message and forwards it to the sender of the answer. In order to make sure that requests are only answered once, the handler keeps track of the time stamp of the latest answered request. Only request answers are considered, which time stamp is greater than that. The subscriber `DataSub` to the message topic awaits the return messages. The message references are inserted into a vector. If as many messages have been received as had been sent, the references are sorted and sent back to the OpenCV application. This strategy ensures, that no message is skipped in between.

The worker applications only consist of two active objects. The `RequestHandler` subscribes to the requests and returns them in case the worker is idle. The `MessageHandler` listens for messages having the same id as the worker application. It then creates a message and converts the sent bytes into thresholded hue values. The optimized code to calculate the hue is shown in listing 5.1. It was deliberately decided to print its C++ implementation here as it was chosen due to its optimizations for the C++ language.

Listing 5.1: An optimized algorithm to compute hue

```
1  // returns H/2 to fit into uchar [0,179]
2  uchar convertToHue(uchar r, uchar g, uchar b){
3      float K = 0.f;
4      float max(255.0f);
5      if (g < b)
6      {
7          std::swap(g, b);
8          K = -1.f;
9      }
10
11     if (r < g)
12     {
13         std::swap(r, g);
14         K = -2.f / 6.f - K;
15     }
16
17     float chroma = float(r - std::min(g, b))/max;
18     return uchar(180* fabs(K + (float(g - b)/max) / (6.f *
           chroma + 1e-20f)));
19 }
```

5.2.3 Tests

The distributed application has only been tested using single images. An example image of 640 columns and 400 rows however, had an execution time of almost three seconds with no other application running. As the application was only executed on the laptop, where only four processing cores are available and one of them is already blocked by the OpenCV process, only up to three workers could be added. They did however not improve the run-time severely.

Measuring the time between sending out a message to a worker and receiving its answer by comparing their stamp with the current system time, showed that in general a message of maximum message size was processed in approximately 200 microseconds. A total amount of about 600 messages had to be sent and processed, in order to have the complete image covered, accumulating to a time of about 120 milliseconds.

It is possible, that the usage of a single topic severely slowed down processing, as each message had to be distributed over the network to all workers, not only to the actual recipient of the message. For future tests, it is recommended to rethink the usage of RODOS topics for distribution of messages and also to refrain from UDP, as sometimes a bundle of messages got lost.

Execution of the distributed application on the UDOO board was not possible so far, as there were problems of building the project for the UDOO processor. It is assumed, that a customization of existing build scripts will allow building for the UDOO board. The next step would be the usage of a even less capable processor, e.g. a microcontroller.

Chapter 6

Results and Conclusion

In the present thesis, an algorithm has been designed to estimate the pose of a cylindrical object. Motivation behind this algorithm has been the usage in an OOS scenario, where the cylindrical object is part of a robotic manipulator. The design and especially the tests of this algorithm have shown the difficulties with the given problem: at first it quickly emerged, that a cylinder is a rather feature-less object, especially considering occlusion, and that it is difficult, to identify a cylinder in arbitrary images without knowledge of any further objects.

The algorithm relies on identification of the two lines, which correspond to the cylinder's sides. The knowledge of their parallelism and orthogonal distance is used to solve the inverse perspective problem. The pose of the cylindrical object is estimated as the direction vector of its main axis and a point on this axis. The accuracy tests on synthetic images yielded correct results, up to expected errors due to limited image resolution.

When the test sequences of the real test were recorded, it became clear, that in the given surroundings it is not possible to simulate a realistic space environment. The tests running with the recorded images therefore produced results worse than expected based on the synthetic test scenarios. Still, the results reveal that the assumptions made in the beginning did not hold in reality. As the pose estimation algorithms introduced in the first chapter were partially tailored to a special case (communication satellites with rectangular sides, cylindrically shaped satellite), further knowledge about the satellite, where the manipulator is attached to, might also be helpful to remove or exclude objects which disturb the detection of the cylinder, but will also reduce the algorithm to a single use case.

The distribution aspect of this thesis has fallen short. Although a distributed implementation has been created, the *true* distribution was not fully manageable due to issues with the used tool-chain. The actual tests using the distributed implementation therefore are pending. In general, this thesis has shown, what has been already known: image processing, especially point operators, consume a lot of time. Distribution might offer a way out, which has to be investigated further on.

List of Figures

Algorithms

Listings

Bibliography

[1] Larry Matthies, Mark Maimone, Andrew Johnson, Yang Cheng, Reg Willson, Carlos Villalpando, Steve Goldberg, Andres Huertas, Andrew Stein, and Anelia Angelova. **Computer vision on mars**. *International Journal of Computer Vision*, 75(1):67–92, mar 2007. URL: `http://dx.doi.org/10.1007/s11263-007-0046-z`.

[2] European Space Agency (ESA). **Exomars rover**. http://exploration.esa.int/mars/45084-exomars-rover/, 2015.

[3] Peter Yuen, Yang Gao, Andrew Griffiths, Andrew Coates, Jan-Peter Muller, Alan Smith, Dave Walton, Craig Leff, Barry Hancock, and Dongjoe Shin. **ExoMars rover PanCam: Autonomous & computational intelligence [application notes]**. *IEEE Computational Intelligence Magazine*, 8(4):52–61, nov 2013. URL: `http://dx.doi.org/10.1109/MCI.2013.2279561`.

[4] *IEEE IROS Workshop on Robot Vision for Space Applications*, 2005.

[5] Stephane Ruel, Tim Luu, and Andrew Berube. **On-orbit testing of targetless tridar 3d rendezvous and docking sensor**. In *Proc of the International Symposium on Artificial Intelligent, Robotics and Automation in Space (i-SAIRAS 2010)*, 2010.

[6] Andrew Ogilvie, Justin Allport, Michael Hannah, and John Lymer. **Autonomous satellite servicing using the orbital express demonstration manipulator system**. In *Proc. of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS'08)*, pages 25–29, 2008.

[7] Japan Aerospace eXploration Agency (JAXA). **Hayabusa - a technology demonstration for sample and return**. Brochure, 2016. URL: `http://www.telespazio.it/docs/brodoc/GCC_eng.pdf`.

[8] Brent E. Tweddle, A. Saenz-Otero, and D.W. Miller. **The SPHERES VERTIGO goggles: Vision based mapping and localization onboard the international space station**. In *i-SAIRAS International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2012.

[9] M.D. Lichter and S. Dubowsky. **State, shape, and parameter estimation of space objects from range images**. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA 04. 2004.* Institute of Electrical and Electronics Engineers (IEEE), 2004. URL: `http://dx.doi.org/10.1109/ROBOT.2004.1307513`.

[10] J.M. Kelsey, J. Byrne, M. Cosgrove, S. Seereeram, and R.K. Mehra. **Vision-based relative pose estimation for autonomous rendezvous and docking**. In *Proceedings, 2006 IEEE Aerospace Conference*. Institute of Electrical and Electronics Engineers (IEEE), 2006. URL: http://dx.doi.org/10.1109/AERO.2006.1655916.

[11] Xiaodong Du, Bin Liang, Wenfu Xu, and Yue Qiu. **Pose measurement of large non-cooperative satellite based on collaborative cameras**. *Acta Astronautica*, 68(11-12):2047–2065, jun 2011. URL: http://dx.doi.org/10.1016/j.actaastro.2010.10.021.

[12] Haidong Hu, Xiaoyan Mao, Zhibin Zhu, Chunling Wei, and Yingzi He. **A vision system for autonomous satellite grapple with attitude thruster**. In *Proceedings of the 11th International Conference on Informatics in Control, Automation and Robotics*. Scitepress, 2014. URL: http://dx.doi.org/10.5220/0005020203330337.

[13] Chang Liu and Weiduo Hu. **Relative pose estimation for cylinder-shaped spacecrafts using single image**. *IEEE Transactions on Aerospace and Electronic Systems*, 50(4):3036–3056, oct 2014. URL: http://dx.doi.org/10.1109/TAES.2014.120757.

[14] Chang Liu and Weiduo Hu. **Effective method for ellipse extraction and integration for spacecraft images**. *Optical Engineering*, 52(5):057002, may 2013. URL: https://doi.org/10.1117%2F1.oe.52.5.057002.

[15] Florian Kempf, Alexander Hilgarth, Ali Kheirkhah, Tobias Mikschl, Tristan Tzschichholz, Sergio Montenegro, and Klaus Schilling. **Reliable networked distributed on-board data handling using a modular approach with heterogeneus components**. In *4S Symposium*, 2014.

[16] Klaus Schilling. **Perspectives for miniaturized, distributed, networked cooperating systems for space exploration**. *Robotics and Autonomous Systems*, oct 2016. URL: http://dx.doi.org/10.1016/j.robot.2016.10.007.

[17] Roberto Tron and Rene Vidal. **Distributed computer vision algorithms**. *IEEE Signal Processing Magazine*, 28(3):32–45, may 2011. URL: http://dx.doi.org/10.1109/MSP.2011.940399.

[18] Richard J. Radke. **A survey of distributed computer vision algorithms**. In *Handbook of Ambient Intelligence and Smart Environments*, pages 35–55. Springer Science + Business Media, 2010. URL: http://dx.doi.org/10.1007/978-0-387-93808-0_2.

[19] Richard Szeliski. *Computer Vision*. Springer London, 2011. URL: http://dx.doi.org/10.1007/978-1-84882-935-0.

[20] R.M. Haralick. **Monocular vision using inverse perspective projection geometry: analytic relations**. In *Proceedings CVPR '89: IEEE Computer*

Society Conference on Computer Vision and Pattern Recognition. Institute of Electrical and Electronics Engineers (IEEE), 1989. URL: http://dx.doi.org/10.1109/CVPR.1989.37874.

[21] R. B. Fisher, K. Dawson-Howe, A. Fitzgibbon, C. Robertson, and E. Trucco, editors. *Dictionary of Computer Vision and Image Processing*. Wiley-Blackwell, jun 2006. URL: https://doi.org/10.1002%2F0470016302.

[22] J. R. Parker. *Algorithms for image processing and computer vision*. Wiley Pub, Indianapolis, Ind, 2011.

[23] Lutz Priese. *Computer Vision*. Springer Berlin Heidelberg, 2015. URL: https://doi.org/10.1007%2F978-3-662-45129-8.

[24] Reinhard Klette. *Concise Computer Vision*. Springer London, 2014. URL: https://doi.org/10.1007%2F978-1-4471-6320-6.

[25] Irwin Edward Sobel. *Camera Models and Machine Perception*. PhD thesis, Stanford, CA, USA, 1970. AAI7102831.

[26] John Canny. **A computational approach to edge detection**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, nov 1986. URL: https://doi.org/10.1109%2Ftpami.1986.4767851.

[27] Alexander Schill and Thomas Springer. *Verteilte Systeme*. Springer Berlin Heidelberg, 2012. URL: https://doi.org/10.1007%2F978-3-642-25796-4.

[28] Andrew D. Birrell and Bruce Jay Nelson. **Implementing remote procedure calls**. *ACM Transactions on Computer Systems*, 2(1):39–59, feb 1984. URL: https://doi.org/10.1145%2F2080.357392.

[29] Urs Ramer. **An iterative procedure for the polygonal approximation of plane curves**. *Computer Graphics and Image Processing*, 1(3):244–256, nov 1972. URL: https://doi.org/10.1016%2Fs0146-664x%2872%2980017-0.

[30] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. John Wiley & Sons, New York, 5 edition, 2015.

[31] A.W. Fitzgibbon, M. Pilu, and R.B. Fisher. **Direct least squares fitting of ellipses**. In *Proceedings of 13th International Conference on Pattern Recognition*. Institute of Electrical and Electronics Engineers (IEEE), 1996. URL: http://dx.doi.org/10.1109/icpr.1996.546029.

[32] E. Rohmer, S. P. N. Singh, and M. Freese. **V-rep: a versatile and scalable robot simulation framework**. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.