NTNU
Det skapende universitet

Bruk av robotsyn for griping av et objekt i bevegelse

Jens Arne K Engesæter

Undervannsteknologi
Innlevert: juni 2015
Hovedveileder: Olav Egeland, IPK

Norges teknisk-naturvitenskapelige universitet
Institutt for produksjons- og kvalitetsteknikk

NTNU – Trondheim
Norwegian University of
Science and Technology

Robotic Vision for Grasping of an Object in Motion

Jens Arne Krakhella Engesæter

Juni 2015

Master Thesis
Department of Production and Quality Engineering
Norwegian University of Science and Technology

Supervisor 1: Professor Olav Egeland

NTNU
Fakultet for ingeniørvitenskap og teknologi
Institutt for produksjons- og kvalitetsteknikk

Vår dato 22.01.2015 Vår referanse OEG

MASTEROPPGAVE VÅREN 2015

Jens Arne Engesæter

Tittel: Bruk av robotsyn for griping av et objekt i bevegelse.

Tittel (engelsk): Robotic vision for grasping of an object in motion

Oppgavens tekst:
Ved følging av objekter i bevegelse er det viktig at det benyttes algoritmer som kan kjøres med tilstrekkelig høy oppdateringsrate, som gjerne er kameraets bildefrekvens gitt av bilder pr. sekund. I denne oppgave skal metodene SIFT og SURF benyttes for å følge et objekt.

1. Presenter metodene SIFT og SURF.
2. Implementer SIFT og SURF for bildestrøm i sann tid fra bevegelige objekter. Prøv ut metodene og diskuter beregningstid og robusthet.
3. Implementer en eksperimentoppstilling hvor en Agilus robot styres fra et kamerasystem ved bruk av SIFT og SURF.
4. Prøv ut griping av et bevegelig objekt ved bruk av robotsyn.

Oppgave utlevert: 2015-01-22

Innlevering: 2015-06-10

Utført ved Institutt for produksjons- og kvalitetsteknikk

Trondheim, 2015-01-22

Olav Egeland

Professor Olav Egeland
Faglærer

Preface

This is a concluding Master's thesis in study of program Subsea Technology at NTNU. The work was carried out between January and June 2015.

The interest of computer vision started whit the course "Robotics" lectured by professor Olav Egeland, and the opportunity to implement computer vision in a system of an actual robot was very tempting. This resulted in the topic of "Robotic Vision for Grasping an Object in Motion" for this Final Master's thesis.

Trondheim, 2012-12-16

Jens Arne Krakhella Engesaeter

Acknowledgment

I would like to give a special thanks to PhD candidate Adam Leon Kleppe for his cooperation and help during the project. He has been a good adviser and his involvement in the work has been very helpful in selection of the right tools, software and tutorials for the experiments. He has also implemented the communication utilities between the KUKA Agilus Controller and the Ubuntu computer, which is an essential part of the setup.

I would like to thank professor Olav Egeland for his supervising, inputs and always having the office open for students with questions. I am very grateful for the opportunity to both have a practical and theoretical work during the project, and also be able to implement my theoretical work on an actual robot.

The workshop employees at the lab have been very helpful, and I would like to thank them for been available for questions when advices and necessary equipments were needed.

Finally, I would like to thank my good friends and fellow students at the Subsea Technology program for good cooperation and two great years together.

(JAKE)

Summary

The main objective of this project was to study grasping of an object in motion by using a combination of computer vision and a manipulator. The two tracking methods SIFT and SURF were compared, and implemented as a part of a visual system in the robotic lab at the Department of Production and Quality Engineering.

SIFT and SURF are both key-point detectors, but based on different mathematical background and structure. SURF is claimed to be faster than SIFT, and both are supposed to be scale and rotation invariant. To investigate the performance of the two methods, different experiments were done to measure robustness and computation time. In all the experiments, a 2D camera was used to observe a particular object moving with a constant velocity along a constant path. Different tags were attached to the object, and the methods were implemented with a range of parameters to study resulting performance.

To track an object in motion and predict a position for grasping an object, images have to be processed as close to real time as possible. Therefor it was important to be familiar with the behavior of the methods and optimize the implementations for computational efficiency. To understand which information is important in a captured image, fundamental knowledge about camera, transformations and image processing are presented in the report. In order to obtain good detection of the object, there will also be an explanation of which parameters are adjustable, and which limitations are to be expected.

The report presents how the computer vision has been combined with a manipulator with relationship to the communication, transformations and the user interface between robot, camera and the computer. Equations for converting a position observed in a 2D camera to a world position represented in 3 dimensions are included, and tools to determine a suitable trajectory with respect to kinematic constraints are presented.

The evaluation of robustness and measurement of computation time are presented as plots. Both SIFT and SURF performed well in the tracking of the object in motion and all experiments carried out successfully. Some problems occurred due to communication delay between the computer and the manipulator, and possible solutions are suggested and presented in the report.

Sammendrag

Målsetningen for dette prosjektet var å få en manipulator til å gripe et objekt i bevegelse ved bruk av robot syn. To deteksjonsmetoder ble sammenlignet og implementert i et kamerabasert styringssystem i robotcellen ved Institutt for produksjons- og kvalitetsteknikk.

Begge metodene er basert på de samme grunnfunksjonene, men strukturen og matematikken bak algoritmene er annerledes. Den ene metoden er påstått å være mye raskere enn den andre, og begge metodene skal være upåvirket av skalering- og rotasjonsendinger i et bilde. For å underbygge disse påstandene, er det blitt gjort forskjellige ekskrementer for å måle robusthet og beregningstid. Alle eksperimentene er gjennomført ved bruk av et 2D kamera som observerer et bestemt objekt som beveger seg i en fast rute med en konstant fart. Metodene ble målt i forhold til forskjellige klistrelapper som ble festet på objektet, og metodene ble implementert med forskjellige begrensinger og egenskaper slik at det kunne samles data fra ulike situasjoner.

For å kunne gjennomføre et godt praktisk eksperiment, hvor begge metodene ville kunne kombineres med roboten, var det viktig å måle beregningstiden til metodene, og eventuelt komme med forslag til forbedringer. For å kunne følge objektet og forutse dens neste bevegelse, er det viktig at bildene fra kameraet bli behandlet så nært sann tid som mulig. For å forstå hvilken informasjon man skal se etter i et bilde, er grunnleggende kunnskap om kamera, transformasjoner og bildebehandling presentert i rapporten. For å kunne detektere objektet best mulig, blir det også forklart hvilke justerbare parametere, begrensinger og resultater man kan forvente for begge metodene.

Kommunikasjon, transformasjoner og bruker grensesnitt for kunne navigere en manipulator ved bruk av robot syn vil bli forklart, og formler for å konvertere posisjoner sett med 2D kamera til et 3 dimensjonalt punkt i verden vil bli utledet. Det blir også gitt en introduksjon til verktøy som kan bergene en passende bane for roboten på hensyn av kinematiske restriksjoner.

Alle eksperimentene har vært gjennomførbare, og evalueringen av robusthet og måling av beregningstid blir presentert i plot. Det oppsto noen problemer med kommunikasjonen mellom datamaskinen og roboten. Mulige løsninger er blitt foreslått, og underbygget med tilhørende plott av målinger.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In 2014, the worldwide sales of industrial robots reached a new record with an increase of 27% soled units in one year [8]. This shows that industry is increasingly willing to invest in robotics and automotive solutions to do work traditionally done by humans. This progress is especially large within automation, chemical, rubber, plastic and food industry, and the strongest drivers of the growth were the automotive followed by the electronics industry [8].

The main topic of this report is relates to automation, and is specialized to solutions for subsea technology and mass production. Within subsea technology, robots are used in petroleum production and mining of minerals. Subsea structures are currently deployed on thousands of meters depth, and the industry need customized units to operate in these harsh environments. Remotely operated vehicle (ROV) and autonomous underwater vehicle (AUV) are developed for this purpose, and new discoveries make the conditions increasingly challenging. High pressure, currents, obstacles and sudden temperature changes may cause vibration and unwanted movement on the underwater vehicle. These challenges make operations more time-consuming and solutions to decrease the influence of these challenges are needed. The vibrations may result in issues of brightness, rotation, scale, noise and illumination in the sight of view. Since the navigation system of the vehicle depends on information about the surroundings, computer vision should be able to collect reliable data close to real time. Similar detecting properties is also relevant for mas production, where computer vision should be able to classify specific object according to design, patterns or colors.

To day, some algorithms for detecting objects are developed and results have been compared [10]. This report have some similarities to this previous comparisons, but the current data is based on a live tracking of an object instead of tracking an object in a collection of different images. This evaluation will give an indication according to which a feature detector is useful in situations related to subsea technology and mass production, or not.

Scale Invariant Feature Transform (SIFT) and Speeded Up Robust Feature (SURF) are the two methods tested in this report. Different analytical experiments were performed to gather reliable data for an evaluation, and a practical experiment was performed to prove the utility of a feature detector. The experiments were implemented using C++ with an open source library called OpenCV included. For acquiring the relevant competence for solving the given objectives, a literature study was also a part of the project.

1.2 Objectives

The main objectives of this Master's thesis has been as follows:

- Present the methods SIFT and SURF
- Implement SIFT and SURF for an image stream and track an object in motion. Test the methods, and discuss the results according to computation time and robustness.
- Implement an experiment where an Agilus robot is navigated according to a camera system using SIFT and SURF.
- Try to grasp an object in motion by using computer vision.

1.3 Structure of the Report

The rest of the report is organized as follows:

- Chapter 2 presents fundamental information about a camera, relationship between 2D and 3D, and which information an image contains.
- Chapter 3 describes the mathematics behind SIFT and SURF, together with an introduction to a matching method.
- Chapter 4 explain fundamental knowledge about robotics, communication and user interface.
- Chapter 5 explain the experiments and presents collected data.
- Chapter 6 presents results and improvements.
- Chapter 7 summarize the experiments with a conclusion, discussion and presents recommendations to further work.
- Appendix A includes additional information for Chapter 5.
- Appendix B includes additional information for Chapter 6.
- Appendix C includes C++ source code.
- Appendix D Digital Appendix contains source code and movies from the experiments.

Chapter 2

Image Formation and Image Processing

To track an object in motion, continuous collection of accurate data is important. Data must be easy to analyze and the total operation should be as close to real time as possible. To understand which information is important and how to detect it, this chapter will discuss fundamental knowledge about camera, transformations and image processing.

2.1 Camera

Pinhole is a simple way to present camera basics. It is a closed box with only light coming thought a small hole in center of one of the walls. The hole is facing the object and is made to admit enough light to make an inverted projection of the object on the inside wall ahead of the hole, see figure 2.1. The length between these walls is called focal length, f, and if f is known, the model can be used to find the geometry of a point in the world, expressed in two dimensions.

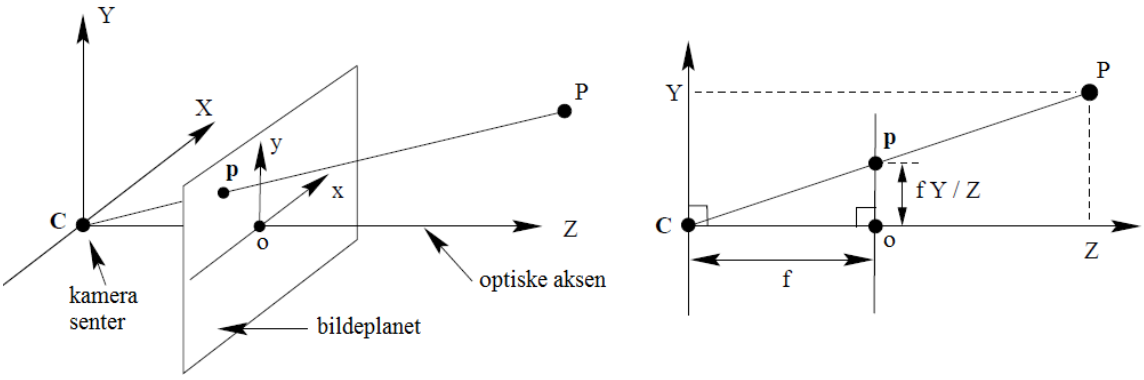


Figure 2.1: Pinhole camera geometry [7].

The figure illustrates the camera center C, where the optical axis intersects the image plane with a focal length $z = f$, seen from camera. This point of intersection is called the *principal point* o and defines origin of the image plane. Using similar triangles, the point $\boldsymbol{P} = (X, Y, Z)$ in the world can be projected in the image plane as $\boldsymbol{p} = (x, y)$. Where

$$x = f\frac{X}{Z}, \;\; y = f\frac{Y}{Z} \tag{2.1}$$

that is a perspective projection from world to the image plan, and represent the point transformed from $\mathbb{R}^3$ to $\mathbb{R}^2$.

2.2 Digital Image

A digital image plane is represented as a $W \times H$ grid of equally spaced points called pixels, where W is width and H is height. The grid is organized in pixel coordinates as two vectors, u and v. The vectors are both positive integers and have origin $= (0, 0)$ defined in the upper left corner in the image plane. The position from camera to world is represented in Cartesian coordinates (x, y) with origin in center of the image plane. To create a correspondence between the pixel coordinates and the Cartesian coordinates, a relationship may be expressed with formula

$$u = \frac{x}{\rho_w} + u_0, \quad v = \frac{y}{\rho_h} + v_0, \tag{2.2}$$

where ρ_w is the width and ρ_h is the height of a single pixel, and $o = (u_0, v_0)$ is the coordinate of the principal point, see Figure 2.2.

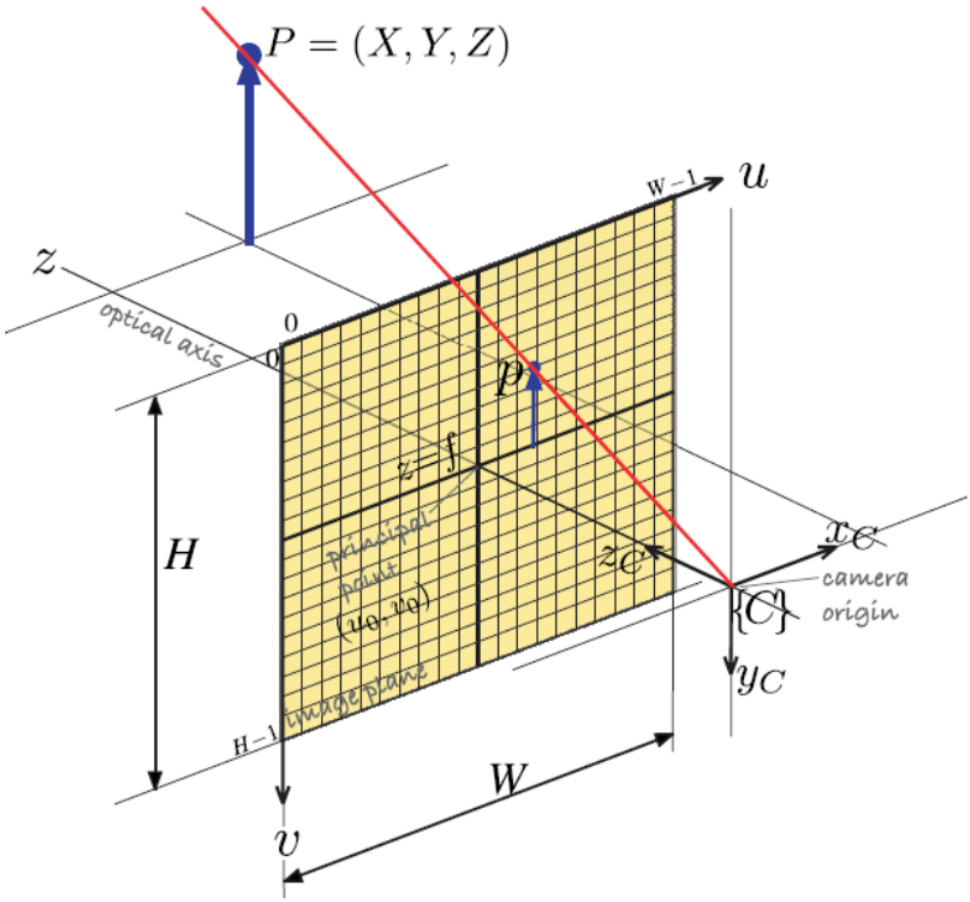


Figure 2.2: *Central projection model* representing image plane and discrete pixels [4].

2.3 Homogeneous Transformation

When the relationship between Cartesian coordinates and pixel coordinates are known, homogeneous transformation may be used to make a projection of world point $\boldsymbol{P} = (X, Y, Z)^{\mathrm{T}}$ in the image plane. By scaling the Cartesian X and Y coordinates with the Cartesian Z coordinate, the transformation between the Cartesian and pixel coordinates becomes linear, and the world point $\boldsymbol{P}$ can be represented in homogeneous form $\widetilde{\boldsymbol{p}} = Z(x, y, 1)^{\mathrm{T}}$ (tilde is used to express that the coordinates are homogeneous) in the image plane. The world point coordinates are converted pixel coordinates, with

$$u = \frac{x}{\rho_w} + u_0, \quad v' = \frac{y}{\rho_h} + v_0. \tag{2.3}$$

Then the homogeneous pixel coordinates of point $\widetilde{\boldsymbol{P}}$ is represented as

$$\tilde{\boldsymbol{p}} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2.4}$$

in the image plane. When the world point is represented as homogeneous pixel coordinates in the image plane, the transformation between the point and camera have to be defined. Next

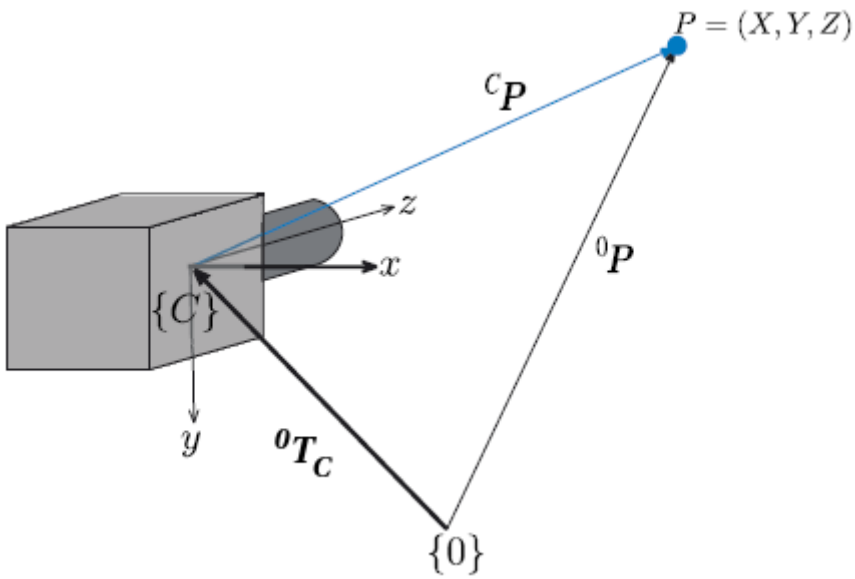


Figure 2.3: Transformations between camera and a world point [4].

step is to define the transformation from world point $\widetilde{\boldsymbol{P}}$ to the camera with

$${}^{C}\widetilde{\boldsymbol{P}} = ({}^{0}\boldsymbol{T}_{C})^{-1}\, {}^{0}\widetilde{\boldsymbol{P}}, \tag{2.5}$$

where ${}^{0}\boldsymbol{T}_{C}$ is the transformation from world $\{0\}$ to camera $\{C\}$ and ${}^{0}\widetilde{\boldsymbol{P}}$ is transformation from world to $\widetilde{\boldsymbol{P}}$, see Figure 2.3. When relationship between pixel coordinates and Cartesian coordinates, the transformations and the internal camera parameters are known, the projection

of a world point in the image plane may be express with respect to the camera. The general form of the expression is

$$\tilde{\boldsymbol{p}} = \underbrace{\begin{bmatrix} f/\rho_w & 0 & u_0 \\ 0 & f/\rho_h & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{K}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\boldsymbol{P}_0} ({}^0\boldsymbol{T}_C)^{-1} \tilde{\boldsymbol{P}} \tag{2.6}$$

$$= \boldsymbol{K}\boldsymbol{P}_0({}^0\boldsymbol{T}_C)^{-1}\tilde{\boldsymbol{P}} \tag{2.7}$$

$$= \boldsymbol{C}\tilde{\boldsymbol{P}}. \tag{2.8}$$

where $\boldsymbol{C}$ is a 3×4 matrix that express transformation due to scale, translations and perspective projection [4]. This matrix is often referred to as the *camera calibration matrix*. By repeating this operation on each point in an image, a complete object will be expressed in the image plane, and SIFT or SURF may be used to detect robust an invariant regions.

2.4 Grey scale

To detect regions in an image that are invariant to variations in transformation, scale, noise and rotation, it is necessary to emphasize and compare the pixels. One method is to convert the image to *grey scale*, which means that the pixels are represented according to intensity. The intensity is described by a value between 0 and 1, where black equals 0 and white equals 1, see Figure 2.4

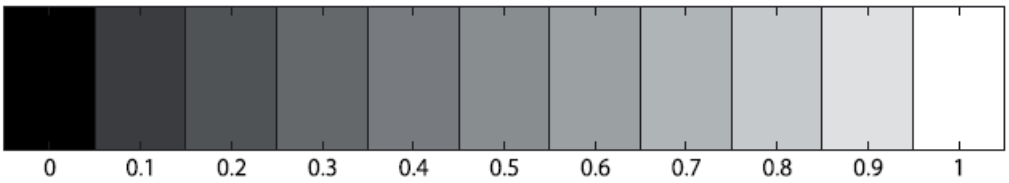


Figure 2.4: Example of emphasizing intensity in gray scale [4].

This method makes it possible to detect and evaluate extrema values in the image. A tool that is important for both SIFT and SURF is the estimation of robustness and orientation of the selected regions. An example of an image that is converted from color to grey scale is illustrated in Figure 2.5.

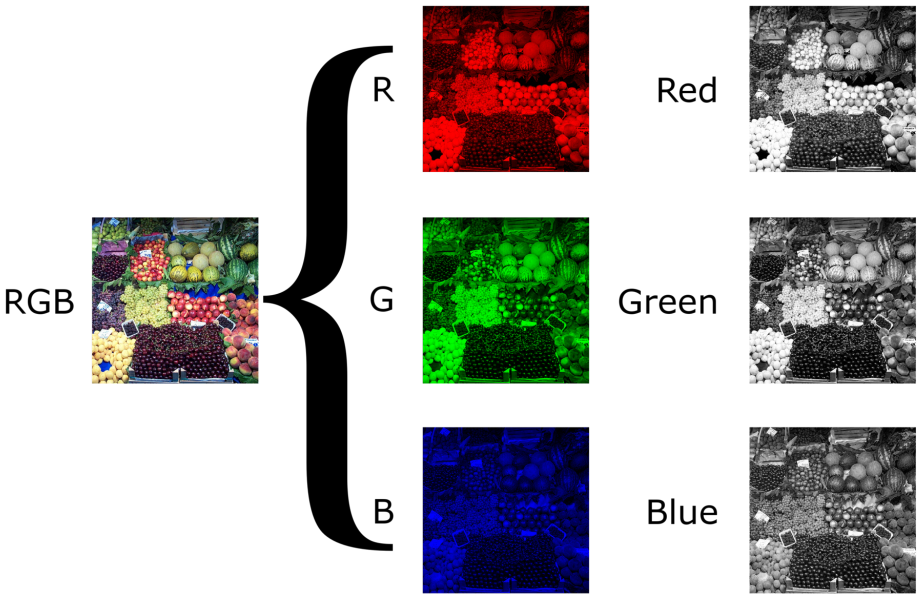


Figure 2.5: An example of a RGB image converted to grey scale [20].

2.5 Features

To track a moving object, a sequence of images of object motion must be processed and compared. For easy detection, the selection of an object or tag with distinct points is important. An object presented in images may be influenced by scaling, affine invariance, rotation and transformation. Detection of unique regions that are highly reliable is therefore necessary to unambiguously identify the object in other images. This type of homogeneous regions (*blobs*), lines and distinct points in a scene are called *features* [4]. A feature can be represented by the pixel coordinates of an extrema point in center of a unique region, the orientation of the image gradient and a scale value representing the scale where the extrema point was first detected. A feature is usually represented as a clock with a pointer, see Figure 2.6. The radius of the clock indicates the uniqueness of the feature and the pointer indicates the direction of the greatest image gradient. The radius and orientation are determined depending on the method. More details are described in Chapter 3.1 about SIFT and Chapter 3.2 about SURF.

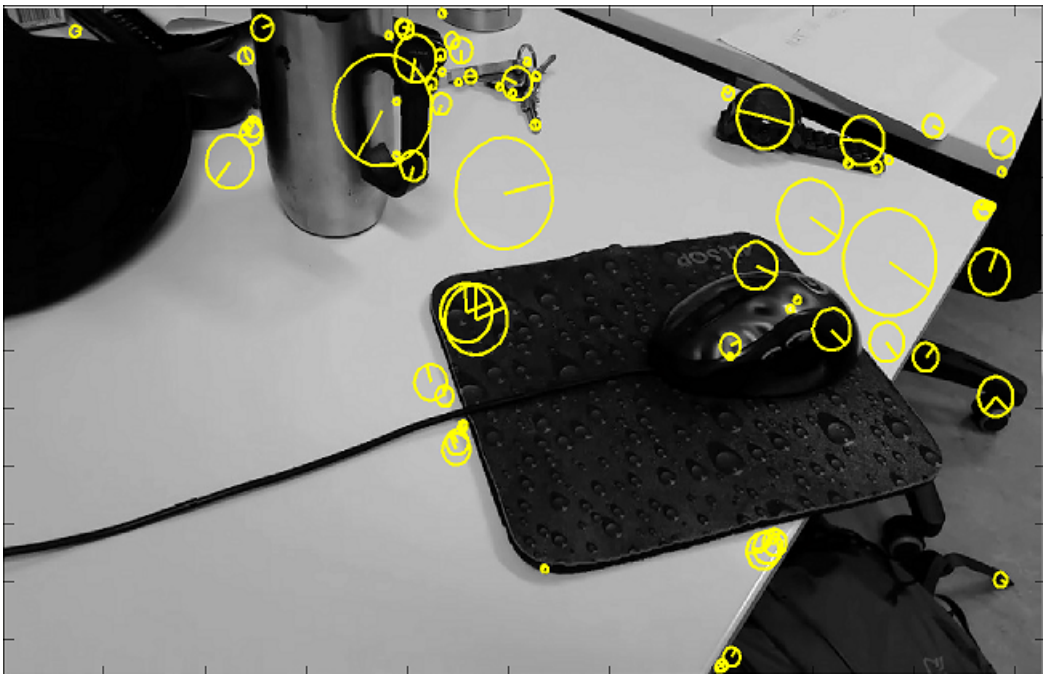


Figure 2.6: Illustration of features.

2.6 Descriptors

The main property of a feature is the ability to be repeatedly identified in other images. The repeatability expresses reliability in terms of how each feature may be detected in different images. To make a feature detectable in a new image among thousands of other points, each single feature is awarded a distinctive and unique description of their closest pixels. This description contains vectors that represents the orientation and the image gradient in the specific area. The description is called a *descriptor*, and is structured and dimensioned depending on method. The descriptor can basically be illustrated in a grid of vectors, see Figure 2.7. The SIFT features in Figure 2.6 are described with descriptors in Figure 2.8. Details about how descriptors are constructed according to method are discussed in chapter about SIFT 3.1 and SURF 3.2.

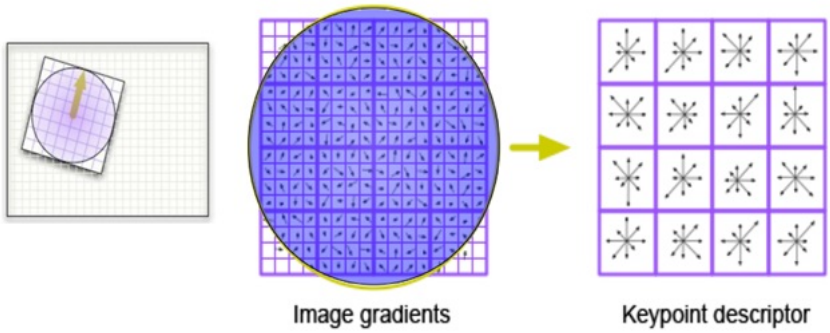


Figure 2.7: Illustration of a SIFT-descriptor [3].

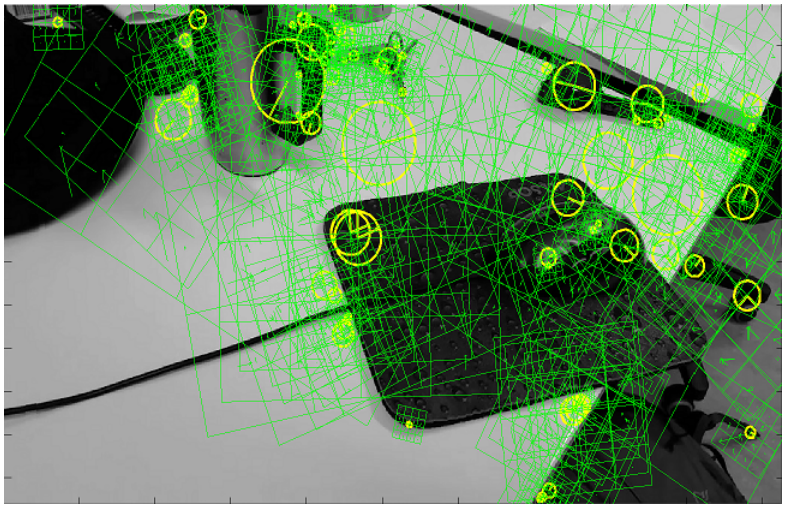


Figure 2.8: Illustration of features with descriptors. The descriptors are illustrated as green grids.

Chapter 3

Computer Vision

When using SIFT and SURF, it is useful to be familiar with the structure of the algorithms, the limitations, and expected results. It is also necessary to know an accurate and fast matching method to compare the results afterwards. This chapter gives a description of this and illustrates the SIFT and SURF methods and a matching method called FLANN.

3.1 SIFT

SIFT is an acronym for scale-invariant feature transform and is an widely used algorithm within *computer vision* [13]. SIFT was published by David Lowe (1999) and patented by the University of British Columbia [14]. The purpose of SIFT is to extract scale and rotation invariant features form an image, that are partially invariant to variation in illumination and 3-dimensional camera viewpoint. The goal is to make each feature so distinct that it cannot be confused with other features in a database holding thousands of features from different images [13]. To detect these unique features, SIFT is built up in four stages where the regions of interest are identified.

3.1.1 Detection of scale space extrema

The first stage in SIFT is scale invariant feature detecting. This is done by converting the image to grey-scale and then search for features that are stable across all possible scales. The method is based on a function with continuous scale change called *scale space* [21]. The purpose of scale space is to detect pixels that express either a minima or maxima in a specific scale. By increasingly scale of the image, regions becomes increasingly blurred and smaller regions progressively disappears from view [4]. The scale space of an image is defined as a function $L(u, v, \sigma)$ where u, v are pixel coordinates and σ is scale. This function is produced of a variable-scale Gaussian $G(u, v, \sigma)$ in convolution with an input image $I(u, v)$. This gives formula

$$L(u, v, \sigma) = G(u, v, \sigma) * I(u, v), \tag{3.1}$$

where $*$ is a convolution operator and

$$G(u, v, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(u^2+v^2)/2\sigma^2}. \tag{3.2}$$

To effectively detect maxima and minima in the scale space, Lowe have proposed using *Difference of Gaussian* (DiffG) in convolution with the image, $D(u, v, \sigma)$. A DiffG-filter is illustrated

in Figure 3.1. The convolution with DiffG and the image may be expressed with formula

$$D(u, v, \sigma) = (G(u, v, k\sigma) - G(u, v, \sigma)) * I(u, v) = L(u, v, k\sigma) - L(u, v, \sigma). \tag{3.3}$$

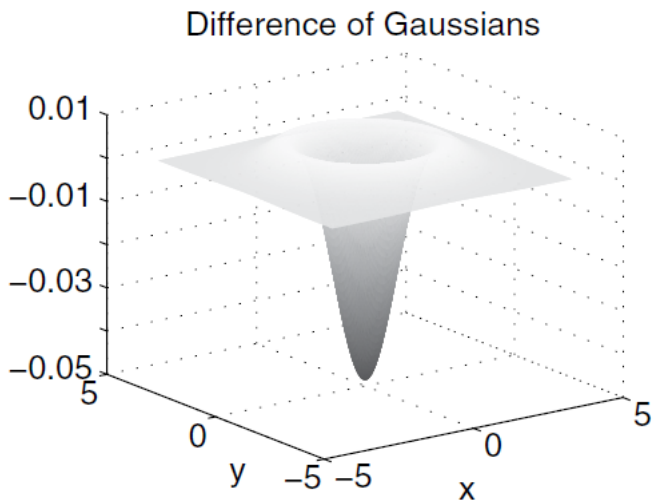


Figure 3.1: Illustration of a DiffG-filter used in SIFT [17].

The scaling process is built up in octaves where scale is increasing within each octave. The scaling is increased with a multiplication factor k for each step. This means that the first filter may have $\sigma_0 = \sigma$, see for instance left filter in Figure 3.2, the second filter have $\sigma_1 = k\sigma$, see right filter in Figure 3.2 and the third filter have $\sigma_2 = k^2\sigma$. A typically Gaussian filter is illustrated in Figure 3.3, where the filter is highly weighted in the center.

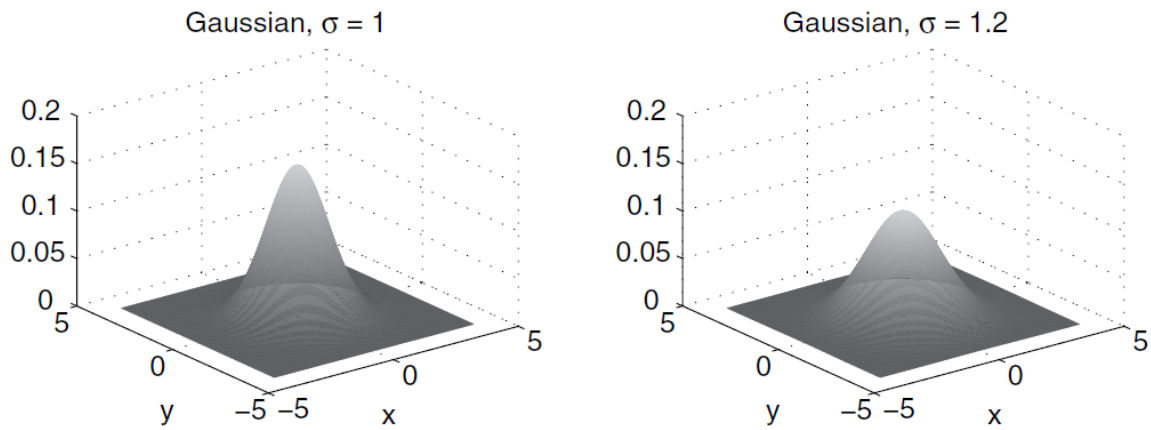


Figure 3.2: Illustration of two Gaussian-filters with different scale used in SIFT [17].

To scale the image across all possible scales Lowe proposed values based on experience from experimental data. The scales are chosen according to the highest percentage repeatability in respect of efficiency [13]. This leads to the multiplicative factor $k = 2^{\frac{1}{s}}$, where s is an integer. It is preferred to have $s = 3$ and $s + 1$ images in each octave, with a base scale of $\sigma = 1.6$ applied to the first image in each octave [17]. It is also recommended to reduce the image

resolution with a factor of 2 for each octave, while the size of the sequence of Gaussian filters applied at each octave remains the same. This means that an image with resolution of 512×512 in convolution with a DiffG-filter in the first octave, will have resolution 256×256 in second octave, etc. This is illustrated in Figure 3.4.

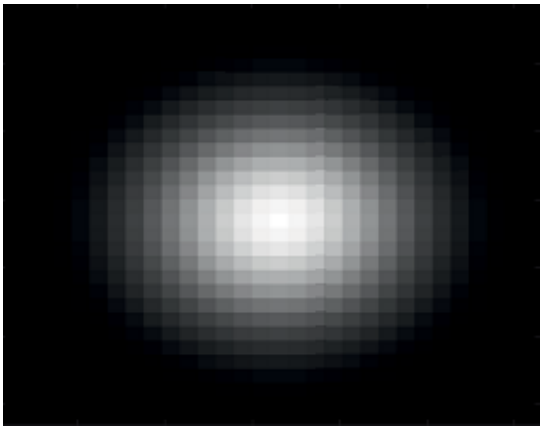


Figure 3.3: Gaussian-filter in SIFT with $\sigma = 5$ implemented in MATLAB.

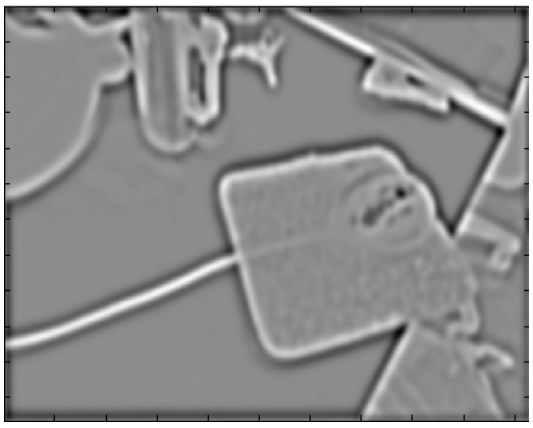
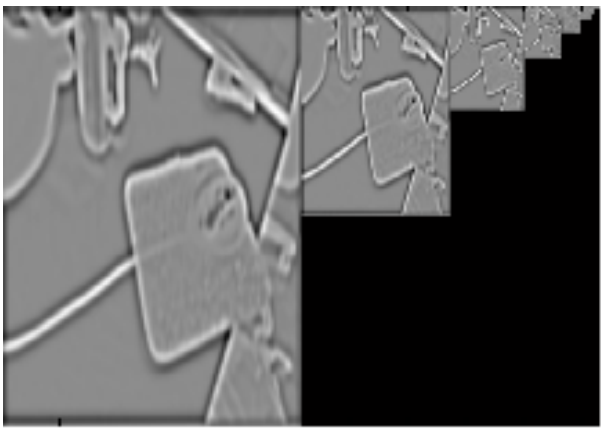


Figure 3.4: DiffG made of $G(u, v, k^5\sigma)$ and $G(u, v, k^4\sigma)$ in SIFT.

This simplifies the calculation so that smaller images are applied to a one-sized filter, instead of applying very large filters to original-sized images [17]. When the image have been scaled across all possible scales, all the resulting $D(u, v, \sigma)$ images are stacked in a pyramid shaped scale space, see Figure 3.5. Lowe's octave structure for computing the scale space DiffG representation of an image is illustrated in Figure 3.6.



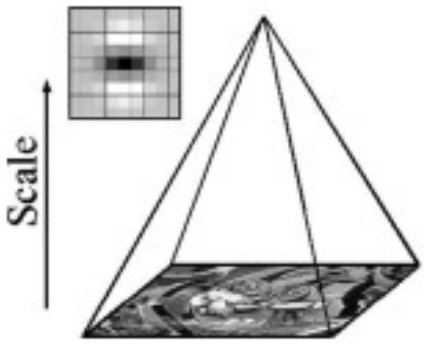


Figure 3.5: Pyramid of different DiffG-filters in convolution with an images in different sizes. Left: Illustration implemented in MATLAB. Right: Image from [1].

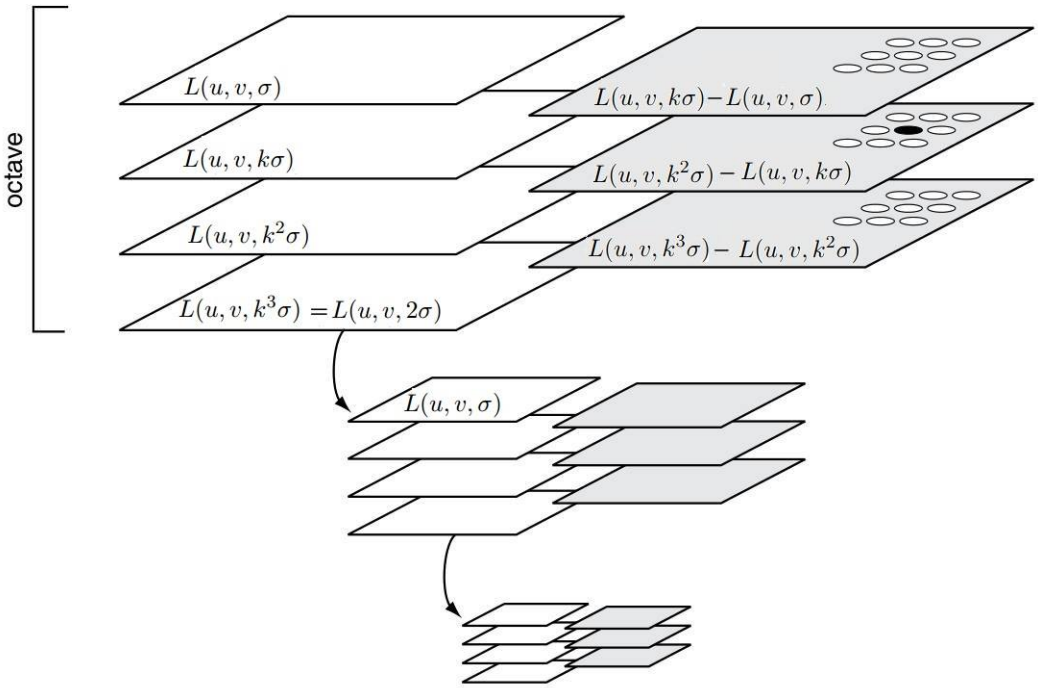


Figure 3.6: Lowe's octave structure for computing the scale space DoG representation of an image [17].

3.1.2 Local extrema detection

When an image is scaled across all possible scales, the maxima and the minima must be located. This is done by comparing each single pixel with its 28 neighbors. The pixels are picked from images $D(u, v, \sigma)$ stacked in the pyramid and compared with the eight closest pixels in the current scale, the nine closest pixels in the scale above and the nine closest pixels in the scale below, see Figure 3.7. If a pixel appears to express either maxima or minima among the neighbors, this pixel is the important one.

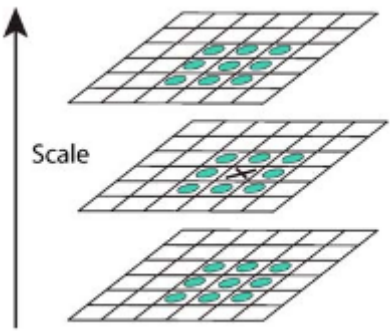


Figure 3.7: Detection of local maxima and minima in SIFT. Pixel X will be compared with its 28 neighbors [13].

3.1.3 Elimination of unstable extrema values

The comparison of neighbors may accept many local extrema and not all of these will be stable and required forward in the process. To define the required pixels, accurate mapping eliminates pixels related to low contrasts and pixels along edges. The first step is to eliminate pixels with low contrast, for instance pixels in black dots on a white paper. These regions of pixels will appear to be unique compared to the white paper, but uniform compared to other black dots on the same paper. These pixels are not required further in the evaluation. To determine the exact location of stable extrema values, interpolation of collected data describing the neighbor pixels is used. The interpolation is done with *Taylor expansion*, where each sample point from $D(u, v, \sigma)$ is defined as the origin. An expression of the interpolation may be

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x^T}\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}, \tag{3.4}$$

where $\boldsymbol{x} = (u, v, \sigma)$ is the offset from the pixel that is evaluated (origin). By deriving the Taylor expansion with respect to $\boldsymbol{x}$ and setting the expression to zero, the location of the extrema value $\hat{\boldsymbol{x}}$ can be determined with formula

$$\hat{\mathbf{x}} = -\frac{\partial^2 \mathrm{D}^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}}. \tag{3.5}$$

If $\hat{\mathbf{x}}$ is greater than 0.5 in any of the directions u, v or σ, the extrema value will be closer to another pixel $\mathbf{x}$. In that case, $\mathbf{x} = (u, v, \sigma)$ must be changed and a new interpolation is performed with respect a new point. This operation is repeated until $\hat{\mathbf{x}}$ is less than 0.5 in all u, v and σ, or until interpolations are carried out a predetermined number of times. The resulting $\hat{\boldsymbol{x}} = (u, v, \sigma)$ will then be summarized with the location of the evaluated pixel location in $D(u, v, \sigma)$ to determine the interpolated estimate of the location of the extrema value. The next step is then to remove extrema values of small contrasts using $D(\hat{\boldsymbol{x}})$. By combining the formula (3.4) and (3.5) to

$$D(\hat{\boldsymbol{x}}) = D + \frac{1}{2}\frac{\partial D^T}{\partial \boldsymbol{x}}\hat{\boldsymbol{x}}, \tag{3.6}$$

a minimum threshold of $D(\hat{\boldsymbol{x}})$ may be determined. This eliminates the smaller $D(\hat{\boldsymbol{x}})$ values. Lowe suggests a threshold value of 0.03 as default in his paper [13]. Figure 3.8 illustrates a result of increasing this threshold.

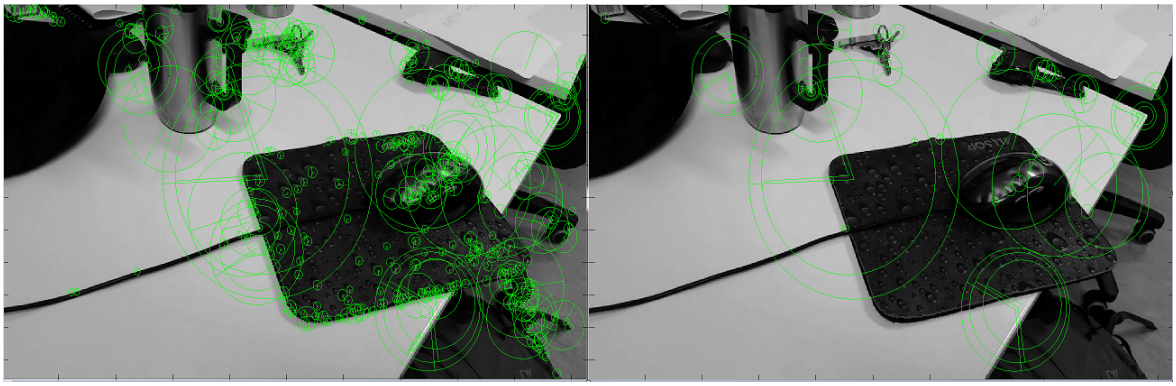


Figure 3.8: Increased threshold to eliminate low contrasts in SIFT. Example is implemented in MATLAB.. Left: Low contrasts threshold = 0.02. Right: Low contrasts threshold = 0.05.

After elimination of extrema values related to small contrasts, the third stage is to eliminate pixels along edges. Pixels along edges will appear as high contrast across the edge, but low contrast along the edge. These regions make unstable features, and are not required further in the evaluation. The method used to eliminate these regions is based on the *principal curvature* in images picked from the scale space pyramid. The principal curvature depends on the image contrast. A large curvature will represent a high contrast and a small curvature will represent a low contrast. The principal curvature of the image D will be proportional with the *eigenvalues* of the *Hessian matrix*. This means that limitations between maximum and minimum of the principal curvature may be defined by estimating the relationship between the eigenvalues in a Hessian matrix. Hessian matrix is given by

$$\boldsymbol{H} = \begin{bmatrix} D_{uu} & D_{uv} \\ D_{vu} & D_{vv} \end{bmatrix} \tag{3.7}$$

and the relationship between the eigenvalues are estimated by computing the *determinant* and the *trace* of the Hessian matrix. The trace and determinant are expressed as

$$\mathrm{Tr}(\boldsymbol{H}) = D_{uu} + D_{vv} = \alpha + \beta, \tag{3.8}$$

$$\mathrm{Det}(\boldsymbol{H}) = D_{uu}D_{vv} - (D_{uv})^2 = \alpha\beta, \tag{3.9}$$

where α represents the largest eigenvalue and β represents the smallest. By estimating the ratio r between these two eigenvalues, the relationship may be expressed as $\alpha = r\beta$. This gives the formula

$$\frac{\mathrm{Tr}(\boldsymbol{H})^2}{\mathrm{Det}(\boldsymbol{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r+1)^2}{r}. \tag{3.10}$$

By checking if the principal curvature have a ratio less than r, pixels along edges will be eliminated. This may be determined with formula [12]

$$\frac{\mathrm{Det}(\boldsymbol{H})}{\mathrm{Tr}(\boldsymbol{H})^2} < \frac{r}{(r+1)^2}. \tag{3.11}$$

This maximum limit is recommended to be $r = 10$ by Lowe [13], and Figure 3.9 illustrates a increasing of the threshold for edge elimination.

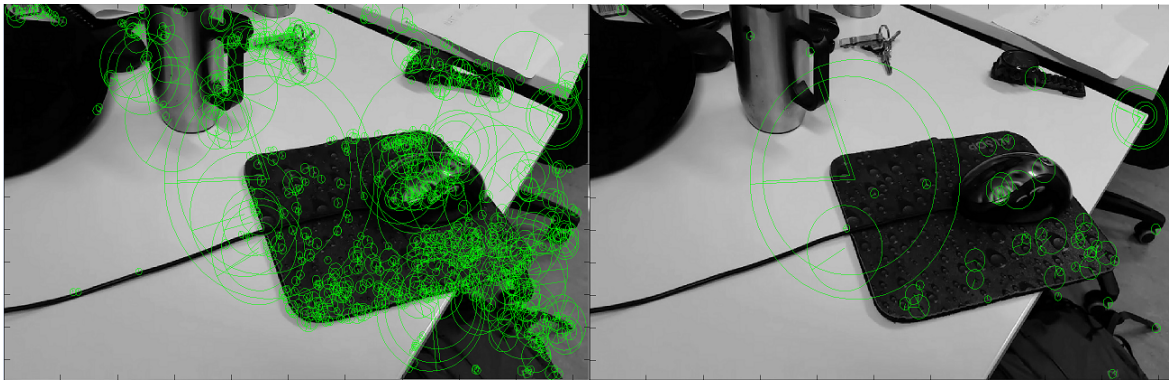


Figure 3.9: Increasing of threshold for edge elimination in SIFT (high contrasts). Example is implemented in MATLAB. Left: Edge threshold = 0.1. Right: Edge threshold = 0.7.

3.1.4 Feature orientation

At this point, each distinctive point (key point) has an estimated image coordinate and a scale. The remaining part is to assign a consistent orientation to each key point to ensure rotation invariance. The orientation is estimated according to the gradient magnitude $m(u, v)$ and orientation $\theta(u, v)$ for a Gaussian smoothed images, L, with the closest scale with respect to each specific key point. For instance, if an extrema value was detected in $D(u, v, k^5\sigma)$, then $m(u, v)$ and $\theta(u, v)$ are determined with respect of image plane $L(u, v)$ with $k^5\sigma$ in the scale axis, see Figure 3.5. The purpose of estimating a consistent orientation based on local image properties is to represent the descriptor with respect to this orientation. This makes features recognizable independent of image rotation. Since the extrema values are detected at different scales, some of the features will be more robust than others. To determine the gradient magnitude and the orientation of one specific Gaussian smoothed images, Lowe has derived two equations, (3.12) and (3.13) [13].

$$m(u, v) = \sqrt{(L(u+1, v) - L(u-1, v))^2 + (L(u, v+1) - L(u, v-1))^2}, \tag{3.12}$$

$$\theta(u, v) = \tan^{-1}\frac{(L(u, v+1) - L(u, v-1)}{(L(u+1, v) - L(u-1, v)}. \tag{3.13}$$

These orientations are then plotted in a histogram divided in 36 columns. Each column represent 10° of a total 360° of possible orientation. Each orientation is weighted with respect to the gradient magnitude and the scale σ representing the Gaussian smoothed image. When all 360° are plotted, a vector expressing the gradient magnitude and the orientation can be defined. This vector is a result of column values that are located within 80% of the highest column in the histogram. The final orientation vector is added to the feature, and the completed feature can be represented with coordinates, scale and orientation. The feature is usually visualized as a clock with a pointer. The gradient magnitude is represented as the clock radius and the orientation is represented as the pointer position. This is illustrated in Figure 3.10.

3.1.5 SIFT-descriptors

Each feature is assigned a descriptor based on the characteristic coordinate u, v, orientation θ and the scale σ of the feature. First, a oriented square centered in u, v is created in the original image. The orientation is defined by θ, and each side of the square is a multiple of the scale σ. The image is blurred with a Gaussian of the similar σ, and the square is divided in a grid of 16×16. Precomputed gradients, see Chapter 3.1.4 are defined in each grid, and weighted with a Gaussian filter centered in the middle of the square. This means that gradients close to u, v will be highly weighted. The next step is dividing the square into a 4×4 grid, where a histogram of eight orientation bins are defined in each grid. A collection of all these histograms results in a 128-dimensional vector [17]. To reduce the effect of illumination changes, is the vector normalized into unit length. This cancels extremely large values, and will reduce affects because of contrast changes. The final vector will then be the specific descriptor of the specific feature. The process is illustrated in Figure 3.10.

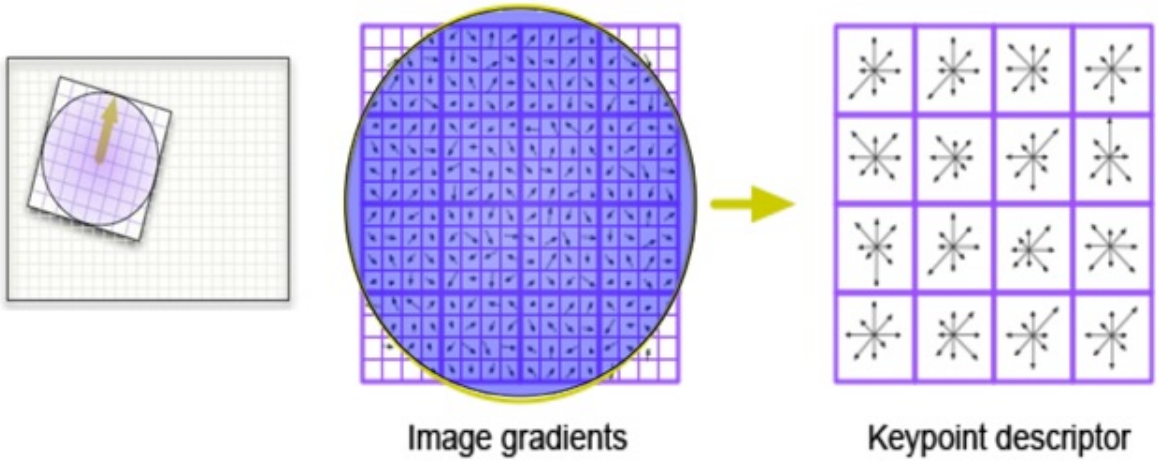


Figure 3.10: Illustration of a SIFT-descriptor [3].

3.2 SURF

SURF is an acronym for Speeded Up Robust Features [1] and was first presented by Herbert Bay (2006). SURF is a local feature detector partly inspired by SIFT, and includes many similar functions. The main difference relates to the mathematics that underlies the algorithms, and according to [1] SURF is less complicated and organized to produce fast computation compared to SIFT. To become more familiar with the algorithm, this chapter presents the structure and the procedures for local features detection according to the SURF method.

3.2.1 Interest point detection

First step in SURF is to detect pixels of interest. Similar to SIFT, all images are initially smoothed in SURF. The difference is that SURF uses a more efficient method called a *box filter*, where the filter size is easily estimated using a method called *Integral Image* [19]. This is an iterative calculation of the sum of pixel values within a rectangle of an image. In image $I(u, v)$, the integral image I_Σ will represent the sum of pixel values between a point $(u, v)^{\mathrm{T}}$ and the origin. Formally expressed with formula

$$I_\Sigma(u, v) = \sum_{i=0}^{i \leq u} \sum_{j=0}^{j \leq v} I(i, j). \tag{3.14}$$

When the integral image is defined, the sum of pixel values within the rectangle may easily be calculated, see example in Figure 3.11.

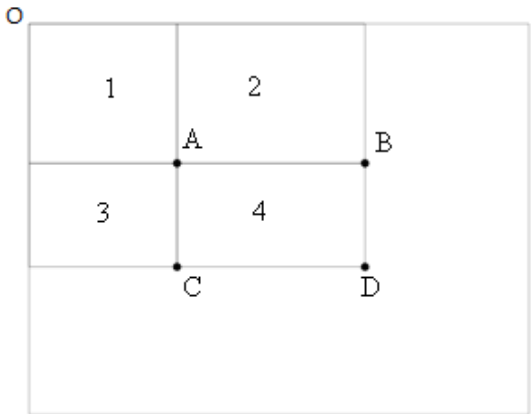


Figure 3.11: Illustration of pixel values estimated with an integral image [19].

This means that the sum of pixel values within the rectangle 4 may be computed according to four reference points. The sum of pixel values at point A is equal the sum of pixel values in rectangle 1. The sum of pixel values at point B is $1+2$, the sum of pixel values at C is $1+3$ and the sum of pixel values at D are equal $1+2+3+4$. The sum of pixel values in rectangular 4 may finally be computed with formula

$$\Sigma = A + D - (C + B). \tag{3.15}$$

Since the computation is independent of filter size, this method is very efficient to filter large areas. SURF save a lot of computing power and can execute fast convolutions on an approximately constant time. To locate the robust extrema values in the image, the detector is based on the maximal determinant of a double derivative Gaussian with respect to the pixel (u, v). This results in a Hessian matrix $\boldsymbol{H}(u, v, \sigma)$ in $I(u, v)$ with a scaling σ. The Hessian matrix expresses with formula

$$\boldsymbol{H} = \begin{bmatrix} L_{uu} & L_{uv} \\ L_{vu} & L_{vv} \end{bmatrix}, \tag{3.16}$$

where L_{uu} is the double derivative Gaussian $\frac{\partial^2}{\partial^2(u,v)^2} g(\sigma)$ in a convolution with image $I(u, v)$, similarly for L_{uv} and L_{vv}. This method requires much computing power and was replaced with box filter due to efficiency, like LoG was replaced with DiffG in SIFT [2]. Box filter is illustrated in Figure 3.12. In this manner, SURF constructs a scale space of box filters in various sizes.

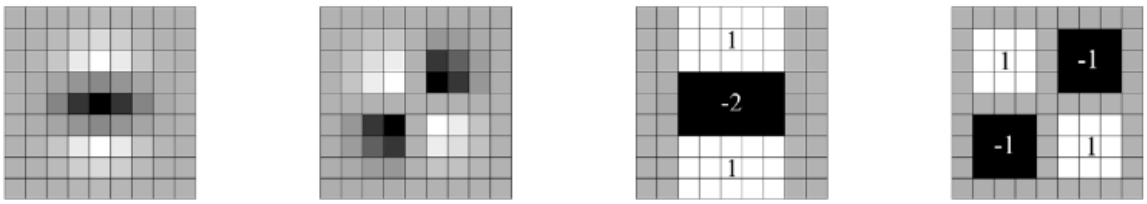


Figure 3.12: Left: Double derivative Gaussian filter in v direction (L_{vv}) and in uv direction (L_{uv}). Right: Approximation of double derivative Gaussian with box filter in same directions (D_{uv}, D_{vv}). Grey area is weighted 0 [1]

The advantage of this approach is that the box filters are defined in terms of integral images, which keeps computation time approximately constant regardless to filter size. The box filter is denoted as D_{uu}, D_{uv} and D_{vv}, and Figure 3.12 illustrates a filter with dimension 9×9 and approximates a Gaussian with $\sigma = 1.2$ [1]. This scale is the lowest scale in a scale space of SURF. The approximation between box filters and the determinant of a Hessian matrix of a double derivative Gaussian may be expressed with formula

$$\det(\boldsymbol{H}_{\text{approx}}) = D_{uu}D_{vv} - (wD_{uv})^2, \tag{3.17}$$

where $w = 0.9$ is a proportion required to make the approximation correct [1]. As mentioned in Chapter 3.1.1 the scale space pyramid in SIFT is constructed of Gaussian smoothed images. Size decreases and scale increases according to the pyramid height. In SURF, it is the other way around. The first scale space layer has a box filter of dimension 9×9, and then increases the dimension according to the pyramid height, see Figure 3.13. Simultaneously as the box filter increases, the associated scaling σ also increase. This scale value is an approximation

σ_{approx}, and is constantly increased according to the actual box filter. This may be determined from [2]

$$\sigma_{approx} = \text{ActualFilterDimension} \cdot \frac{\text{BaseFilterscaling}}{\text{BaseFiltersize}} \tag{3.18}$$

$$\sigma_{approx} = \text{ActualFilterDimension} \cdot \frac{1.2}{9}. \tag{3.19}$$

When the scale is determined for each filter, the scale space pyramid may be constructed. The scale space contains only box filters, and the filters do not have to be in a convolution with the image at all the time. This make a convolution between the original image and any box filter possible, even several box filters in parallel. This reduces the computing power substantially.

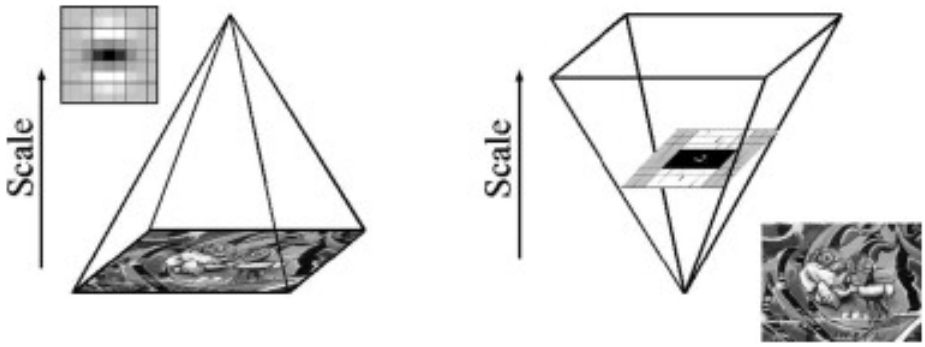


Figure 3.13: Left: Illustrates the scale space pyramid used in SIFT. Right: Illustration the scale space pyramid used in SURF [1]

Similarly with SIFT, the scale space of SURF is divided into octaves. The scaling changes according to a certain value and the pyramid scales the image across all possible scales. To scale the image correctly, the filter center is ensured by increasing the dimension of the filter with 6 pixels for each scale. This means that the first octave contains filters from 9×9 to 15×15, 21×21 and finally 27×27 [1]. Figure 3.14 illustrates how the filter dimensions increase. To scale the image across all possible scales, the scale have to overlap the next octaves. The dimensions of the two first filters in each octave will then be determined according to the first and the last filter in previous octave. All filters after these two filters increase with 6 pixels that duplicates in next octave. This mean that the filter dimensions increases with 6 pixels in first octave, increases with 12 pixels in second and 24 in third. Then first octave consists of filter 9×9, 15×15, 21×21, 27×27 and second octave consists of 15×15, 27×27, 39×39, 51×51. Figure 3.15 illustrates how the scale is distributed in each octave.

3.2.2 Interest point localisation

When the scale space is constructed, extremum values may be located. The approach is similar to the approach in SIFT. The first step is to compare each pixel with its 28 neighbors in scale space, see Chapter 3.1.1. The extrema is then expressed in a Hessian matrix $\boldsymbol{H}(u, v, \sigma)$ and interpolated in a Taylor expansion, similar to $D(u, v, \sigma)$ in SIFT. The 0.5 threshold is also the same as in SIFT, see Chapter 3.1.3. All accepted extrema points are then used to create the features and the descriptors.

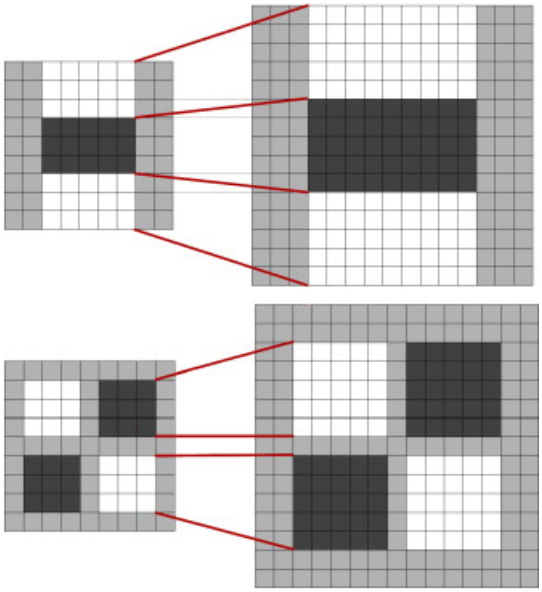


Figure 3.14: Upper filter: D_{vv}. Bottom: D_{uv}. The filter dimension increases from 9×9 to 15×15 [1].

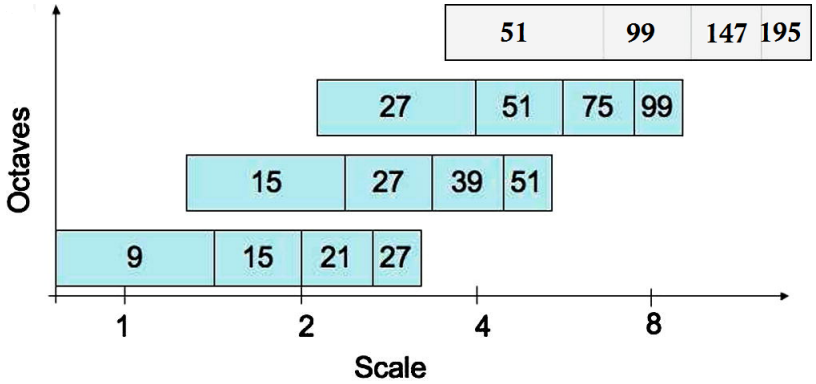


Figure 3.15: Illustration of scale distribution per octave [1].

3.2.3 Orientation assignment

As in SIFT the assignment of a consistent orientation based on the local image properties is important. The determination of the orientation is done using the integral image in combination with a *Haar wavelet* filter to evaluate all points of interest and their neighborhood [1]. Haar wavelet filters are simple filters used to find the gradient in both u and v direction (Figure 3.16). To estimate the orientation, the Haar wavelet filter is defined with a side length of 4σ, searching in a radius of 6σ around each point of interest. The σ value is equal to the representative scale where the extrema was first detected. The pixels are then divided into six windows, where the Haar wavelet responses in the u direction are represented as points along the abscissa, and responses in v direction as points along the ordinate [6]. The dominant orientation in each window is then estimated by calculating the sum of all horizontal and vertical responses within $60°$, and is finally represented as a vector. Each vector is then compared, and the most

dominant of the total six becomes the final feature orientation [1]. See example in Figure 3.17. The feature consists of the same value as in SIFT, a u, v coordinate representing the point of interest, scale σ and orientation, and are also visualized as a clock with a pointer.

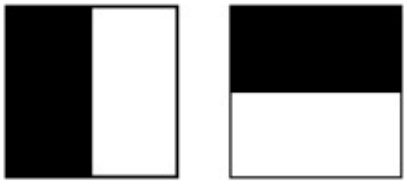


Figure 3.16: Illustration of Haar wavelet filter. Left: Filter for gradients in u direction. Right: Filter for gradients in v direction. The black region is weighted 1 and the white is weighted -1 [1].

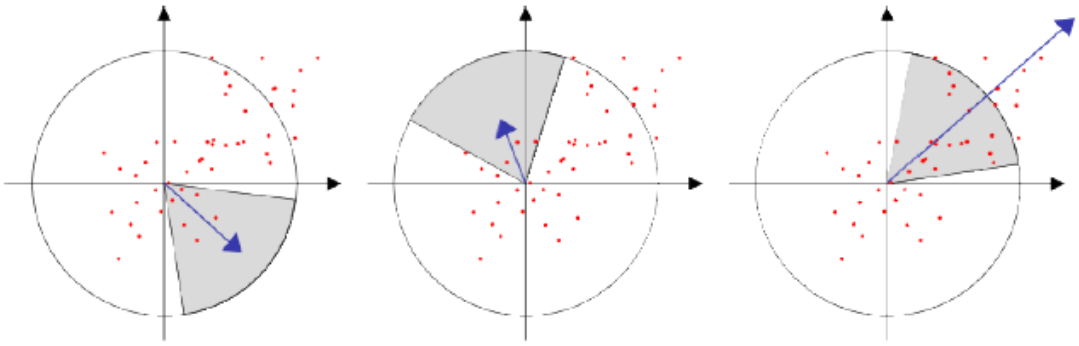


Figure 3.17: Result of Haar wavelet filtering. The red dots represents positive responses, and the blue arrow represents the sum of responses as a vector [1].

3.2.4 SURF-descriptors

The last step is to create a descriptor for each single feature. This is done by constructing a 4×4 grid which is centered by the point of interest, and which is oriented along the orientation of the feature. The grid dimension is set to 20σ, where σ is the representative scale for the specific feature. The next step is to filter the image with a 5×5 Haar wavelet filter to detect the gradients in the region covered by the grid (3.18). When the Haar wavelet filter has filtered the horizontal and vertical direction of each 5×5 region, the responses are summed up in each square of the 4×4 grid (the notation for direction is $x = u$ and $y = v$). In order to capture information about the polarity of intensity changes, the sum of absolute values are also extracted, $|dx|$ and $|dy|$. This results in a $4D$ vector $\boldsymbol{v}$ expressed with formula

$$\vec{\boldsymbol{v}} = \left(\sum dx, \sum dy, \sum |dx|, \sum |dy| \right). \tag{3.20}$$

Concatenation of all values of the 4×4 grid results in a descriptor vector of length 64. This is half the size of a SIFT descriptor, and may result in a faster matching process whit less computation. Figure 3.19 illustrates how Haar wavelet filter responds to three different gradient cases.

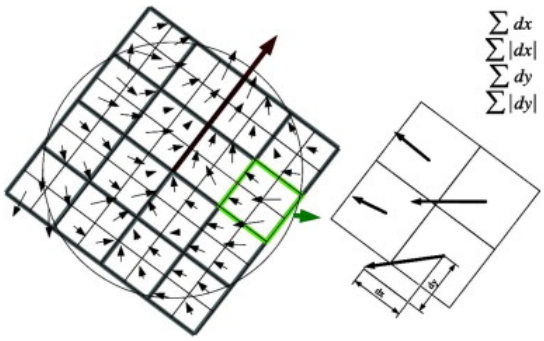


Figure 3.18: Left: Oriented 4×4 grid covering a feature ready to be filtered. Each square of the grid is filtered by a Haar wavelet-filter of 5×5 (illustrated as a 2×2 matrix in right figure) for detection of gradients relative to the orientation of the grid [1].

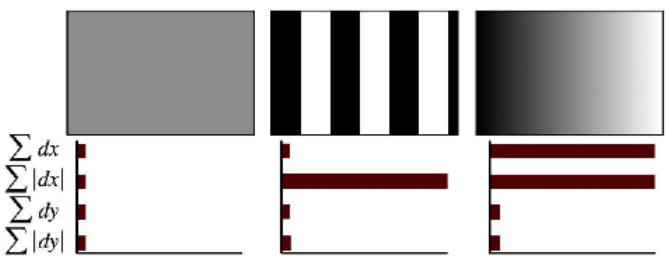


Figure 3.19: Left: Homogeneous region results in a low response due to Haar Wavelet filtering. Middle: If intensity in horizontal direction is frequently changed, the value of $\sum |dx|$ is high and the others remain low. Right: If the intensity is gradually increasing in the horizontal direction, both $\sum dx$ and $\sum |dx|$ will respond with a high value [1].

3.3 Matching

Matching is the procedure where corresponding features in two or more images are detected. Matching is a good way to compare SIFT and SURF to measure robustness and the need for computing power. Since SIFT and SURF are both robust methods, the simple but fast matching method FLANN is used to compare the detected keypoints. FLANN is an acronym for Fast Library for Approximate Nearest Neighbour, and contain a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features [15]. The idea of nearest neighbor matching is to measure distances between features. This means that features located close to eachother will match, and a match is perfect if the distance is equal to zero. This may happen in cases where features have the same location or illustrates a similar object in two or more different images. In Figure 3.20, nearest neighbor matching are illustrated by plotting lines between matches in two equal images.

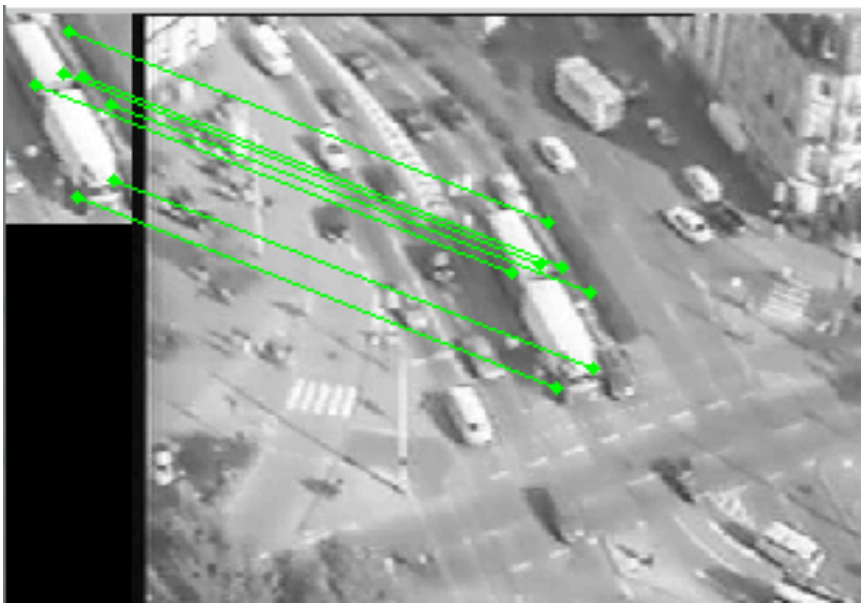


Figure 3.20: Matching two images using SIFT and nearest neighbour.

Chapter 4

Robotics

A KUKA Agilus KR 6 R900 sixx robot was used to grasp the moving object. To interconnect a usb 2D camera, a manipulator and a computer it is necessary to know the forward kinematics of the manipulator and interface protocols of the different systems. In this chapter, the forward kinematics and the operating system ROS is presented.

4.1 Robot Kinematics

The Denavit-Hartenberg parameters are used in the forward kinematics to calculated the Cartesian position and orientation of the gripper. This is described by the The commands sent to the KUKA Agilus is rotation matrix $\boldsymbol{R}$, and the vector $\boldsymbol{p}$, where

$$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \qquad \boldsymbol{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}. \tag{4.1}$$

This can be combined in a 4×4 transformation matrix

$$\boldsymbol{T} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.2}$$

4.2 Denavit-Hartenberg Parameters

A serial-link manipulator consists of a set of links in a chain connected by joints. Each joint has one degree of freedom (1DOF), either translational (a sliding or prismatic joint) or rotational (a revolute joint) [4]. The KUKA Agilus used in this project has six degree of freedom (6DOF), where all joints are revolute. The Denavit-Hartenberg parameters is shown in Table 4.1 and they are specified with respect to the direction of rotation shown in Figure 3.20. The Z axis is pointing downwards in the start position.

Denavit-Hartenberg Parameters				
Link[−]	θ[rad]	d_i[m]	a_{i-1}[mm]	α_{i-1}[rad]
1	θ_1^*	-400	-25	$-\frac{\pi}{2}$
2	θ_2^*	0	315	0
3	θ_3^*	0	35	$\frac{\pi}{2}$
4	θ_4^*	-356	0	$\frac{\pi}{2}$
5	θ_5^*	0	0	$-\frac{\pi}{2}$
6	θ_6^*	-80	0	π

Table 4.1: Denavit-Hartenberg parameters for KUKA Agilus.

These Denavit-Hartenberg parameters are further used to calculate the robots forward kinematics. The transformation from link coordinate frame $\{j-1\}$ to frame $\{j\}$ is defined in terms of elementary rotations and translations as

$$^{j-1}A_j(\theta_j, d_j, a_j, \alpha_j) = T_{Rz}(\theta_j)T_z(d_j)T_x(a_j)T_{Rx}(\alpha_j). \tag{4.3}$$

which can be expanded as

$$^{j-1}A_j(q_j) = \begin{bmatrix} \cos\theta_j & -\sin\theta_j\cos\alpha_j & \sin\theta_j\sin\alpha_j & a_j\cos\theta_j \\ \sin\theta_j & \cos\theta_j\cos\alpha_j & -\cos\theta_j\sin\alpha_j & a_j\sin\theta_j \\ 0 & \sin\alpha_j & \cos\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.4}$$

The parameters α_j and a_j are constants. The joint variables are denoted θ_j, and d_j is constant in a revolute joint [4]. The matrix (4.4) is used on each joint from the robot base to the end-effector, and can be expressed as

$$T_E^W = T_6^0 = A_1^0A_2^1A_3^2A_4^3A_5^4A_6^5. \tag{4.5}$$

This transformation matrix describes the position and rotation of the end-effector as a function of the six joints with respect to the base of the robot.

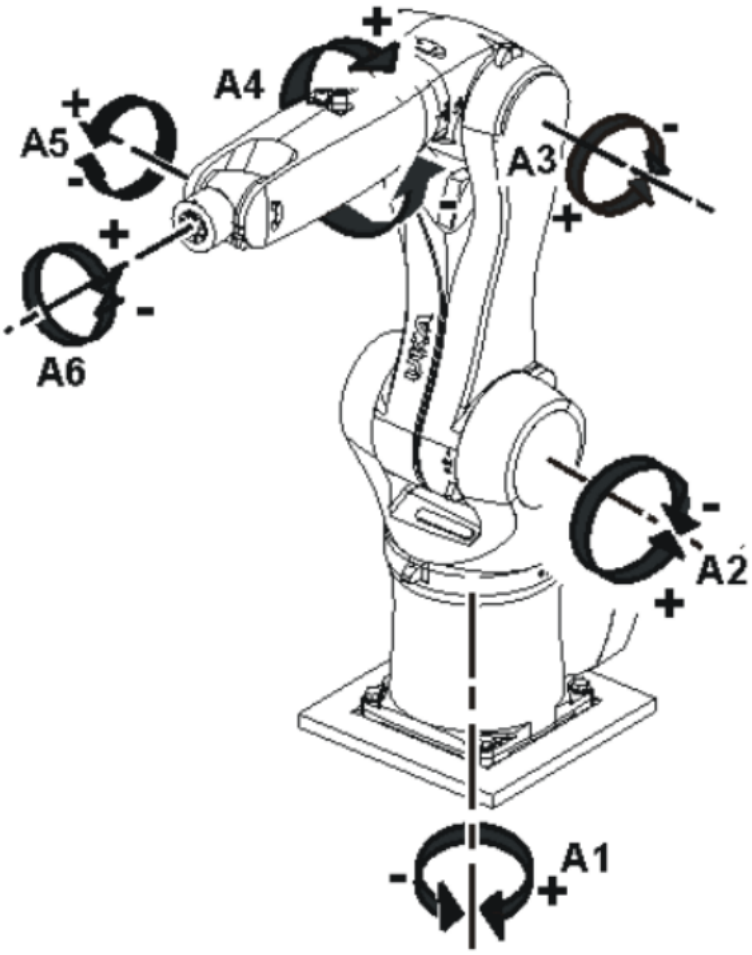


Figure 4.1: Rotation direction of the different joints.

4.3 End-Point Open-Loop Control

End-point open-loop control is used to observe the scene and estimate poses for the manipulator with respect to a tracked object. This is a method where the object, scene and the gripper are observed from a camera fixed at an arbitrary position in the world [4], see Figure 2.5.

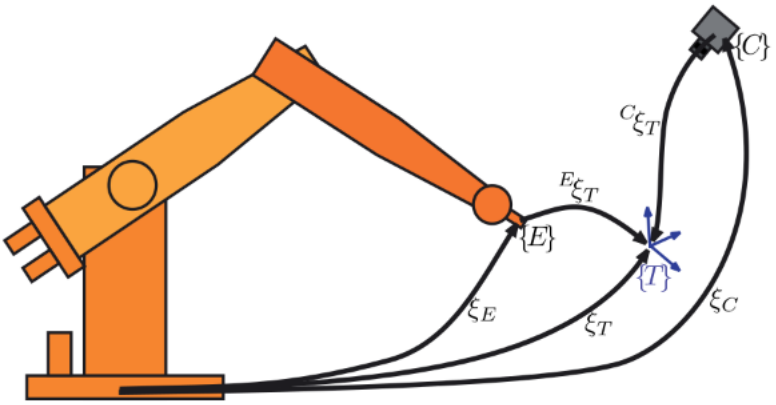


Figure 4.2: End-point open-loop. The camera is fixed in the world and observing a target and the end-effector [4].

4.4 Position-Based Visual Servoing

In this project the *Position-Based Viusal Servoing* (PBVS) approach was applied to estimate poses for the manipulator. PBVS uses observed visual features, a calibrated camera and a known geometric model or to determine the pose of the object with respect to the camera. Positions may then be continuously estimated, while the manipulator moves toward an object. This is illustrated in a control loop in Figure 4.1 [4]. The approach is computationally expensive, and since the tracked object is moving some simplification are done to make the computer work faster in the experiment.

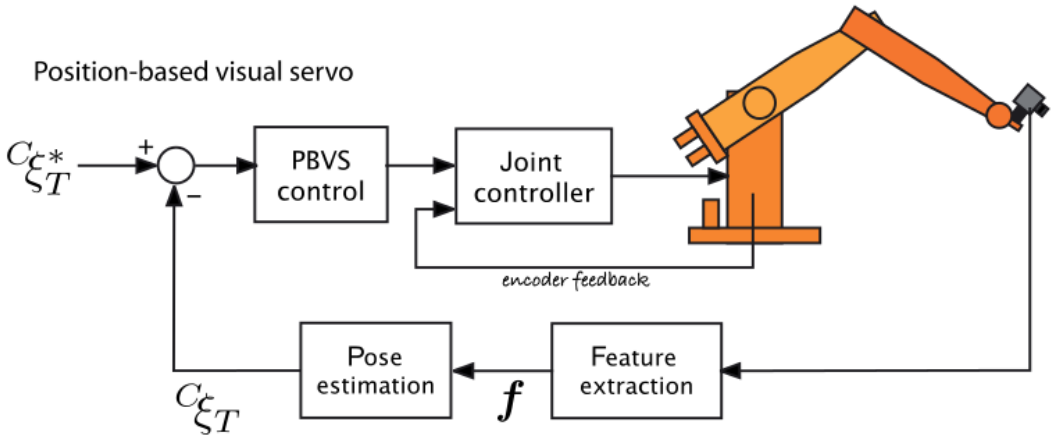


Figure 4.3: Visual servo system with PBVS [4].

4.5 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products [9]. It contains SIFT, SURF and FLANN, has C++ and Python interfaces and support Ubuntu and Windows. It is a well documented library and good utility for implementation of computer vision.

4.6 ROS

The ROS, Robot Operating system is used to communicate between the camera, computer and the robot. ROS is an open source and flexible framework for writing robot software. ROS contains many different tools, libraries, and communicates with several type of robots [18]. ROS also supports C++, Python, and has OpenCV integrated in its library. In ROS, all of the different processes are structured in nodes under a main node called *roscore*. This is a master coordination node, that must be running in order for ROS nodes to communicate. A node is an executable that uses ROS to communicate with other nodes [18], for example web-camera, RSI-communication, a visualisation tool or a simulation tools. An example of a simple communication stream where a camera is passing images to an implementation of SIFT and SURF is illustrated in Figure 4.4. The camera is a node, and represented as an ellipse called "usb_cam". This node publish information to a topic called "usb_camimage_raw". The topic is represented as a box. Nodes can publish data to a topic as well as subscribe to a topic to receive data. In this example, the topic is holding images, and two nodes named "sift" and "surf" are subscribing to this topic. "sift" represents the implementation of SIFT and "surf" represents the implementation of SURF. A more complex communication stream is illustrated in Figure 4.5.

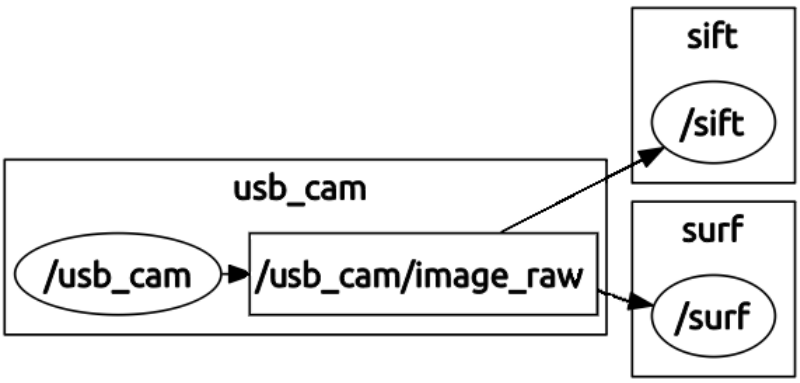


Figure 4.4: How nodes communicates in ROS [4].

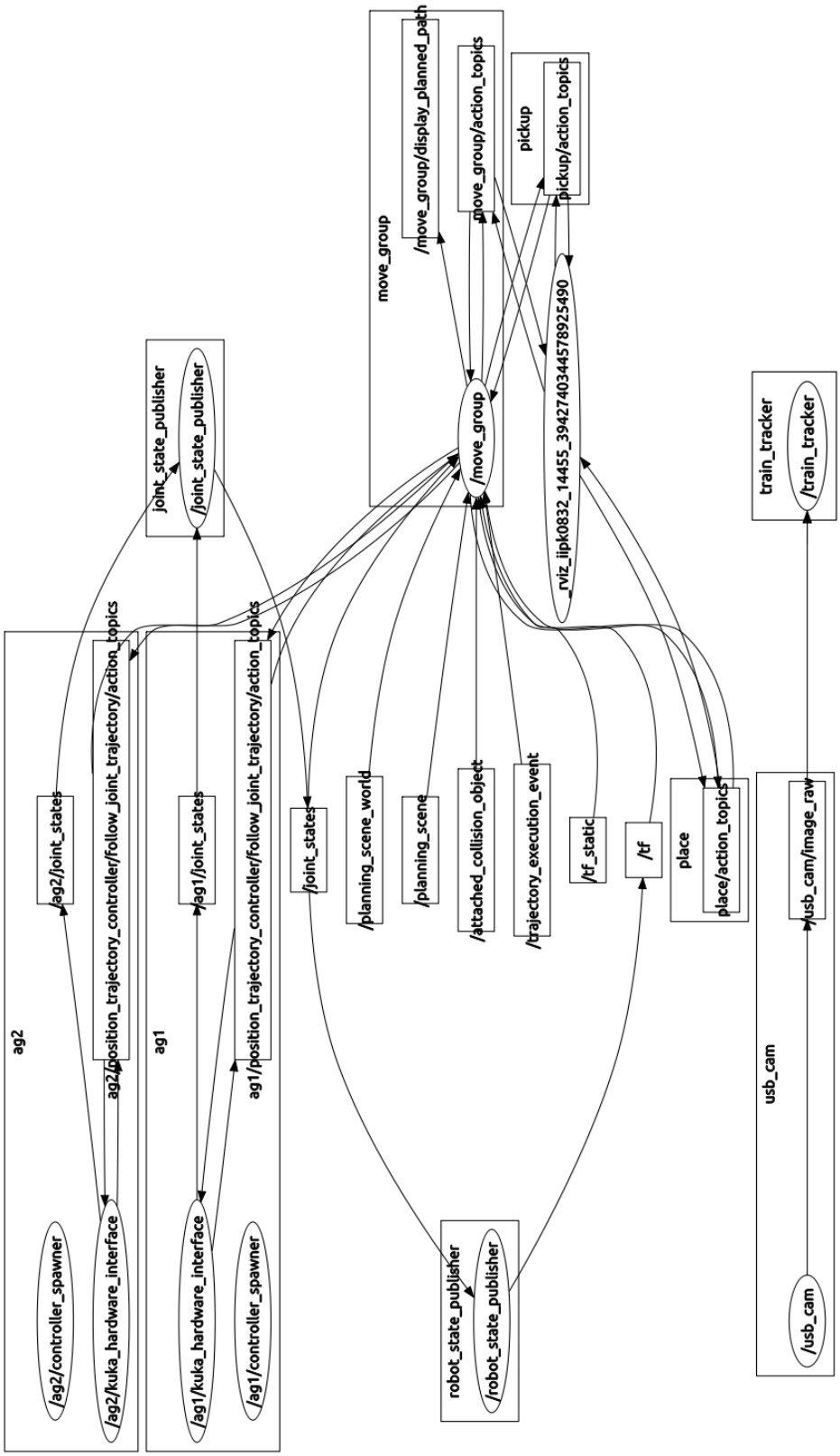


Figure 4.5: Example of a bigger map of nodes.

4.7 Camera Calibration

If a monocular camera (2-Dimensional) is used to measure distances, the intrinsic parameters for the specific camera must be available. This parameters are represented in the camera calibration matrix (2.8) and they are usually unknown. In general the principal point is not at the centre of the photosite array, and the focal length of a lens has only an accuracy of 4%. The focal length is only correct if the lens is focused at infinity. The pixel dimensions ρ_w and ρ_h may be possible to obtain from data sheet deliver by the producer, but the rest have to be determined with a camera calibration tool. Calibration techniques rely on sets of world points whose relative coordinates are known and whose corresponding image-plane coordinates are also known. In this project, these calibration techniques was performed using the ROS integrated *ROS Calibration Tool*. This was a simple method which uses a chessboard whit known square dimensions. The calibration tool gather images of the chessboard in different poses and orientations. These images was then compared and relative poses was estimated. When the bar related to X, Y, *Size* and *Skew* becomes green, the calibration tool had collected enough images and the camera calibration matrix could be estimated (Figure 4.6). All experiments were performed with a Logitech Webcam C930e, and the corresponding calibration matrix K is defined as

$$\boldsymbol{K} = \begin{bmatrix} 516.496569 & 0.000000 & 309.255248 \\ 0.000000 & 516.477238 & 268.673559 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.6}$$

Figure 4.6: ROS Calibration Tool.

Chapter 5

Implementation of SIFT and SURF

This chapter consist of two different experiments. In the first experiment SIFT and SURF were used to track an object in motion. Different data describing the behaviour of the algorithms were logged and the object was changed several times. The purpose of the experiment was to gather data for a comparison and evaluate the methods afterwards. The second experiment was an experiment where the algorithms were used in a practical setting. By combining a 2D camera, a feature detector and a KUKA Agilus manipulator, the manipulator may grasp the object in an estimated position given in world coordinates. SIFT, SURF and FLANN were implemented with C++ using library OpenCV, and the communication was manage by ROS.

5.1 Experiment 1: Comparing SIFT and SURF

The first experiment was a comparison of the methods with respect to speed and robustness. The main purpose was gathering reliable data to evaluate the methods on an equal basis. The experiment was carried out using SIFT and SURF for tracking an object in motion in a real time image stream. To perform an experiment under as equal conditions as possible, the algorithms tracked ten different tags attached to one particular object with constant speed and direction. Threshold limitations were initialized while tracking the first tag. The two tags obtaining best performance was further analyzed in other to measure the computation time for the methods. The computation time included three steps; feature detection, descriptor extraction and matching.

5.1.1 Identify Robustness

Ten tags were selected randomly after a search in Google, where the criteria was distinctive details, see tags in Figure 5.1. The tags were attached to an object in motion, tested one by one. A LEGO train was used as a moving object, where an electrical engine ensured constant velocity and train tracks ensured a constant path. Both methods was tested with respect to scale, rotation and tracking interval, an all behavior was logged to a text file. The train was observed by a camera fixed 0.766 meters above the table, with the image plane facing the scene in parallel with the table, see scene in Figure 5.3.

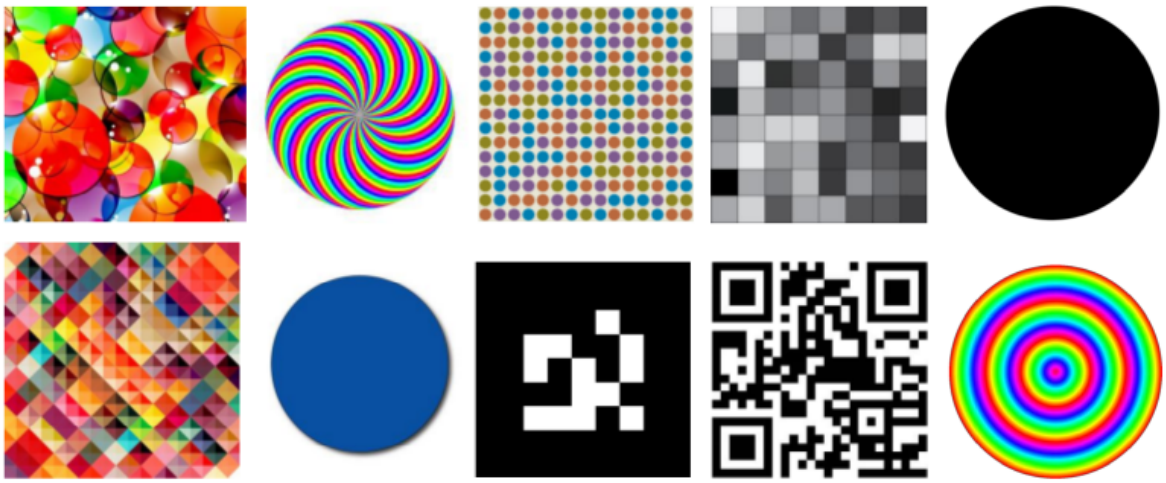


Figure 5.1: Test tags.

The camera was fix in this position during all the experiments to ensure a constant camera view, and reduce disturbance related to angle changes in the camera view. To test the algorithms according to mismatches, only half of the train tracks was within the camera view. This made situations where the train was clearly out of view, and occurrence of mismatches were easy to observe in the plot. Limited distance between the camera and the object was also an advantage to gather good matches and reduce difficulties with tracking. All image captured by the camera was sent (30 fps) to a computer and distributed by ROS to both SIFT and SURF. The ROS administrated communication stream is illustrated in Figure 5.2 and described in Chapter 4.6.

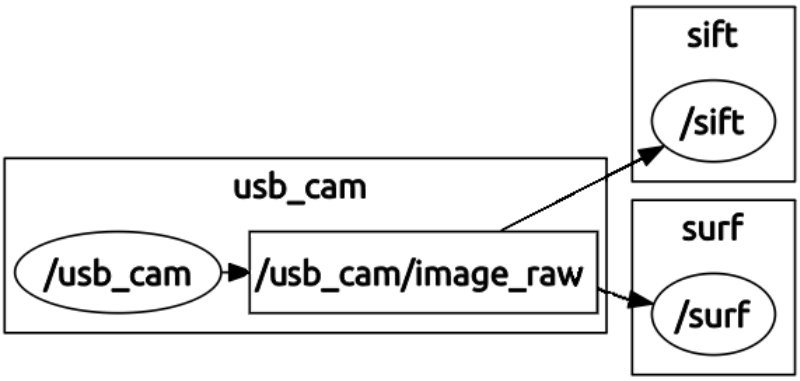


Figure 5.2: Communication between ROS nodes while using SIFT and SURF in experiment 1.

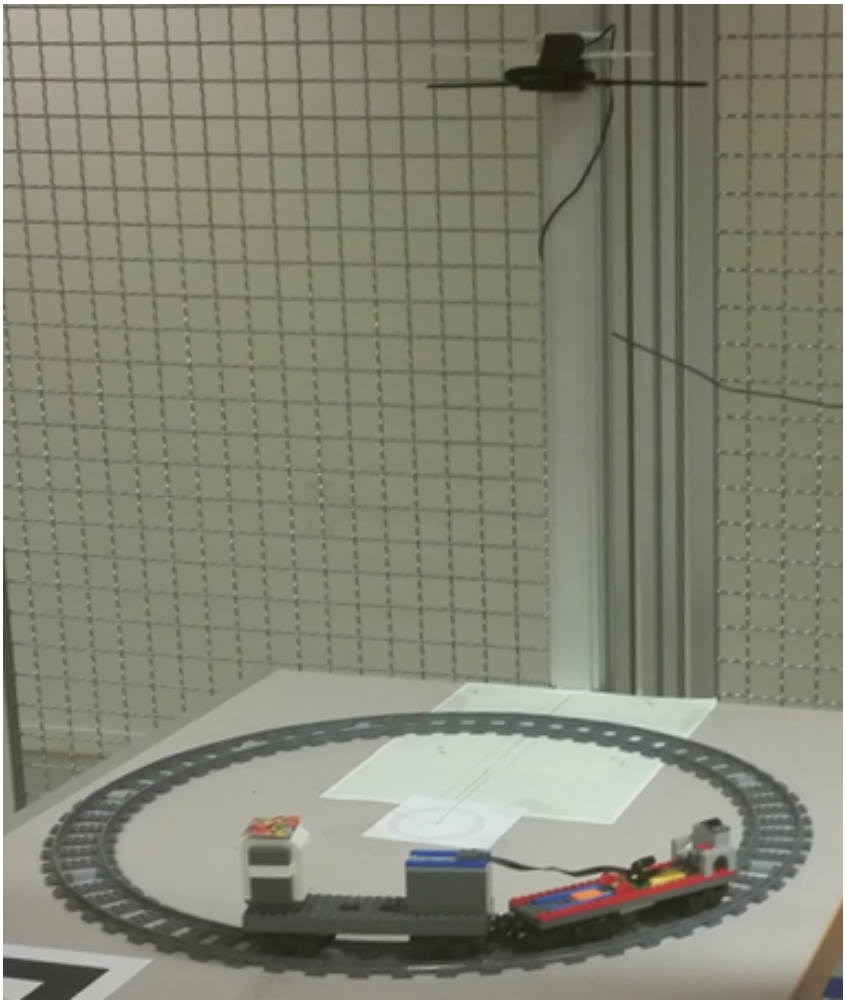


Figure 5.3: Scene of experiment 1.

Figure 5.4: Left: Tag 1 selected from Google. Right: Tag 1 captured with the camera.

```
//Default values
double contrastThreshold = 0.04;
double edgeThreshold = 10;
double sigma = 1.6;

cv::SiftFeatureDetector detector( nFeatures, nOctaveLayers, contrastThreshold, edgeThreshold, sigma );
```

Figure 5.5: SIFT function in OpenCV initialized with recommended values.

```
//Default values
double minHessian = 350; // 300-400 is recomended
int nOctaves=4;
int nOctaveLayers=2;

cv::SurfFeatureDetector detector( minHessian, nOctaves, nOctaveLayers);
```

Figure 5.6: SURF function in OpenCV initialized with recommended values.

The Library of OpenCV includes both SIFT and SURF. The functions and there default values are represented for SIFT in Figure 5.6 and for SURF in Figure 5.5. In SIFT, the *nFeature* is the number of best matches to retain. *nOctaveLayers* is the number of layers in each octave. *contrastThreshold* is the threshold to filter out low contrasts, while edge-like features is filtered out with *edgeThreshold*, see Chapter 3.1.3. *sigma* is the sigma value of the first octave [5]. More details about SIFT in Chapter 3.1. In SURF, the *minHessian* is a limit that reject features with a lower hessian then the defined threshold, see Chapter 3.2.1. *nOctaves* is the number of Gaussian pyramid octaves, which is used by the descriptor. *nOctaveLayers* is the number of images within each octave of a Gaussian pyramid [5]. More details about SURF in Chapter 3.2. SIFT was implemented with default values, but SURF was limited with a hessianThreshold=5000 for to avoid mismatches. All detected features during the test was matched with features from a training image of the current tag. This training image was a camera-captured image of a printed version of the tag. This was necessary to reduce the color difference between the training image and all the camera-captured images during live object tracking, see color differences in Figure 5.4. Each feature was finally matched using matching method FLANN with a maximum Euclidean distance of 90 (three times $min\_dist = 30$) to the nearest neighbor. A strict limitation for detect robust features. Some of the tests are represented in images below with corresponding graph, the rest; 4, 6, 7, 8 and 10, are available in Appendix A.1.

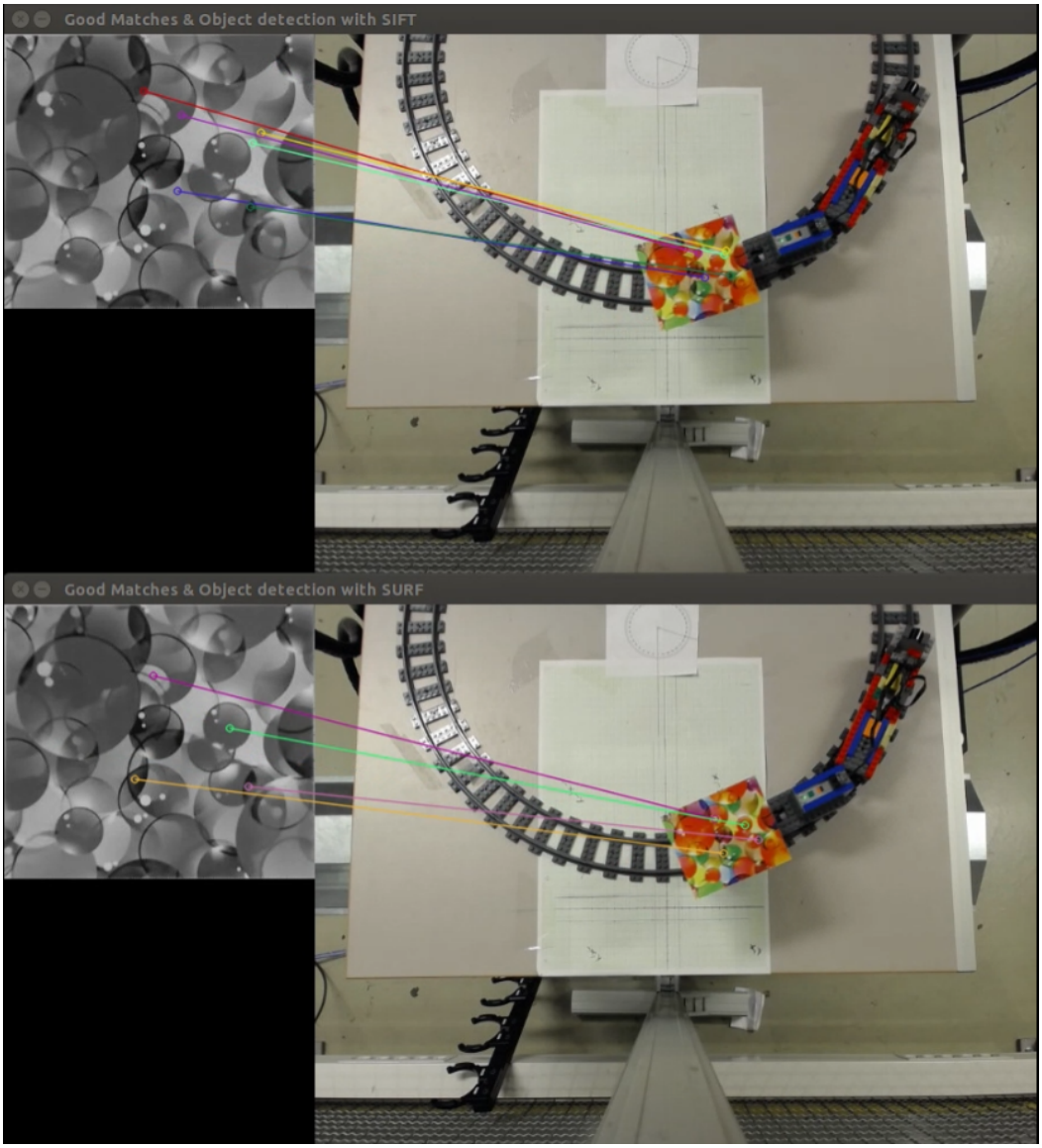


Figure 5.7: Testing tag 1.

The first tag was the reference to the chosen properties for all the tests. SIFT tracked the tag continuous, with no perceptible mismatches and got a maximum of 17 good matches. SURF had rather no perceptible mismatches, and a maximum of seven matches, see Figure 5.7. The results are represented in Figure 5.8.

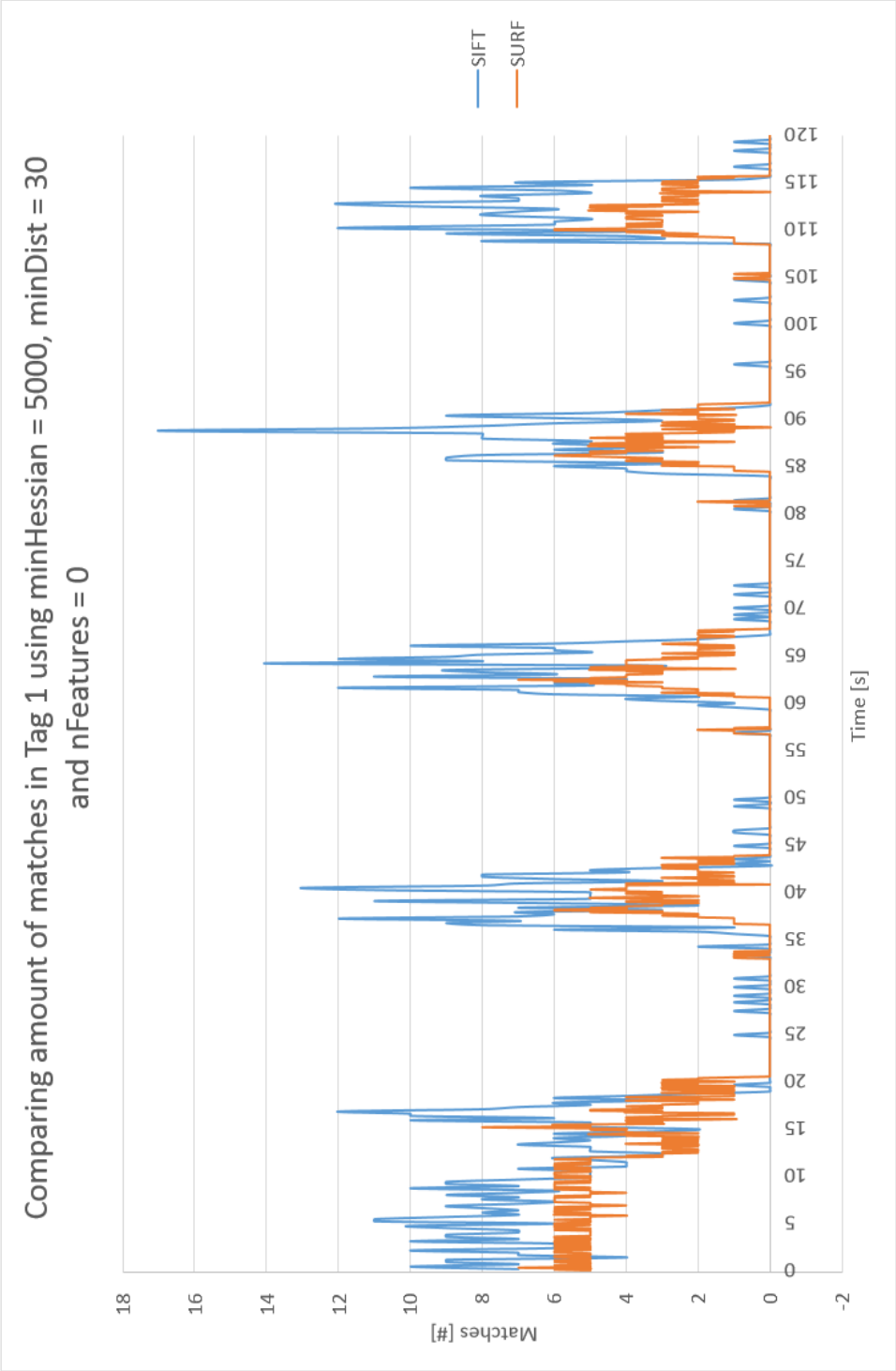


Figure 5.8: Comparing matches in tag 1, analyzed with SIFT and SURF.

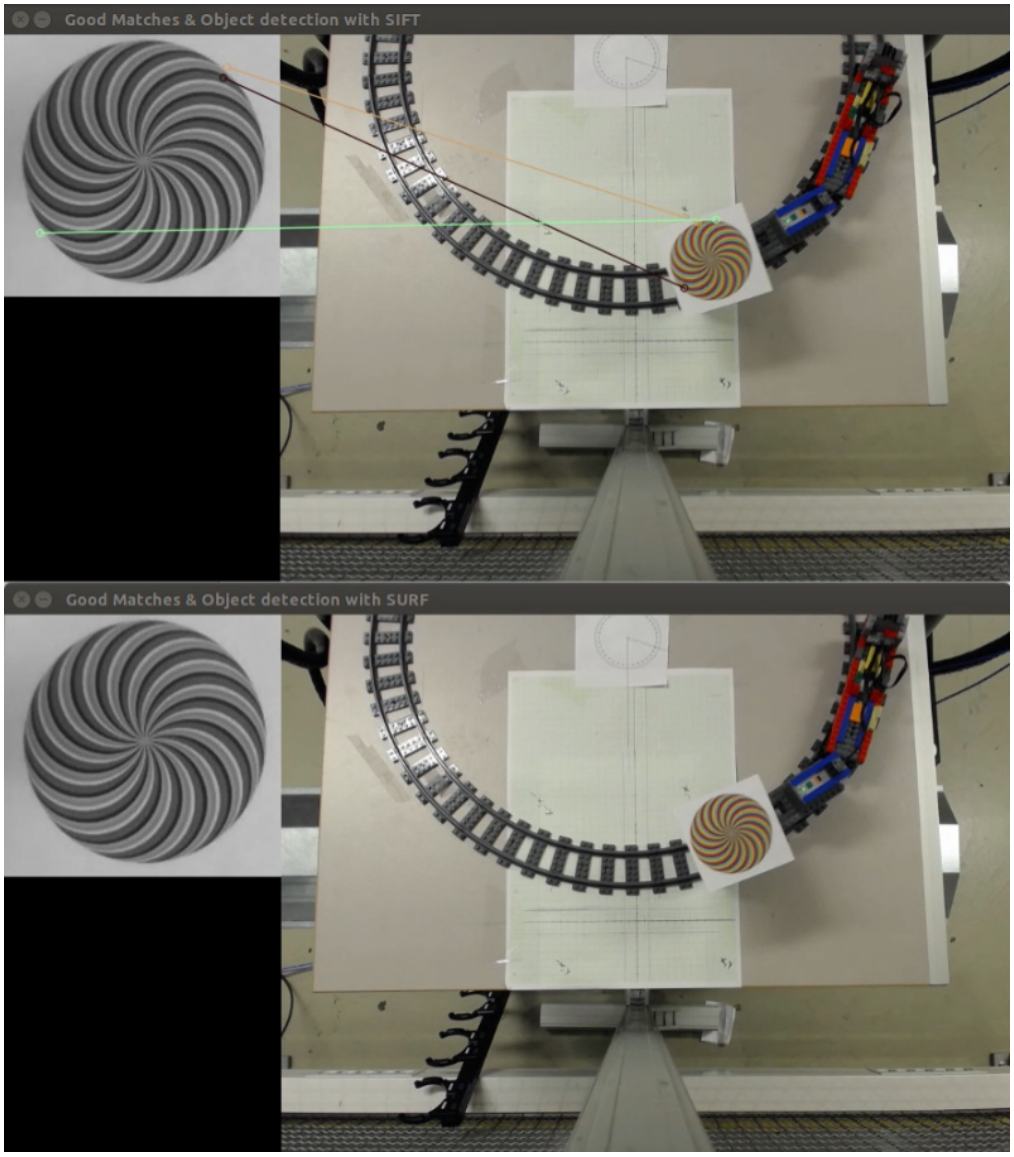


Figure 5.9: Testing tag 2.

The second tag appeared to be difficult to track. SIFT got maximum four matches and the tracking unstable, see Figure 5.10. SURF got no matches at all, see Figure 5.9.

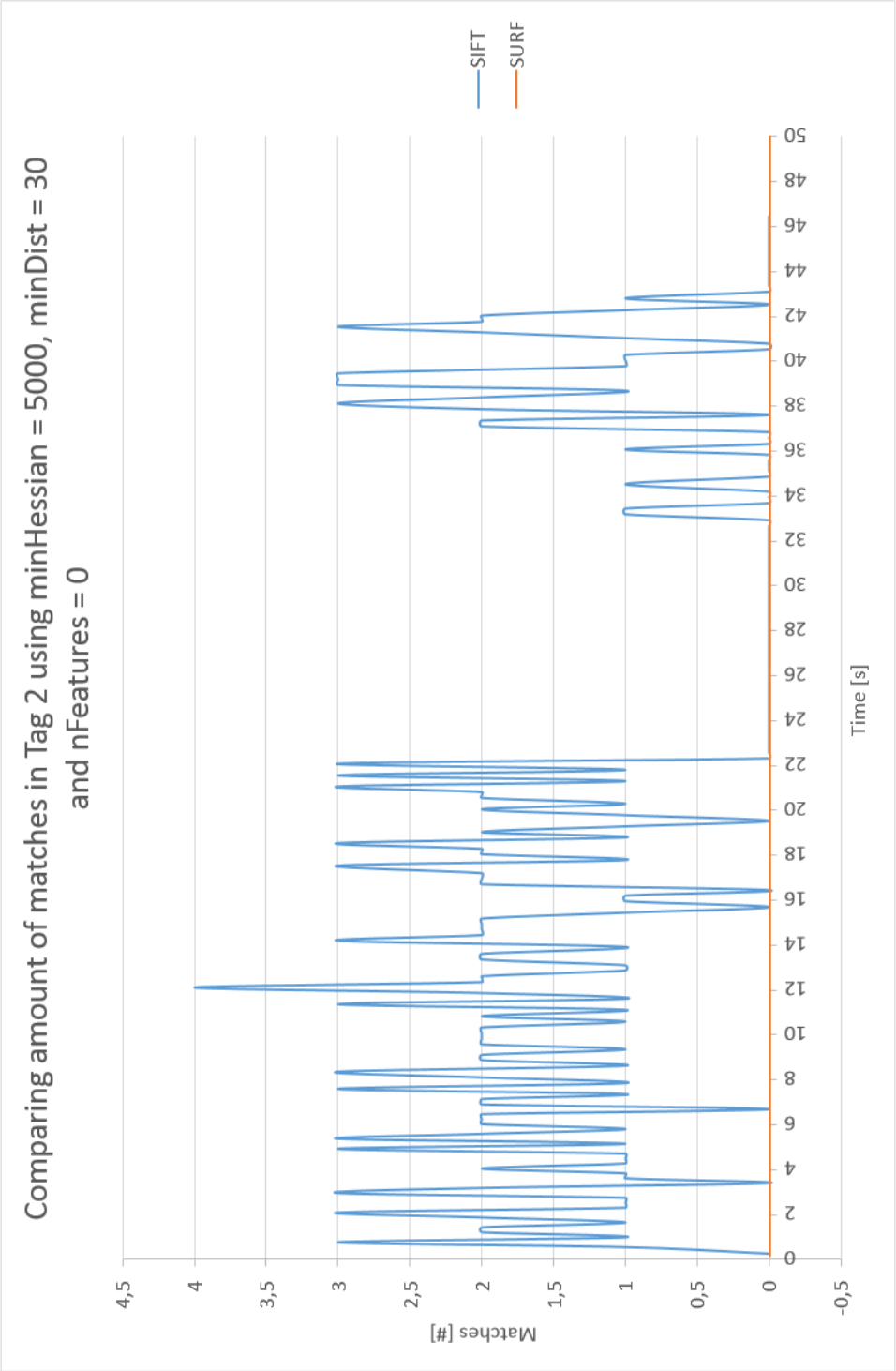


Figure 5.10: Comparing matches in tag 2, analyzed with SIFT and SURF.

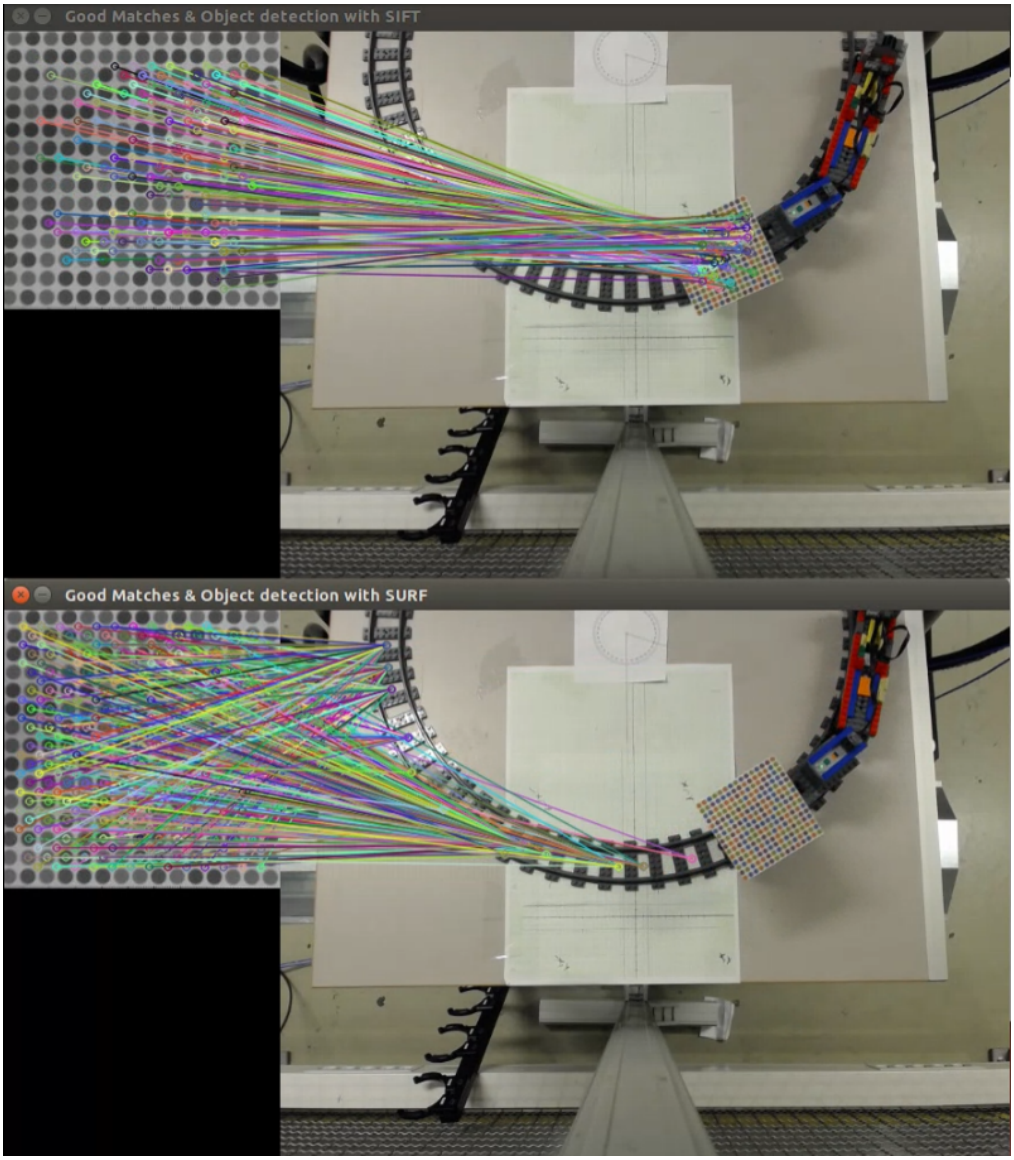


Figure 5.11: Testing tag 3.

The third tag appeared to be very easy for SIFT to track. Over 350 matches were counted. For SURF, the situation was opposite. SURF got several mismatches. Figure 5.11 clearly represents how SURF got contentious mismatches even when the tag was outside the camera view, see graph in Figure 5.12.

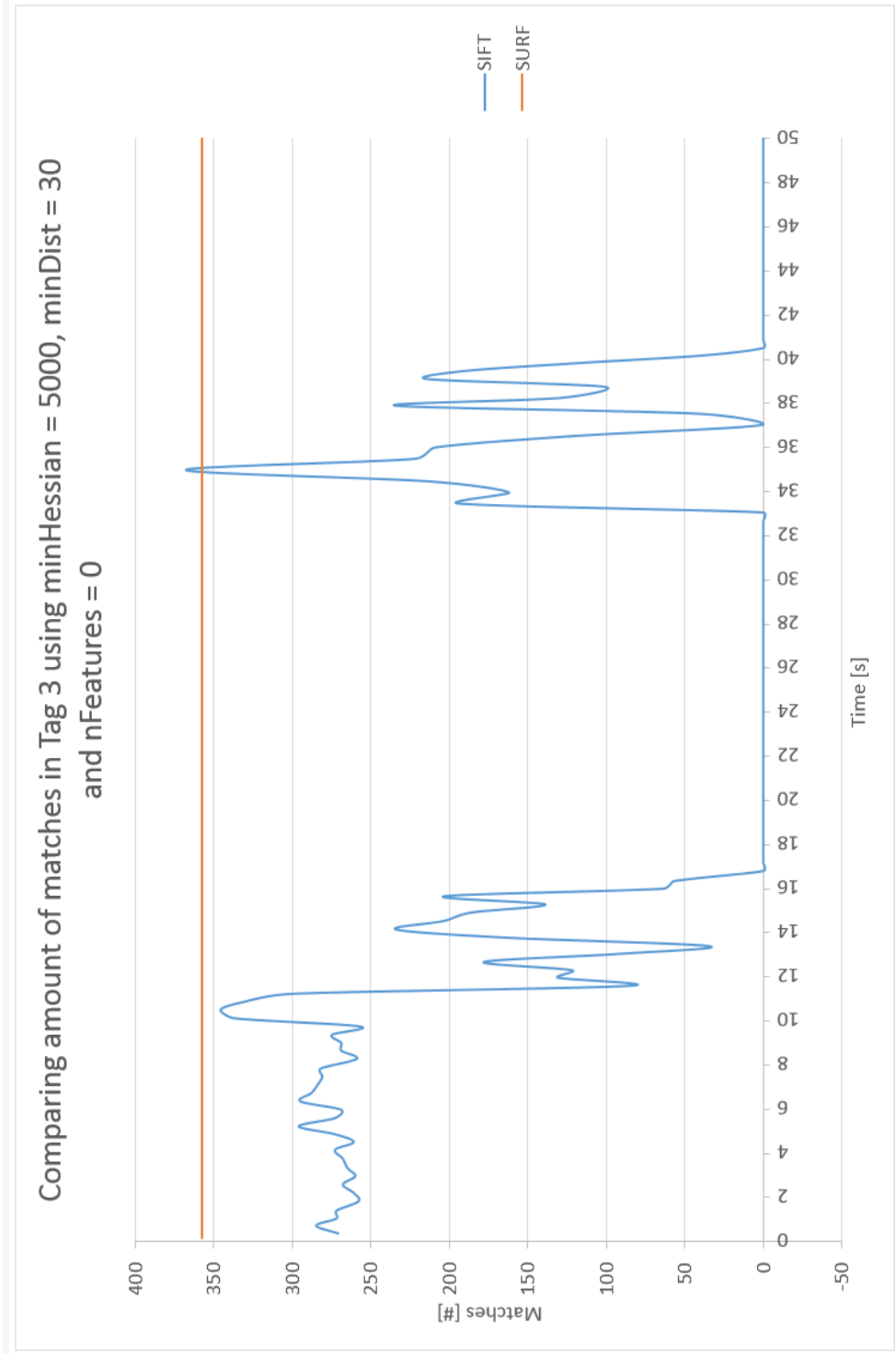


Figure 5.12: Comparing matches in tag 3, analyzed with SIFT and SURF.

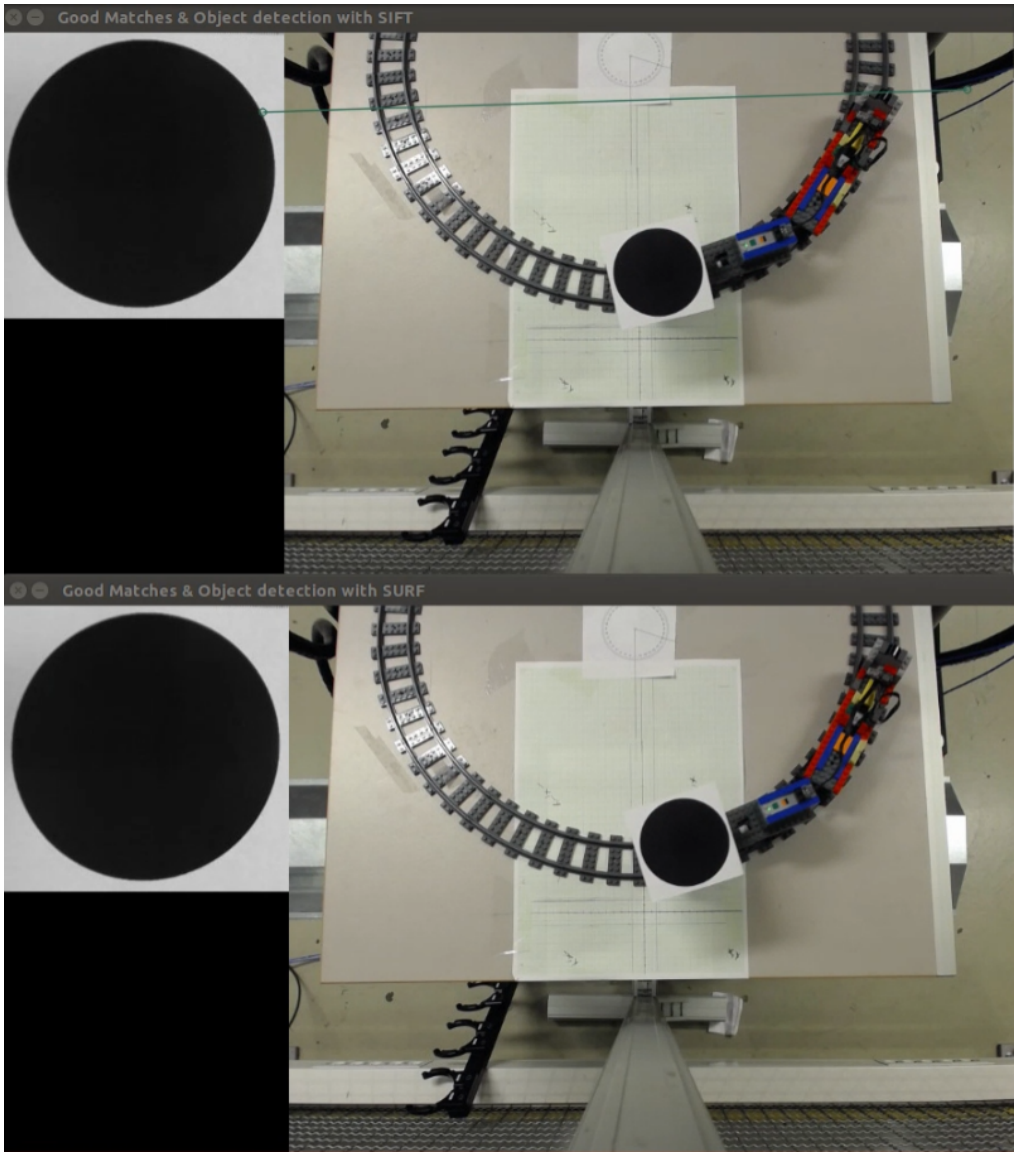


Figure 5.13: Testing tag 5.

The fifth tag appeared to be very difficult to track, see Figure 5.13. Neither SIFT or SURF got feature matching, and the tag was observed to be very shine and illumination effected. The tag was black painted and easily disturbed by light reflection. Only mismatches was registered, see Figure 5.14.

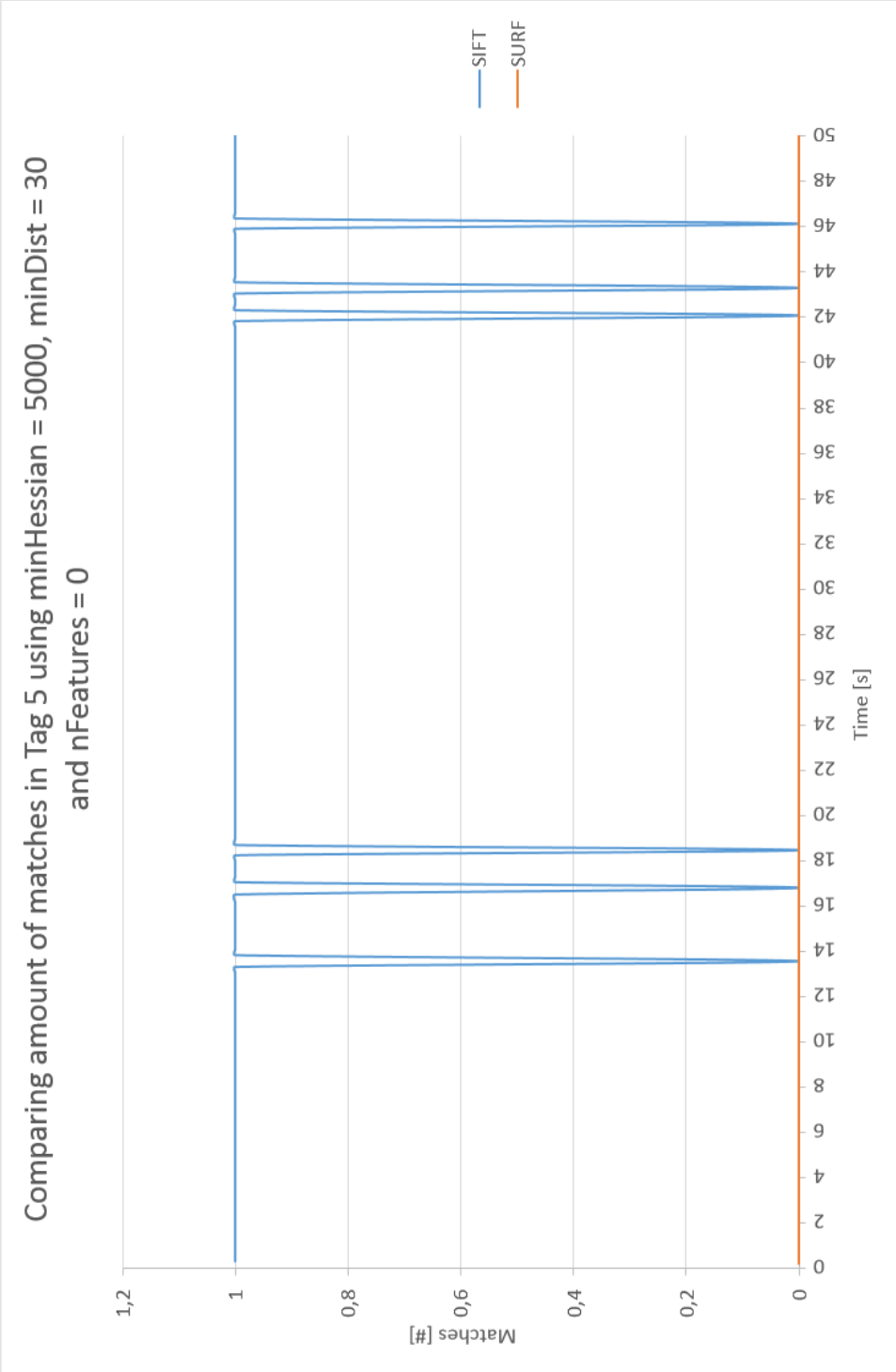


Figure 5.14: Comparing matches in tag 5, analyzed with SIFT and SURF.

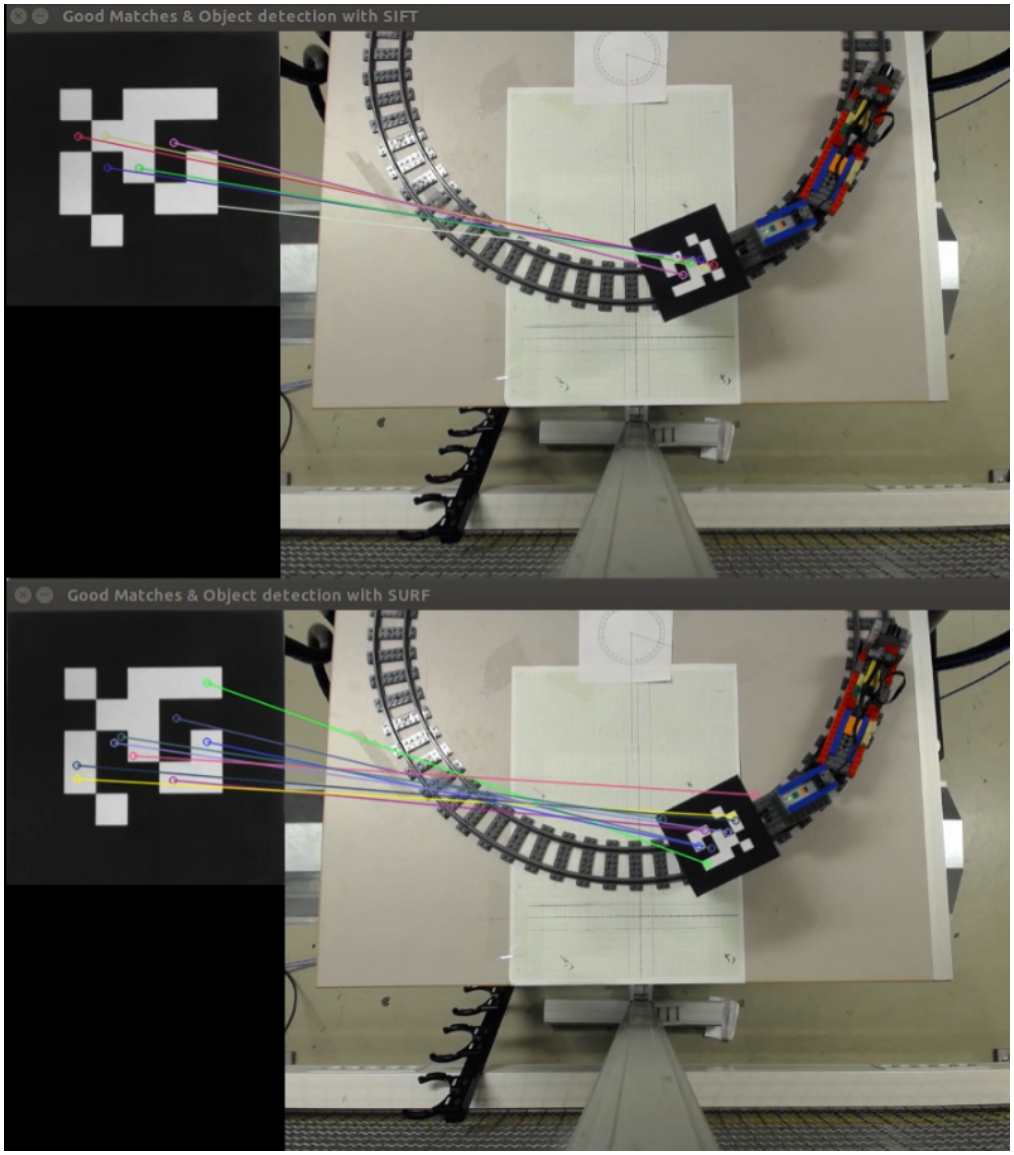


Figure 5.15: Testing tag 9.

The ninth tag appeared to be an easy tag to track, see Figure 5.15. SIFT got 16 matches, SURF got 15 matches and none mismatches was observed, see Figure 5.16. The tracking was stable, and contained for a long period.

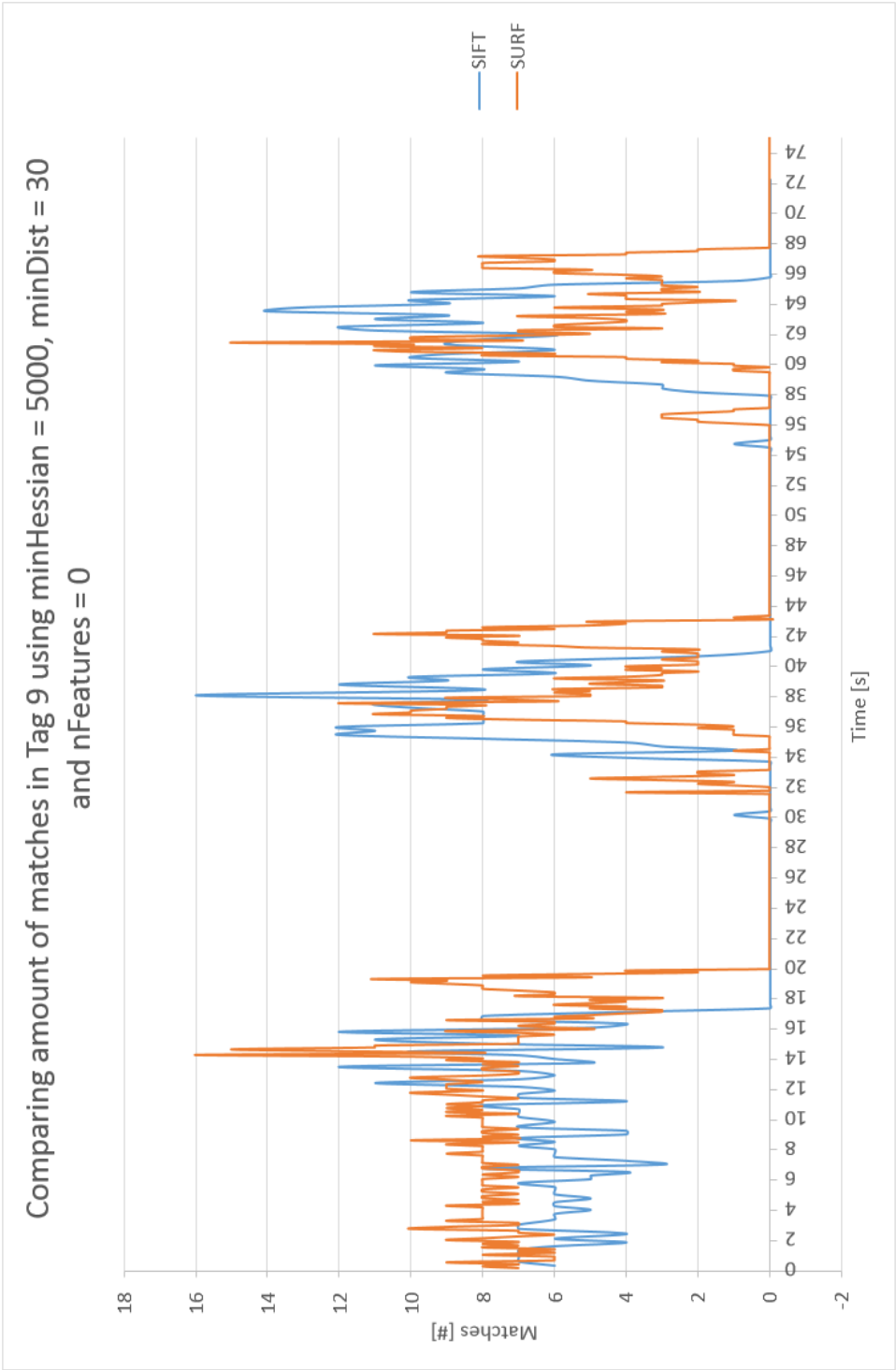


Figure 5.16: Comparing matches in tag 9, analyzed with SIFT and SURF.

5.1.2 Computation Time

According to the developer of SURF, the Fast Hessian integrated in the algorithm is three times faster than the DoG integrated in SIFT [2]. By implementing SURF instead of SIFT the total time of the three steps; detection, description and matching should be performed significantly faster [1]. This was tested by tracking tag 1 and tag 9 with constraints that forced both methods to detect a similar average amount of features. The time spent on the three steps were logged and compared. The detected number of features were decided to be 440 and 689 in tag 1, and 166 and 916 in tag 9. Figure 5.17, 5.18 and 5.19 descried behavior related to 440 features in tag 1. The rest of the graphs are available in Appendix A.2.

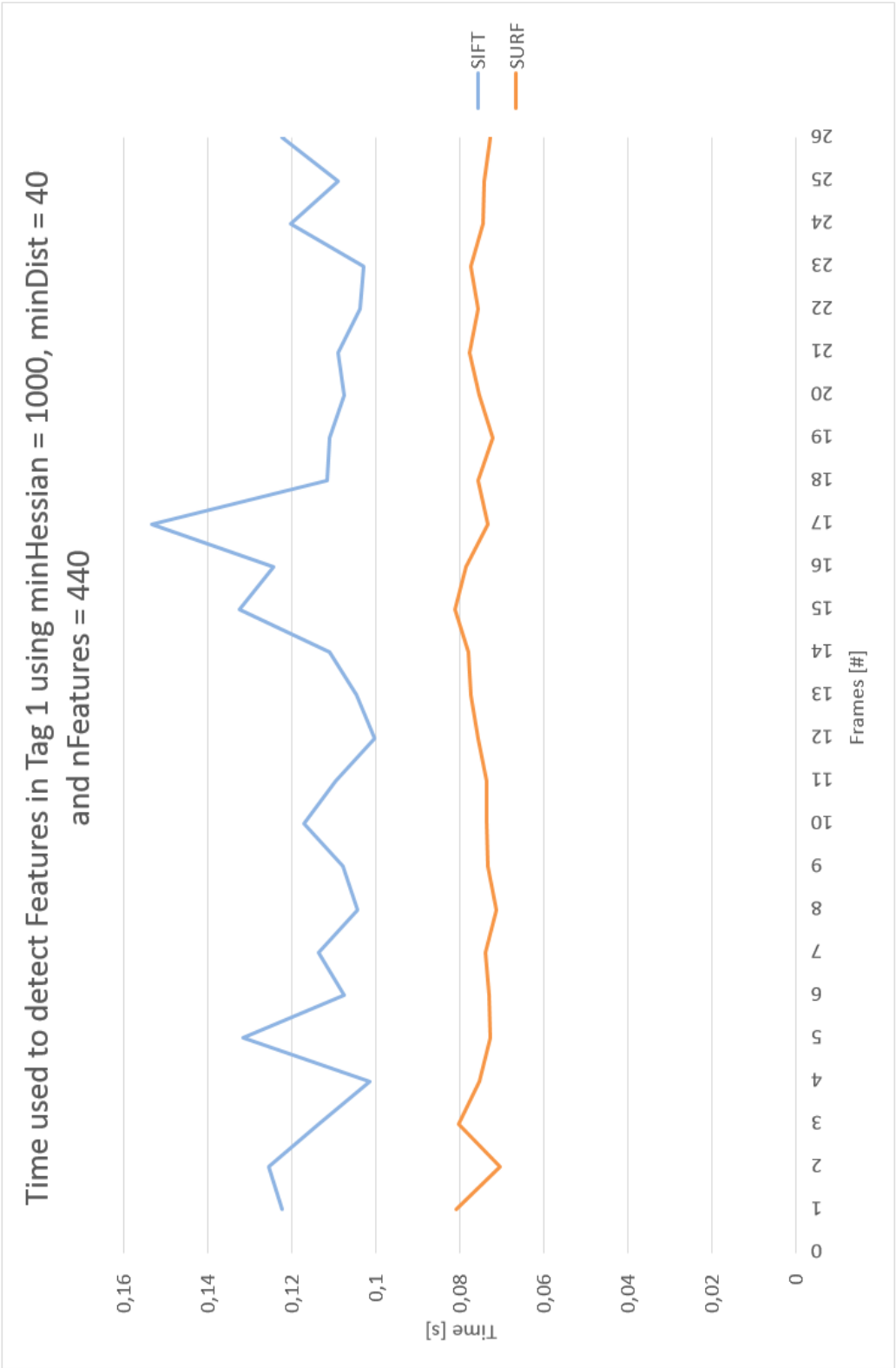


Figure 5.17: Time used to detecting approximately 440 features using tag 1.

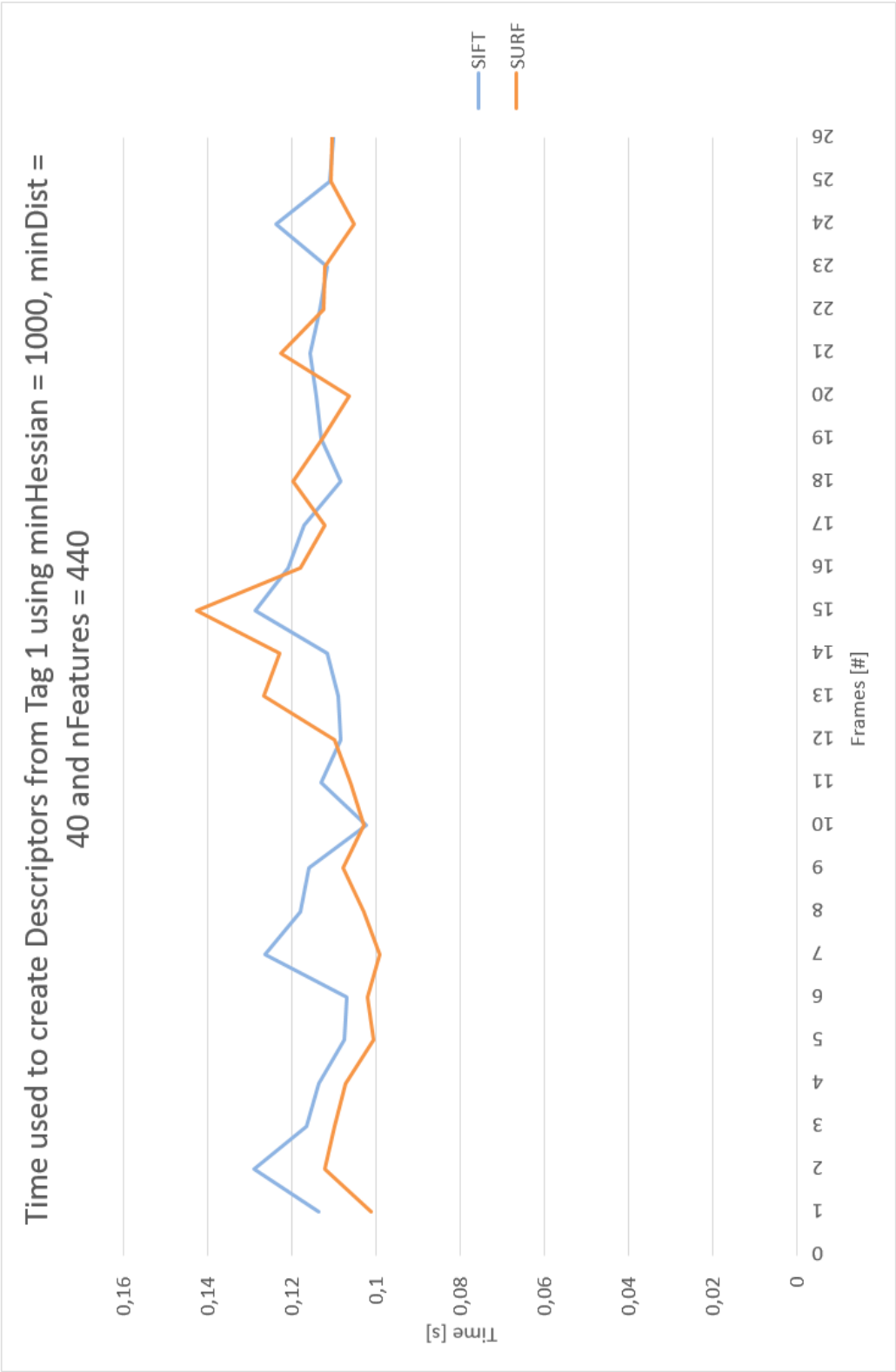


Figure 5.18: Time used to descriptor extraction of approximately 440 features using tag 1.

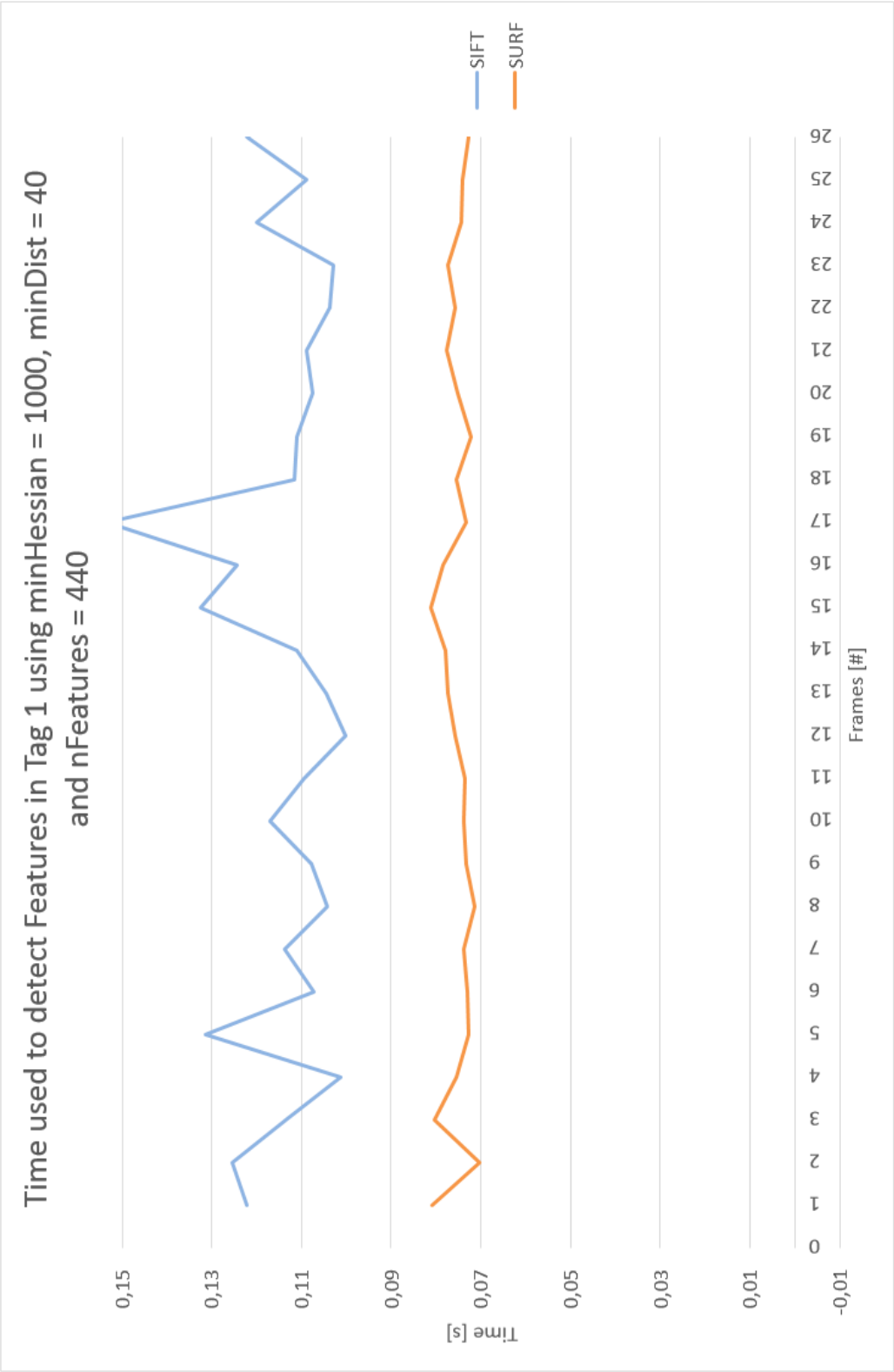


Figure 5.19: Time used to match approximately 440 features suing tag 1.

5.2 Experiment 2: Grasping an object in motion with a KUKA Agilus Manipulator

In experiment 2, SIFT and SURF were used in a practical experiment. The methods were used to detect positions of a tag attached to an object in motion, and captured with a 2D camera. The position was convert to Cartesian world coordinates and transmitted to a KUKA Agilus Manipulator. By estimating a velocity of the tag and an advanced position along the train track, the KUKA Agilus could grasp the moving object at an estimated position. Scene of experiment 2 is represented in Figure 5.20.

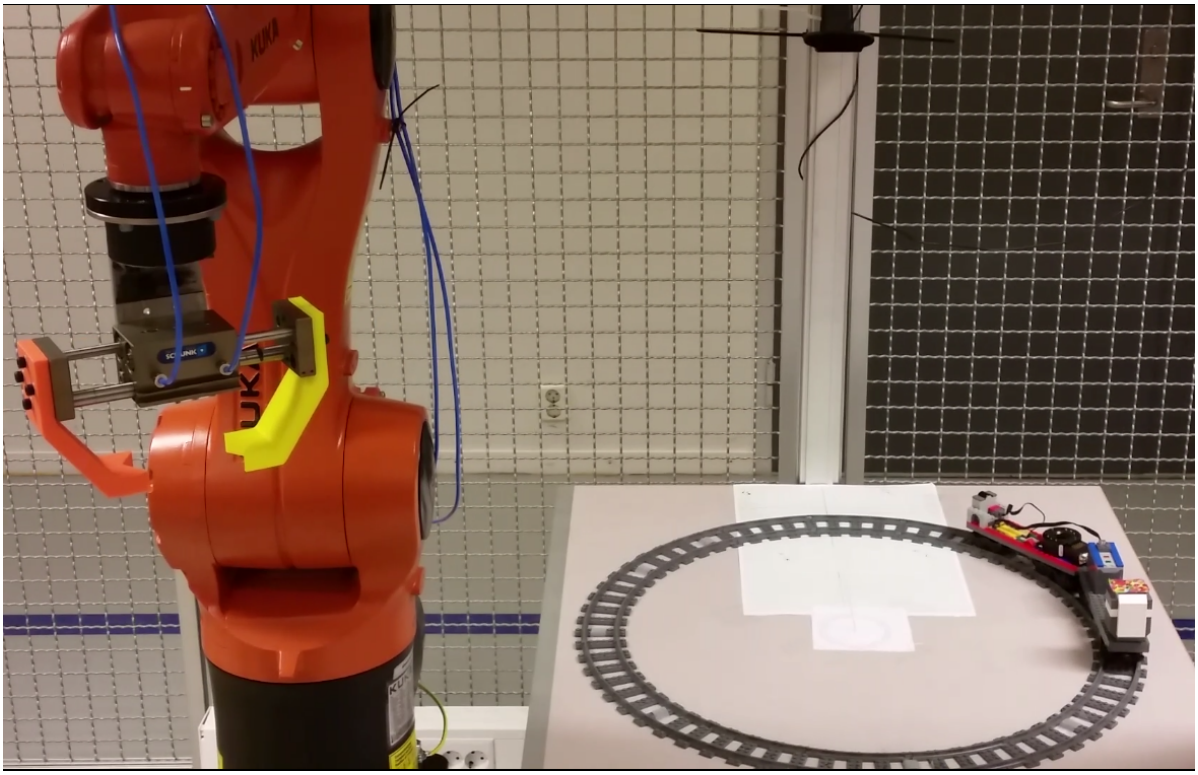


Figure 5.20: Scene of experiment 2.

5.2.1 Tag detection

The first step was detecting features in the camera-captured images. SIFT and SURF were implemented using ROS, similar as in experiment 1. FLANN was used as matching method, and all matches were evaluated according to a predetermined Euclidean distance. The matching features were stacked in an array, and a pixel coordinate was estimated with respect to the average of the stack. This estimated pixel coordinate $\tilde{\boldsymbol{p}}(u, v)$ was approximately the tag center, and the image coordinate $\boldsymbol{p}(x, y)$ could be converted with formula

$$\underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{\boldsymbol{p}} = \boldsymbol{K}^{-1} \underbrace{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}_{\tilde{\boldsymbol{p}}}, \tag{5.1}$$

where $\boldsymbol{K}$ is defined as (4.6). By scaling the image coordinates with the distance from camera to the tag, Z, the unknown world coordinate $\boldsymbol{P}(X, Y, Z)$ could with formula

$$X = \frac{x}{f}Z, \;\; Y = \frac{y}{f}Z. \tag{5.2}$$

This scaling was necessary to make a correspondence between distances in the image plane and distances in the world. Since the metric focal length was unknown, the experiment was simplified by using the normalized image plane where $f = 1$.

5.2.2 Estimate Transformation

The next step was to estimate the transformation from the manipulator to the tag. Since the principal point is not necessarily defined in the camera center (4.7), a circle was implemented to illustrate the principal point, according to the coordinate defined in the camera calibration matrix, see expression 4.6. The circle was redrawn on the table to make a reference point related to the image plane. To define transformation between the robot base and the tag, $\boldsymbol{T}_{Tag}^{W}$, the transformation was divided into three steps. The first transformation was between the robot base and the closest table corner, $\boldsymbol{T}_{B}^{W}$. By running the robot gripper to the corner, the robot translation was displayed on the *smartPAD*, and a rotation could be defined according to the robot base. $\boldsymbol{T}_{B}^{W}$ was expressed as

$$\boldsymbol{T}_{B}^{W} = \begin{bmatrix} 0 & -1 & 0 & 0.67562 \\ 1 & 0 & 0 & 0.20626 \\ 0 & 0 & 1 & 0.17484 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.3}$$

Next step, was estimating the transformation from the table corner to the principal point drawn on the table, $\boldsymbol{T}_{Pp}^{B}$. This was determined by a rotation of 180° around Y axis and a measured translation along the table in x and y direction. The 180° rotation was added since the gripper and the camera were rotated 180° with respect to the robot base. $\boldsymbol{T}_{Pp}^{B}$ is expressed as

$$\boldsymbol{T}_{Pp}^{B} = \begin{bmatrix} -1 & 0 & 0 & 0.391 \\ 0 & 1 & 0 & 1.445 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.4}$$

The transformation between the robot base and the principle point was then expressed as

$$\boldsymbol{T}_{Pp}^{W} = \begin{bmatrix} 0 & -1 & 0 & -076938 \\ -1 & 0 & 0 & 0.59726 \\ 0 & 0 & -1 & 0.17484 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.5}$$

The last transformation was between the principle point and the tag, $\boldsymbol{T}_{Tag}^{Pp}$. Since the camera and the gripper operates in a parallel coordinate system, the tag position could be continuously defined by tag detection using the camera, see previous section 5.2.1. The total transformation of $\boldsymbol{T}_{Tag}^{W}$ is illustrated in Figure 5.23.

5.2.3 Masks

Masks were implemented to limit the feature detection to a predefined pixel area in the image plane. Avoidance of mismatches was the purpose. Since the tag center was an estimation of the average position of the best matches, outliers was preferred to be avoided for making the position as accurate as possible.

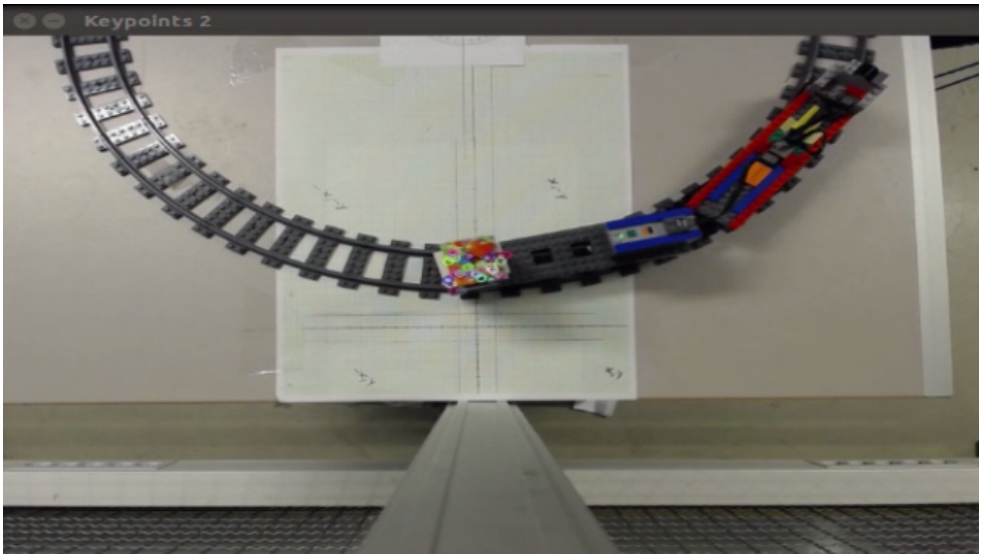


Figure 5.21: A mask defined to detect features in the starting position. From pixel 290 to 330 in u direction (horizontal), and from 200 to 240 in v direction (vertical).

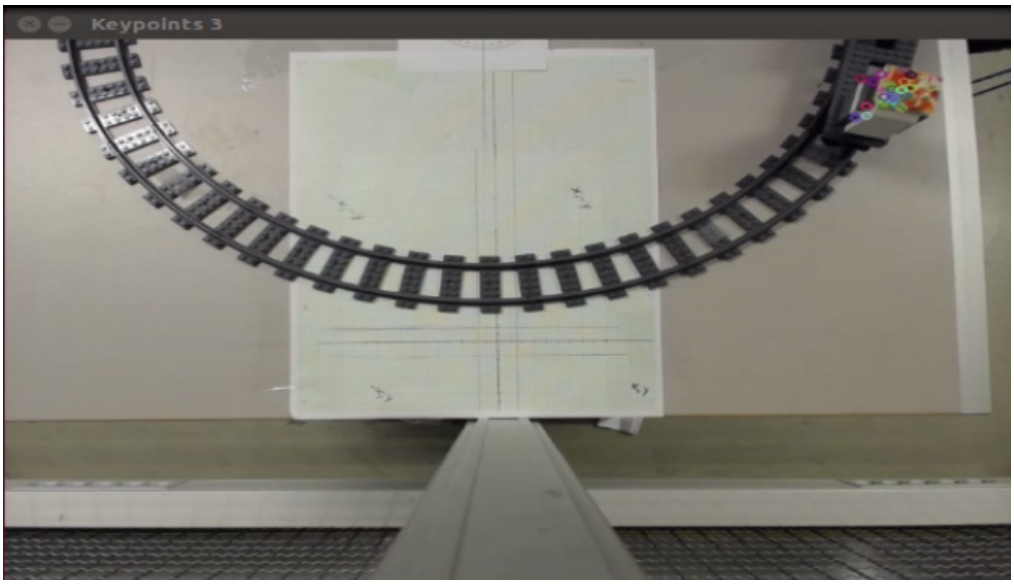


Figure 5.22: A mask defined to detect features in the end position. From pixel 530 to 570 in u direction (horizontal), and from 30 to 70 in v direction (vertical).

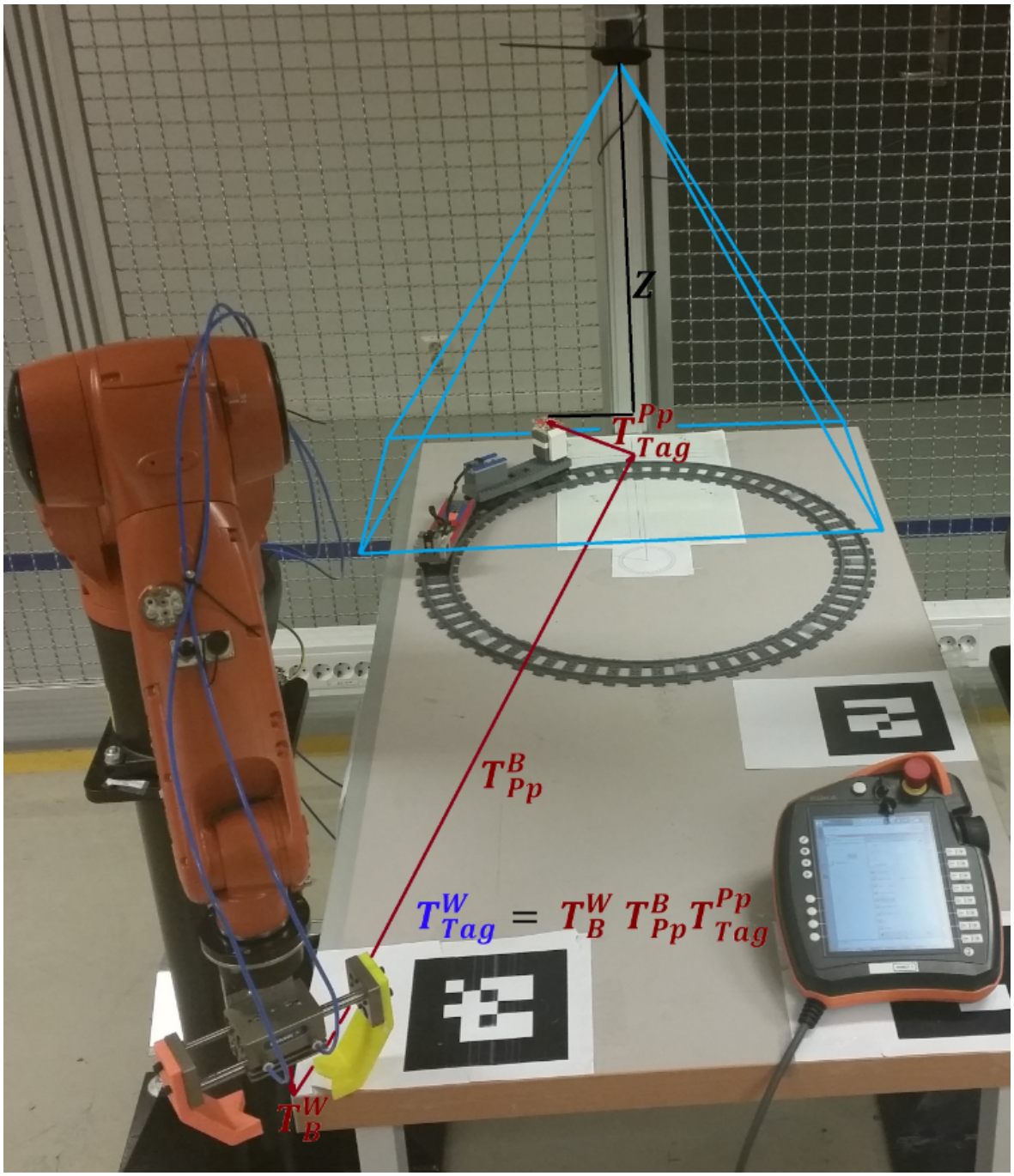


Figure 5.23: Scene in experiment 2, with an illustration of the camera view in blue, the transformations from robot gripper to the tag, Z and a smartPAD.

5.2.4 Angle Velocity Estimation

Next step was estimating velocity of the train, by measuring traveling time between two positions with respect to the centre of the train track circle (TC). The different tag positions was given with respect to the camera, and the transformation from the tag to the centre of the train track circle had to be estimated via the principle point. This was performed by estimating the transformation between the tag and the principle point, $\boldsymbol{T}_{Pp}^{Tag}$, and then the transformation between the principle point and the center of the train track circle $\boldsymbol{T}_{Pp}^{TC}$. The total transformations are represented in Figure 5.24.

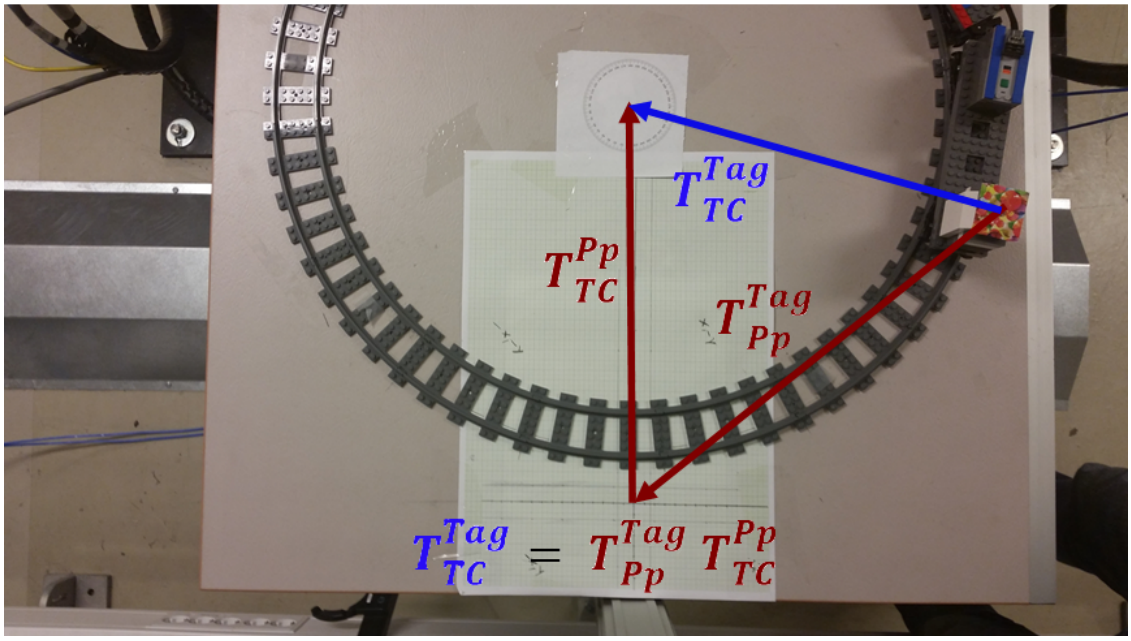


Figure 5.24: Transformation from the tag to the center of the train track circle via the principle point.

The angle velocity was then determined by initializing a stopwatch, and measure time during a predefined interval of detection. For example in ten images. The tag position in the first and the last images was then used to estimate the angle and the curve length along the train track. The angle was estimated by using isosceles triangle theorem and Pythagorean theorem, where two equal triangle sides are constructed by the radius of the train track circle, see Figure 5.25. The angle was determined by using equation (5.6) and then (5.7).

$$|BC| = \sqrt{(x_B - x_C)^2 + (y_B - y_C)^2}, \tag{5.6}$$

$$\theta = 2 \arcsin\left(\frac{|BC|}{2r}\right). \tag{5.7}$$

The train was observed to have some sequential irregular movement, which complicated the time estimation. To make the effect of the movement as small as possible, the lap time was used instead of a shorter interval. This gives an average velocity, and the effect of irregular movements can be neglected. The average angle velocity was estimated with formula

$$\omega = \frac{2\pi r}{t_{tot}}, \tag{5.8}$$

and the travelled distance (curve length) was estimated with

$$l = \theta r. \tag{5.9}$$

When both angle velocity and curve length were determined between the two positions, the time for grasping the object could be estimated with formula

$$t_l = \frac{l}{\omega}. \tag{5.10}$$

In this experiment, the object was grasped at the same position as the tag was last detected. This position is represented as $C(x_C, y_C)$ in Figure 5.25, and the coordinate was immediately transmitted to the robot after estimation, to ensure that the robot had enough time to arrive the new position.

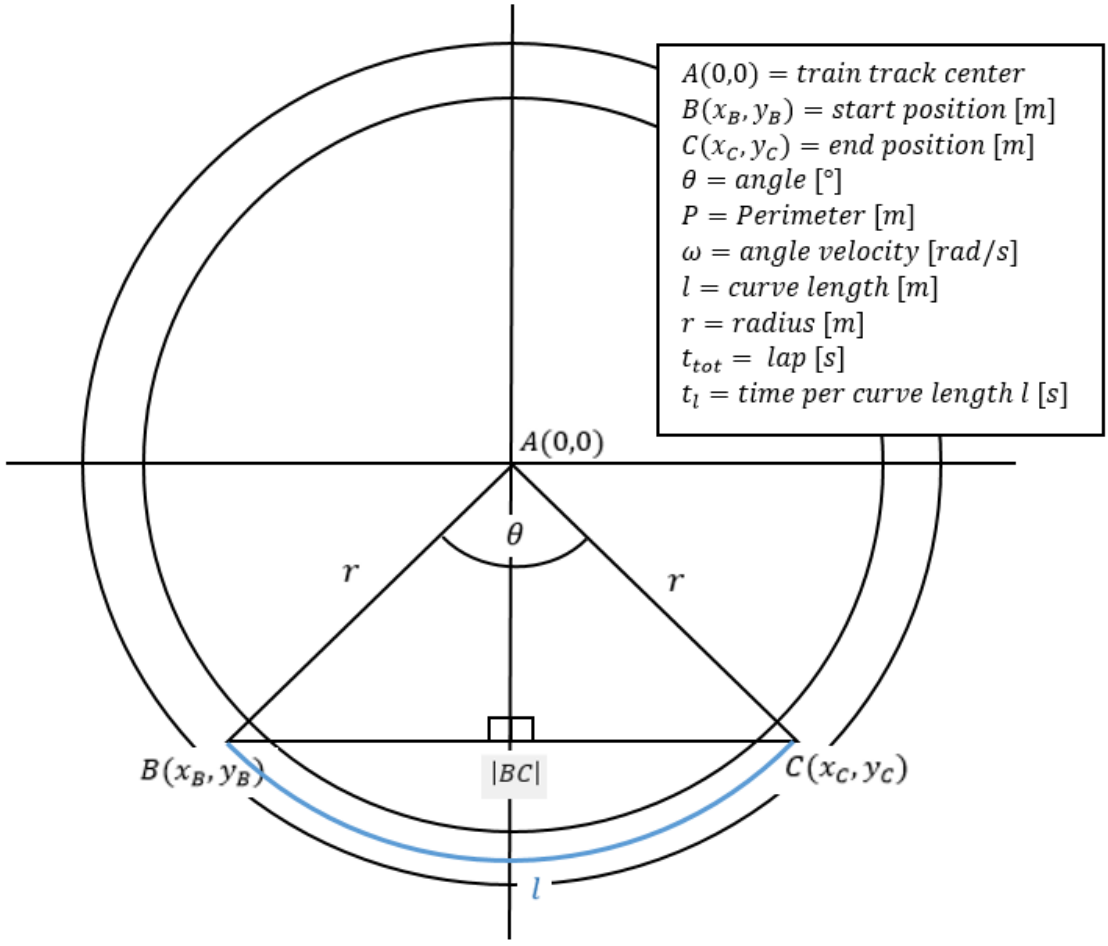


Figure 5.25: Estimation of angle velocity.

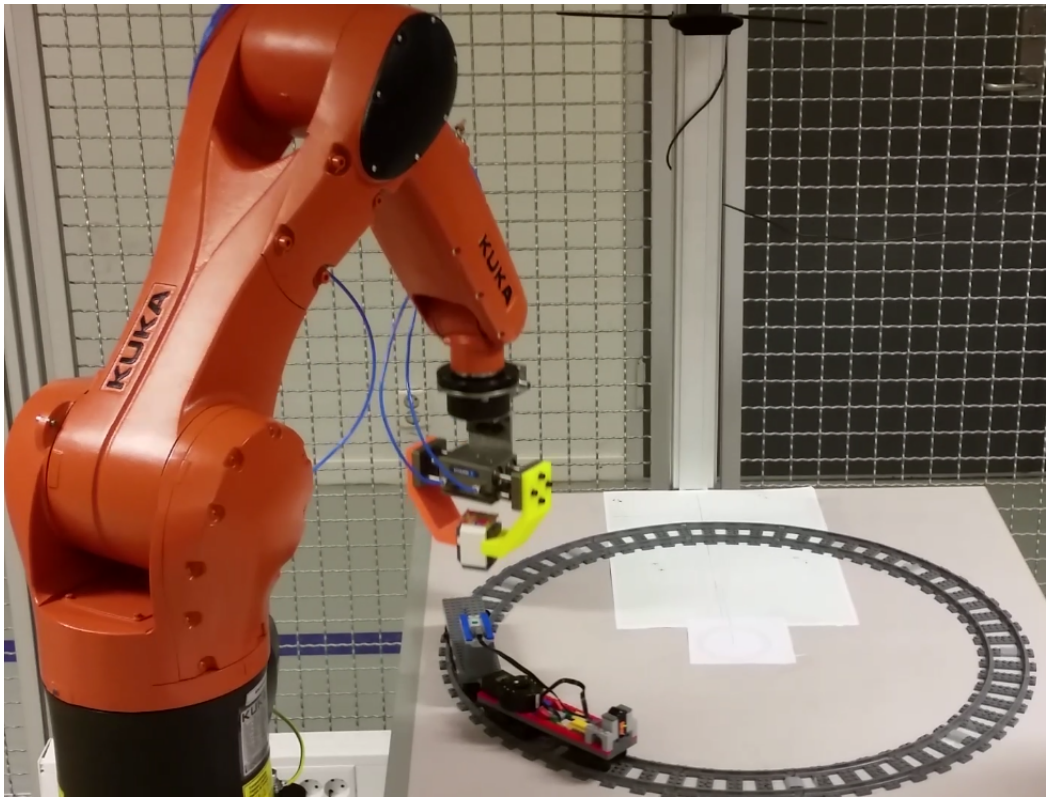


Figure 5.26: The KUKA Agilus grasping an object in motion.

5.2.5 Motion Planning

To move the manipulator to a requested position, the inverse kinematic from the specific point to the robot base have to be estimated. The manipulator may reach a position through many different paths and determining a perfect trajectory for a specific situation is challenging. The trajectory have to be a Cartesian path decided according to constraints related to joint limitations, path, operating time, speed and step size. To compute this perfect trajectory, a ROS integrated package called *MoveIt!* was used. This package communicates with the manipulator through ROS topics and actions [16]. Actions are represented as left right arrows in Figure 5.27. The communication between the camera, the computer, Movit! and the manipulator is illustrated in Figure 5.27. Movit! listens to the "Joint states" and the "TF" topic generated by the robot and determine joint locations according to this. The "Joint states" topic contains information about the joints related to for example angles, and the "TF" topic relates to the forward kinematic of the joints related to the robot base. A node called "Agilus 2", which represents the KUKA Agilus manipulator, continuously updates these topics. This node communicates with the robot trough Robot Sensor Interface (RSI), which is a program that passes information between the KR C4 controller and the computer as a XML file over an Ethernet connection. The KR C4 controller administrates signals between the robot, PLC, safety systems and the smartPAD. KR C4 controller, KUKA Agilus KR 6 R900 sixx and the smartPAD are represented in Figure 5.28.

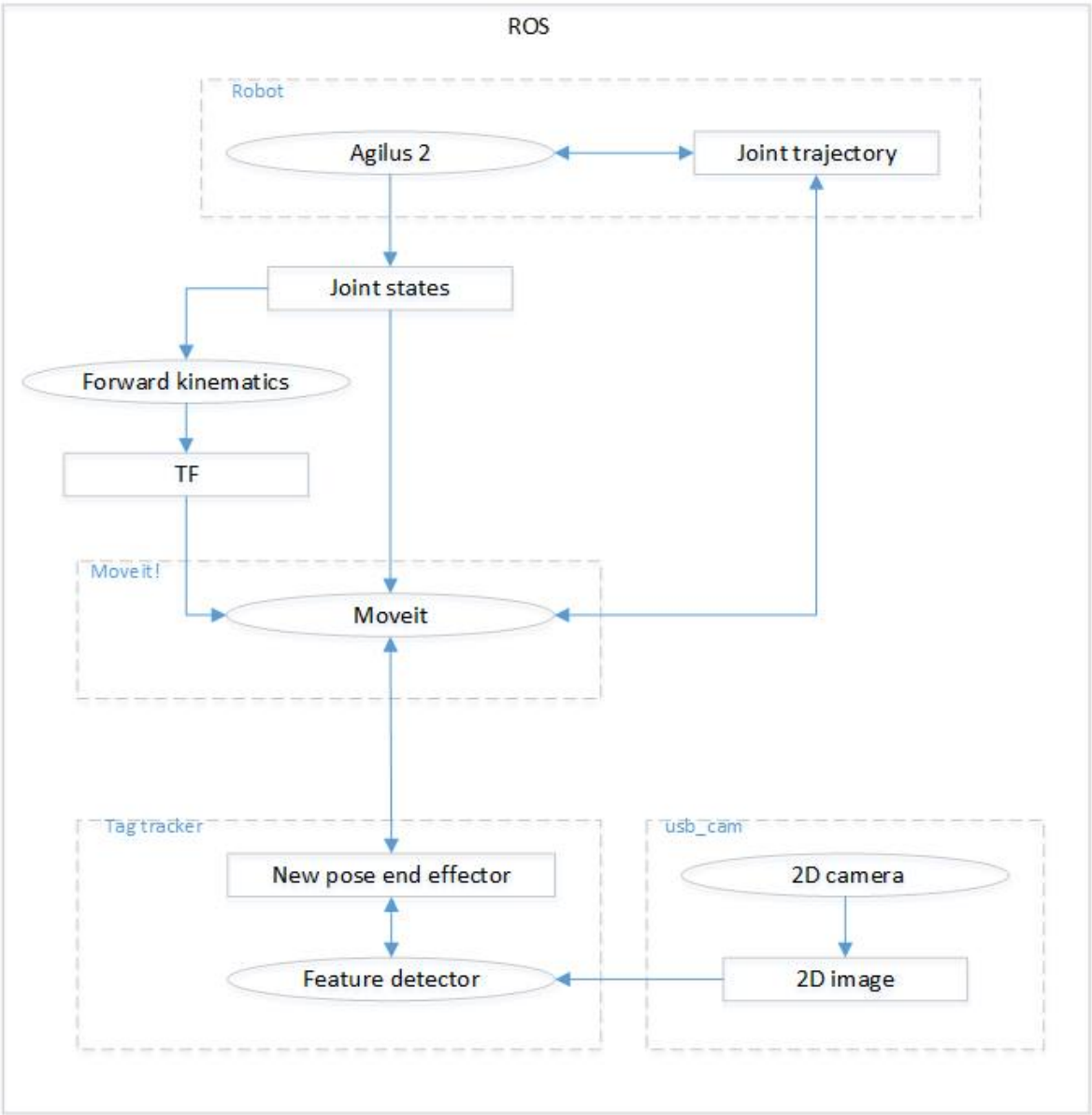


Figure 5.27: Node structure in ROS, involving 2D camera, an implementation of a feature detector, Movit! and a KUKA Agilus KR 6 R900 sixx robot.

When the "Movit" node receives a request for a new position, it generates a desired trajectory with respect to the KUKA Agilus predefined kinematic constraints [16]. These constraints are related to position, orientations, joints and user-specified constraints.

Figure 5.28: KR C4 controller, KUKA Agilus KR 6 R900 sixx and the smartPAD.

This trajectory will move the manipulator to the desired location. The result delivered by "Movit" is a trajectory and not just a path. Movit! will use the desired maximum velocities to generate a trajectory that obeys velocity constraints at the joint level. To obtain the best trajectory according to the constraints, MoveIt! uses an open-source motion planning library OMPL (Open Motion Planning Library) as primary/default set of planners. When the final trajectory is determined, the inverse kinematics to move the joints must be estimated. Movit! uses a integrated default inverse kinematics plugin that is configured using a numerical jacobian-based solver. If any object is mounted on the robot, for example a gripper, this can be configured in Movit! and the trajectory will avoid the obstruction by using a *collision checker* supported in ROS. The trajectory may finally be generated, and the motion can be executed on the manipulator. The trajectory will appear as a linear trajectory, and can also be simulated in a ROS integrated simulation tool called *rviz*, see Figure 5.29.

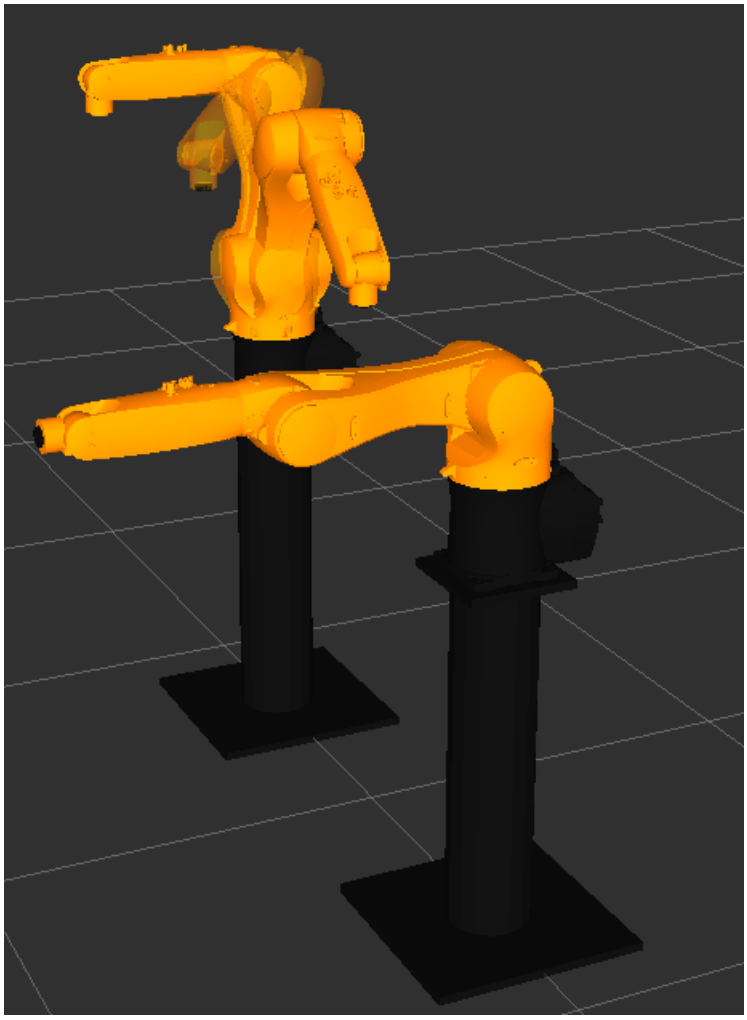


Figure 5.29: rviz simulation of a trajectory defined with MoveIt!.

5.2.6 Grasping Object

Last step of the experiment was grasping the object at the estimated time. The gripper was implemented as an object in the same implementation as the feature detection, and then a part of the same node "Feature detector", see Figure 5.27. This object communicated with a PLC and was able to open and close the gripper if requested, see gripper in Figure 5.30. It appeared to be a delay in the communication with the gripper, and this was compensated by advancing the command with 1.4 seconds.

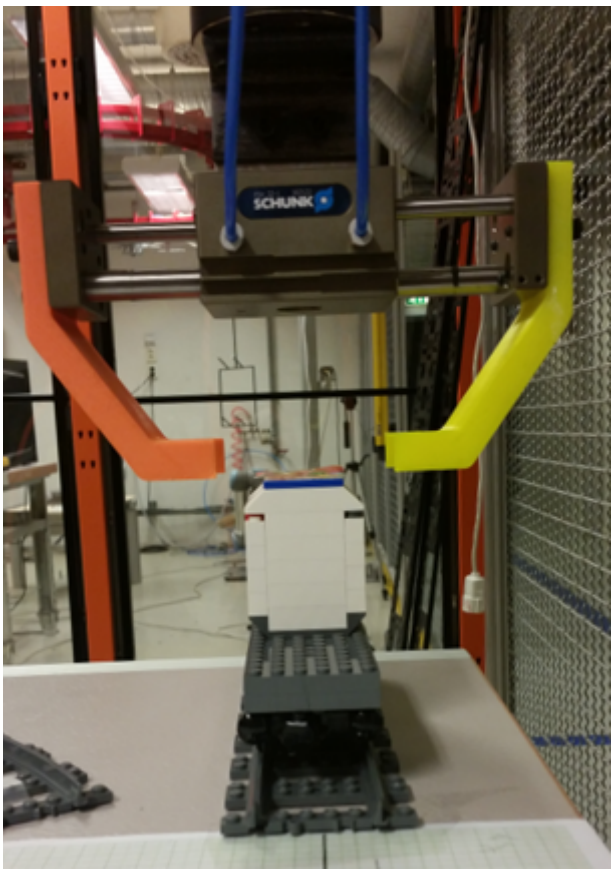


Figure 5.30: Image of the gripper.

Chapter 6

Results

6.1 Result from Experiment 1: Comparing SIFT and SURF

6.1.1 Results from Identifying Robustness

To identify the robustness of each method, ten tags were tracked and the results were compared. The goal was to gather reliable data and evaluate the methods on an equal basis. The results shows variation had a wide variety and some of the tags turned out to be almost untraceable. Duration of matching intervals and the number of maximum matches are represented in Table 6.1.

Identifying robustness				
Tag [-]	Method [SIFT/SURF]	Maximum matches [#]	Best matching interval [-]	Mismatch [Yes/No]
1	SIFT	17	7.448	Yes
	SURF	8	7.1667	No
2	SIFT	4	2.4125	No
	SURF	0	0	No
3	SIFT	368	7.1648	No
	SURF	0	0	Yes
4	SIFT	1	0	Yes
	SURF	0	0	Yes
5	SIFT	1	0	Yes
	SURF	0	0	Yes
6	SIFT	30	9.4086	Yes
	SURF	2	0.101	Yes
7	SIFT	0	0	Yes
	SURF	0	0	No
8	SIFT	78	11.3825	No
	SURF	12	5.6009	No
9	SIFT	16	7.3714	No
	SURF	15	7.693	No
10	SIFT	7	6.6165	Yes
	SURF	1	6.3326	No

Table 6.1: Results from experiment of identifying robustness.

According to the table, SIFT appears to be more robust than SURF. Both method tracked tags

for long intervals unaffected by rotation and scale, but SIFT got a better result than SURF in total. SIFT had usually the longest tracking intervals and the highest number of matches, but in some SURF appeared to get less mismatches than SIFT. This could be a result of the strict value of the Hessian threshold (minHessian=5000) used to detect robust features. The total experience of this test was that SIFT had the most reliable results, but SURF is a good replacement if the input values were adjusted according to the specific object in motion. Masks could also be used to restrict the tracking area if results appears to be affected of mismatches.

6.1.2 Results from Computation Time

In operations where computing power is limited, fast feature detection is necessary. SIFT and SURF were implemented to detect an equal amount of features in two different tags. The results were compared with respect to time spent on the three steps; detecting, description and matching. The time was logged and plotted in graphs, see Chapter 5.1.2, and a percentage values of the time used in SURF compared to SIFT is represented in Table 6.2. These percentage values were based on the average values from four graphs, the plot in Figure 6.1 and three additional graphs in Appendix A.2.

Computation Time						
	Tag [#]	Features [#]	Detecting [s]	Descriptors [s]	Match [s]	Total [s]
SIFT	1	440.075	0.112	0.114	0.007	0.233
SURF	1	439.857	0.076	0.113	0.003	0.192
Difference [%]		99.951	67.956	99.418	43.038	82.492
SIFT	1	698.052	0.111	0.123	0.009	0.243
SURF	1	698.667	0.087	0.166	0.005	0.258
Difference [%]		100.088	78.420	135.474	57.130	106.409
SIFT	9	166.157	0.113	0.077	0.002	0.192
SURF	9	165.616	0.062	0.054	0.001	0.117
Difference [%]		99.675	54.942	69.503	76.612	61.006
SIFT	9	916.041	0.103	0.110	0.006	0.219
SURF	9	915.885	0.081	0.164	0.004	0.250
Difference [%]		99.983	78.623	149.837	69.995	113.939

Table 6.2: Results from experiment of computation time, where the percentage value of SURF compared to SIFT is presented.

The result was not as expected. SIFT and SURF was tested according to a vary amount of features, 440 and 689 in tag 1, and 166 and 916 in tag 9, and SURF was expected to have an approximately constant and low computing time. SIFT was implemented with the default values and a limitation of features. The predefined amount of features was then ensured in SURF by adjusting the detection with a suitable Hessian threshold. SURF obtained best result using a strict minHessian=5000, detecting 166 features, and worst results using the recommended minHessian=400 [5], detecting 440 features. The first detection corresponds to 61% of the time spent in SIFT, while the second situation corresponds to 114%. In the two other cases, the percentages correspondence was 82.5% and 106.5%. According to [2], this is quite unexpected results. SURF is supposed to be up to three time faster than SIFT [2], and these results did not show any significantly faster detection using SURF. The table (Table 6.2) shows that the descriptor extraction is the most affected due to variation in amount of features, and the result was especially unusual when 698 and 916 features were detected. In these cases,

SURF used significantly more time than SIFT to create descriptors that are half the dimension of a SIFT-descriptor.

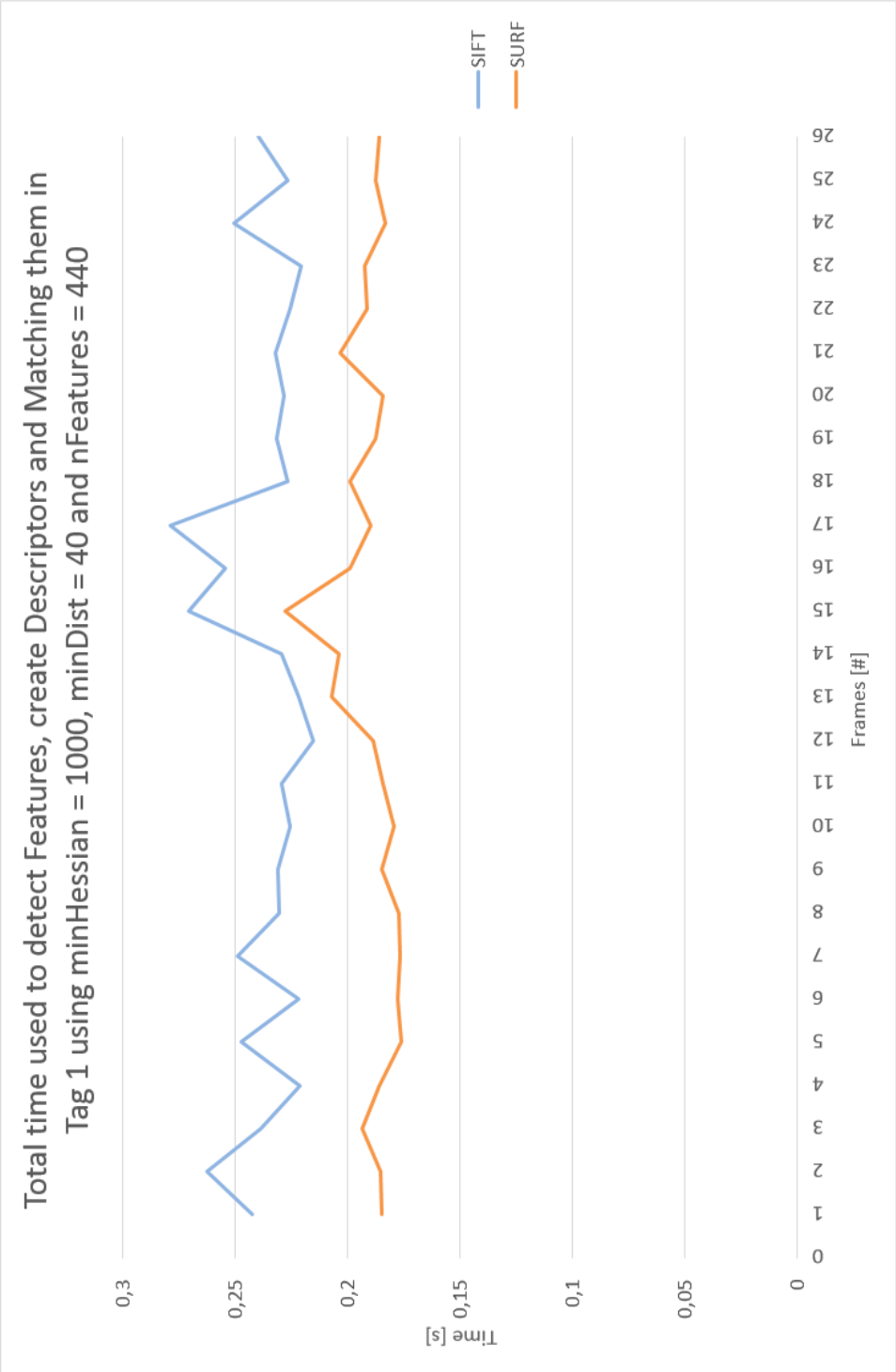


Figure 6.1: Time of detecting features and create descriptor using tag 1.

6.1.3 Improvement of Computation Time

To improve the computation, the implementation of the feature detector in SURF was combined with the implementation of the descriptor extractor in SIFT. The implementation was executed on the same tags as in the previous Chapter 6.1.2 and compared with an implementation of the SIFT feature detector and the SIFT descriptor extractor. 165, 185 and 230 features were detected in this test. The computation time of detecting 165 features is represented as a graph in Figure 6.2. The corresponding graphs from the three other implementations are illustrated in Appendix B.2. The finally results are compared and represented in Table 6.3. The implementation of a combination of the SURF feature detector and the SIFT descriptor extractor appears to be 50% faster then a implementation of a SIFT detector together with a SIFT descriptor extractor. This combination could be a solution for operations where the computation power is limited and the computations must be performed close to real time.

Computation Time						
	Tag [#]	Features [#]	Detecting [s]	Descriptors [s]	Match [s]	Total [s]
SIFT	1	165.146	0.120	0.086	0.003	0.210
SURF/SIFT	1	165.114	0.063	0.032	0.001	0.097
Difference [%]		99.981	52.581	37.266	47.684	46.205
SIFT	1	230.179	0.117	0.096	0.004	0.217
SURF/SIFT	1	230.055	0.065	0.034	0.001	0.100
Difference [%]		99.946	55.686	35.364	28.578	46.197
SIFT	9	185.162	0.113	0.080	0.002	0.195
SURF/SIFT	9	184.819	0.065	0.055	0.001	0.121
Difference [%]		99.815	56.913	68.834	78.603	61.969
SIFT	9	230.146	0.108	0.090	0.001	0.199
SURF/SIFT	9	229.875	0.065	0.034	0.001	0.100
Difference [%]		99.882	59.913	37.434	110.356	50.027

Table 6.3: Results related to an implementation of the feature detector of SURF in combination with the descriptor extractor of SIFT compared with a standard implementation of SIFT, which contains the feature detector and the descriptor extractor of SIFT.

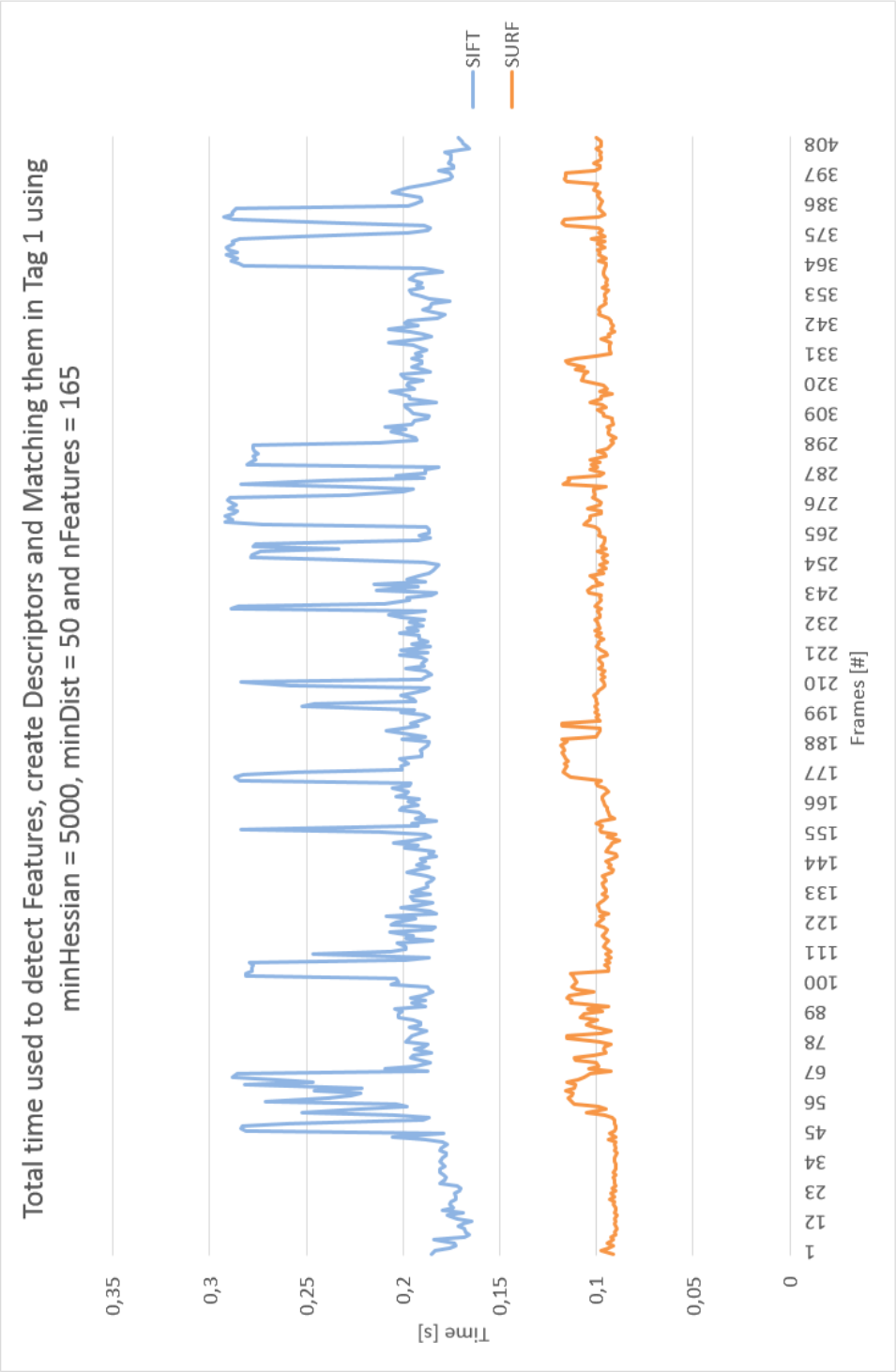


Figure 6.2: Time of detecting features and extracting descriptors using tag 1.

6.2 Results from Experiment 2: Grasping an object in motion with a KUKA Agilus Manipulator

Both SIFT and SURF were useful to estimate position in Cartesian world coordinates with the 2-dimensional camera. SIFT was the best solution when using strict limitations, but since detection areas could be restricted with masks, SURF was a good solution according to computation time. Both methods appeared to perform good tracking of the tag. The practical experiment represents how SIFT and SURF could be used to track objects in motion using the visual characteristics. For example, colors, patterns, design or tags.

6.2.1 Video

A video of the experiment has been produced and placed as Digital Appendix D. It is a movie of an implementation of SIFT, while tracking an object in motion. The angle velocity and a grasping position with a corresponding time are estimated. The grasping position is transmitted to the KUKA Agilus Manipulator and the manipulator executes the requested trajectory. When the KUKA Agilus Manipulator approaches the given position, the gripper becomes activated and the object is grasped at the estimated time. Finally, the KUKA Agilus manipulator delivers the object at a predefined position.

Chapter 7

Summary and Recommendations for Further Work

7.1 Conclution

The main objective of this project was to detect an object in motion using computer vision. The two different methods SIFT and SURF, were compared and implemented in a practical experiment. To perform an overall evaluation of the methods, three experiments used to compare, matches, matching intervals and computation time.

The results were as expected. SIFT was more robust then SURF, and SURF was mostly faster than SIFT. Both methods show weaknesses related to illumination changes, especially in the left area of the camera view, which was close to a light source. This is visible in for example Figure 5.7, where the light reflection appears as a white area on the left side of the train track. It was also performed an experiment combining the SURF feature detector and the SIFT descriptor extractor. The combination proved to be more efficient and a standard implementation, and could be a replacement when computation power are limited.

The practical experiment shows how a position for a manipulator could be estimated according to pixel values. The implementation was successful, and a movie was produced to show the result of the implementation (6.2.1). The methods delivers good performance and show high potential for improving the industry of both mass production and subsea technology. Both methods appeared to be very robust according changes in scale and rotation, especially when the object was stationary. When the tag was at rest in the start position, several tests show high number of matches, and a marked decrease occurred when the tag started to move. See graphs illustrated in Chapter 5.1.1 and Appendix A.1.

7.2 Discussion

SIFT resulted in increased computation time, which in some cases caused a trajectory abortion. The problem was related to the ROS node "Agilus 2" (Figure 5.27), which expected continues information from the controller about trajectory goals. In some cases, SIFT caused an interruption for longer than a predefined limitation of time, and the current robot action was aborted by the node. This caused an incomplete trajectory, and the experiment had to be reinitialized.

This repeated abortion made some limitations for the experiment, and simplifications were necessary to completing the project. When the manipulator was requested to do a rotation of a joint, the trajectory estimation caused an interruption in the connection with the controller, and the trajectory was uncompleted. The interruption happened several times, and the final suggestion was to avoid all rotations. An appropriate position for grasping of the object was defined, and the manipulator could navigate without any interruption. The rotation issue could also be avoided by dividing the total rotation into smaller steps. The computation power will then decrease, and the communication will stay uninterrupted.

The best solution of avoiding the connection delay was by decreasing the need of computation power. A suggestion, is to implement the combination of the SURF feature detector and the SIFT descriptor extractor discussed in Chapter 6.1.3, and save 50% computation time, according to the results in Table 6.3. If this is not saving enough power, another solution could be an additional computer to separate the computer vision form the communication with the robot controller. This will divide the required computation power between these two computers, and an interruption of the continuously connection between the computer and the KUKA controller may be avoided.

During the experiments, the LEGO train was observed to have irregular velocity. The train was highly affected by the battery capacity, and was acting differently during the day. This made some difficulties in the performing of experiment 2, since the position of the object was estimated according to the angle velocity. The experiment was therefore simplified, and successfully completed by using the lap time instead of a shorter interval. In the other tests, both SIFT and SURF was running simultaneously and the velocity was affecting both methods equally. Since the results were based on the performance of the method during each test, the influence of the velocity change was expect to be very low.

7.3 Recommendations for Further Work

In the practical experiment, the manipulator was navigated according to the movement of tag 1. A recommendation to future work is to perform the experiment 2 with a higher velocity by using tag 8. This was an easier tag to track.

Rotation of the gripper was avoided in experiment 2 due to communication interruption between the KUKA Controller and the computer. A recommendation is to implement the combination of SURF feature detector and the SIFT descriptor extractor. Probably this combination will save enough computation power to sustain the required communication.

Implement SIFT using 3D camera. During the project, a SIFT function for feature detecting in a 3 dimensional view was observed. The function is available in an open source library called *Point Cloud Library*, and is an adaption of the original algorithm. This method detects features in a point clouds instead of an images [11]. A recommendation is to implement this on the robotic lab, using an Asus 3D camera or a Kinect.

Appendix A

Additional Information for Chapter 5

A.1 Additional Graphs to Section: Expenditure of Time

Additional figures to Chapter 5.1.1. Figure A.2 represent the amount of matches using tag 4. The tag was possible to track with both methods, but many mismatches and intervals without matching occurred. Figure A.3 and Figure A.4 represents tag 6. Good results for SIFT, and small number off matches using SURF. Figure A.6 represents number of matches using tag 7. This had quite similar results as tag 5, see Chapter 5.1.1. The tag was highly affected of luminous reflectance and difficult to track. Figure A.8 represents the number of matches using tag 8. This had quite similar results as tag 9, see Chapter 5.1.1. Both methods had smooth tracking for a long distance, and many matches occurred. Figure A.10 represents the number of matches using tag 10. Tag 10 was quite easy to track.

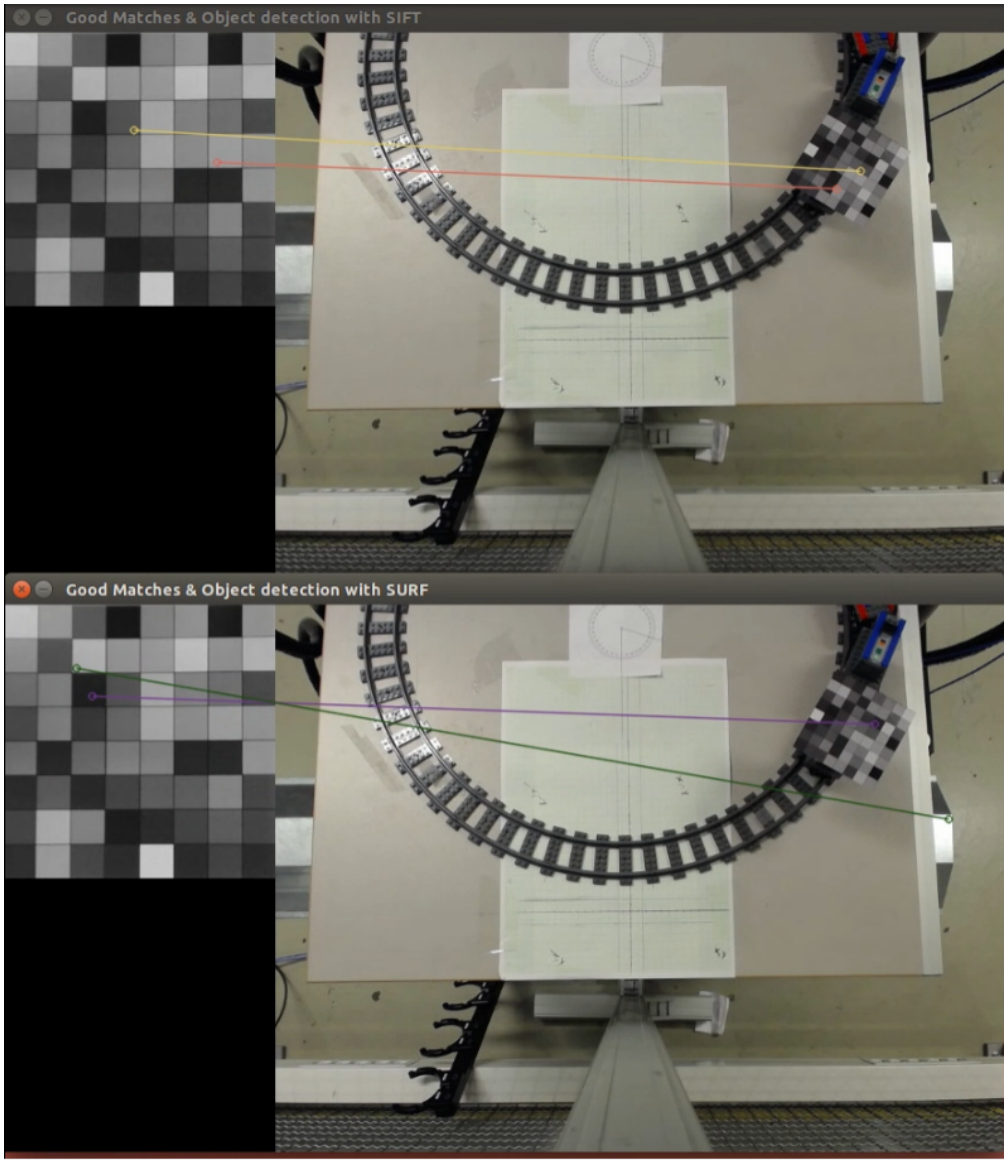


Figure A.1: Testing tag 4.

The forth tag appeared to have many intervals without matches using SIFT, and many mismatches using SURF, see FigureA.2.

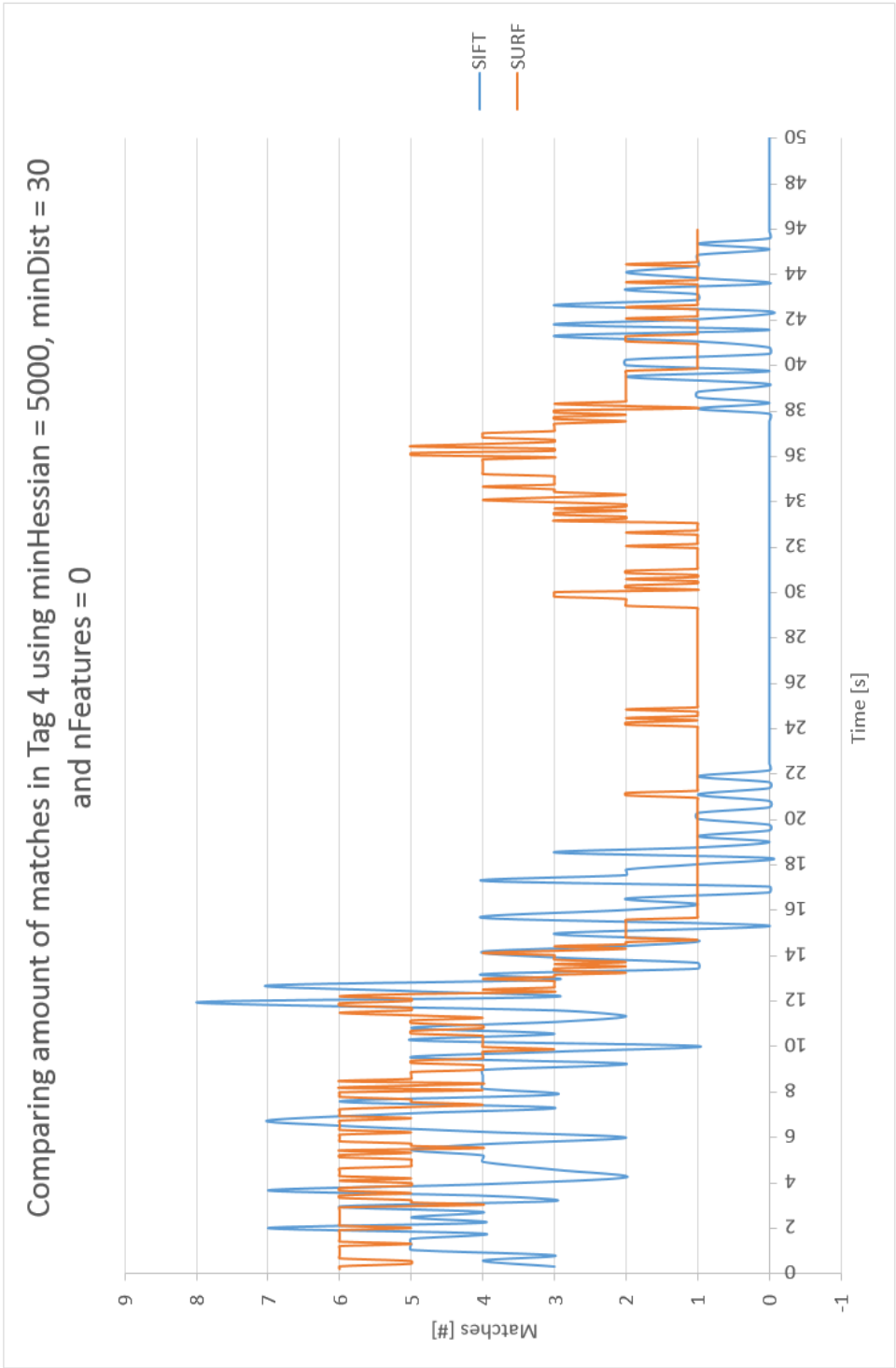


Figure A.2: Comparing matches results using tag 4. Image stream are analysed with SIFT and SURF.

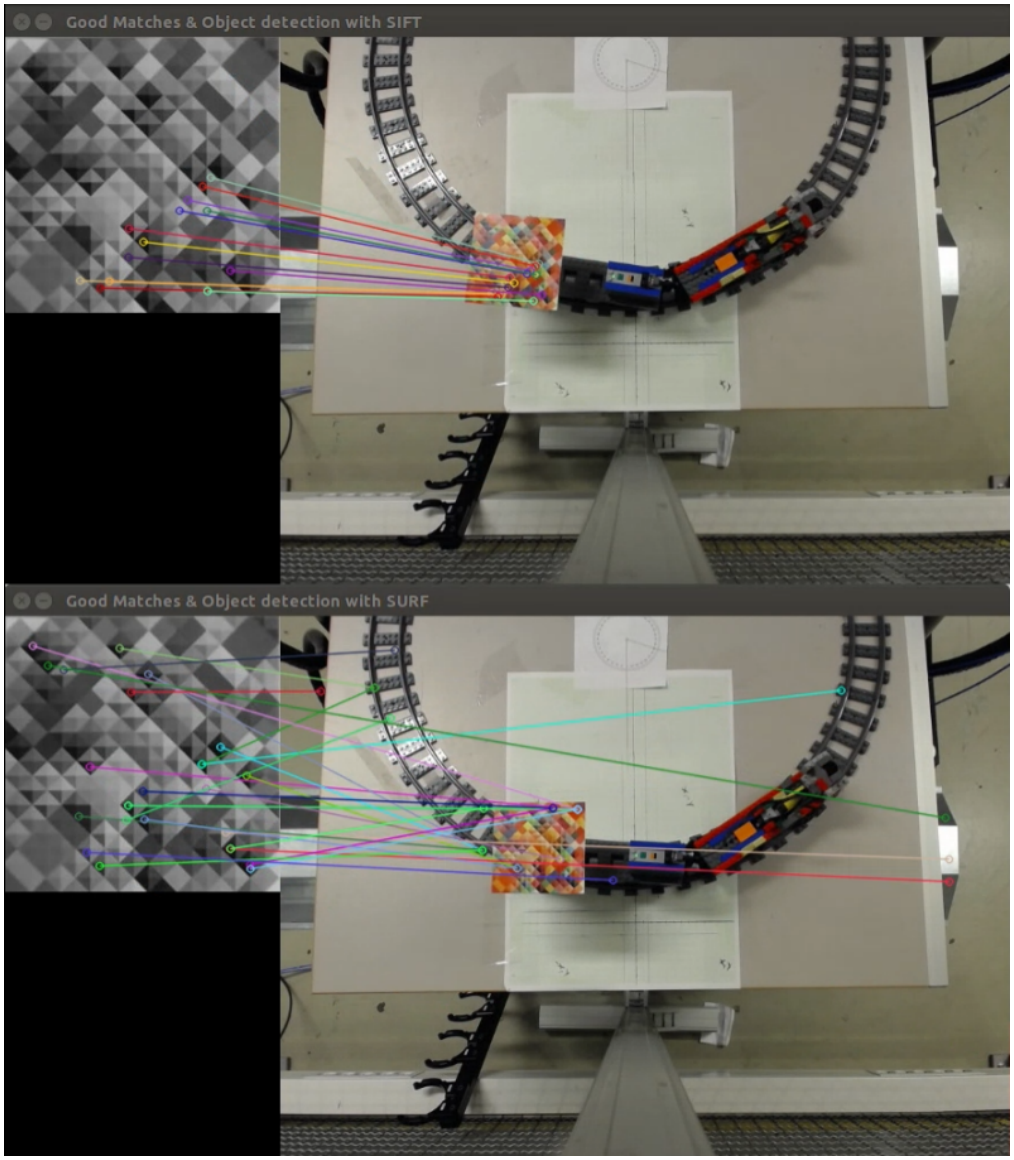


Figure A.3: Testing tag 6.

The sixth tag had very good results using SIFT. SIFT got up to 30 matches and had a stable number of matches for a long period. SURF was not that reliable, and got maximum two good matches, see Figure A.3.

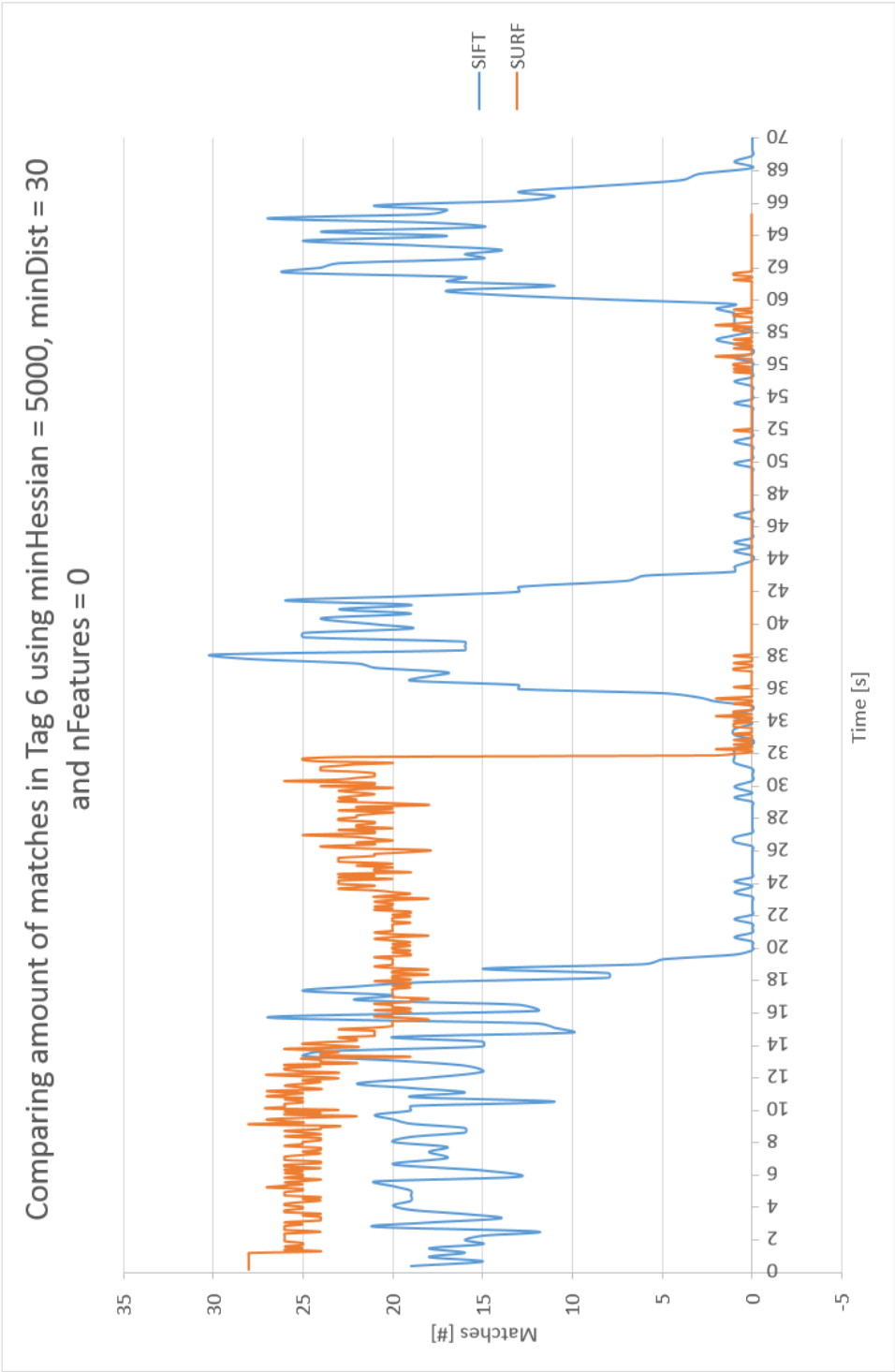


Figure A.4: Comparing matches results using tag 6. Image stream are analysed with SIFT and SURF.

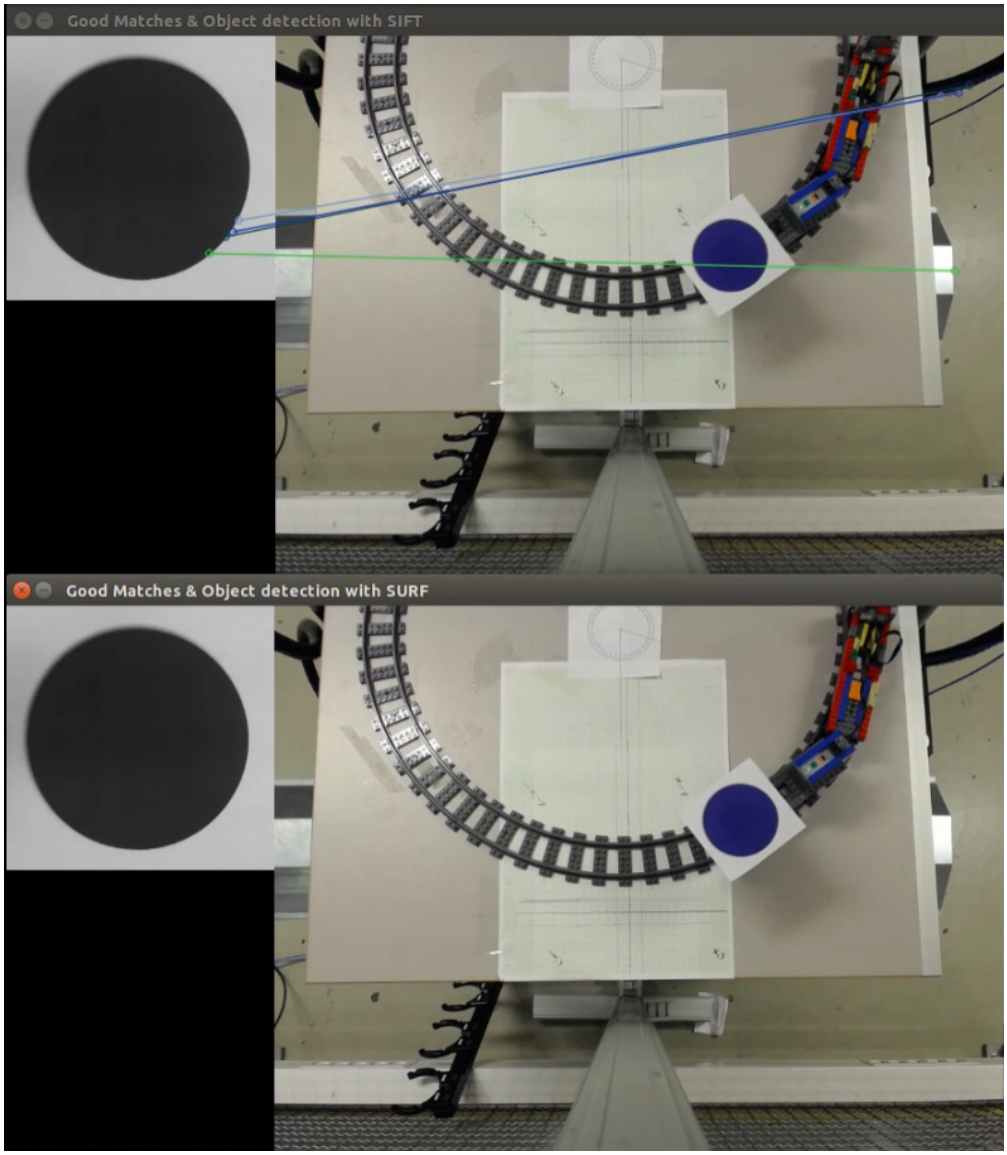


Figure A.5: Testing tag 7.

The seventh tag was almost as bad as the fifth tag. The tag was easy effected by light and many mismatches occured.

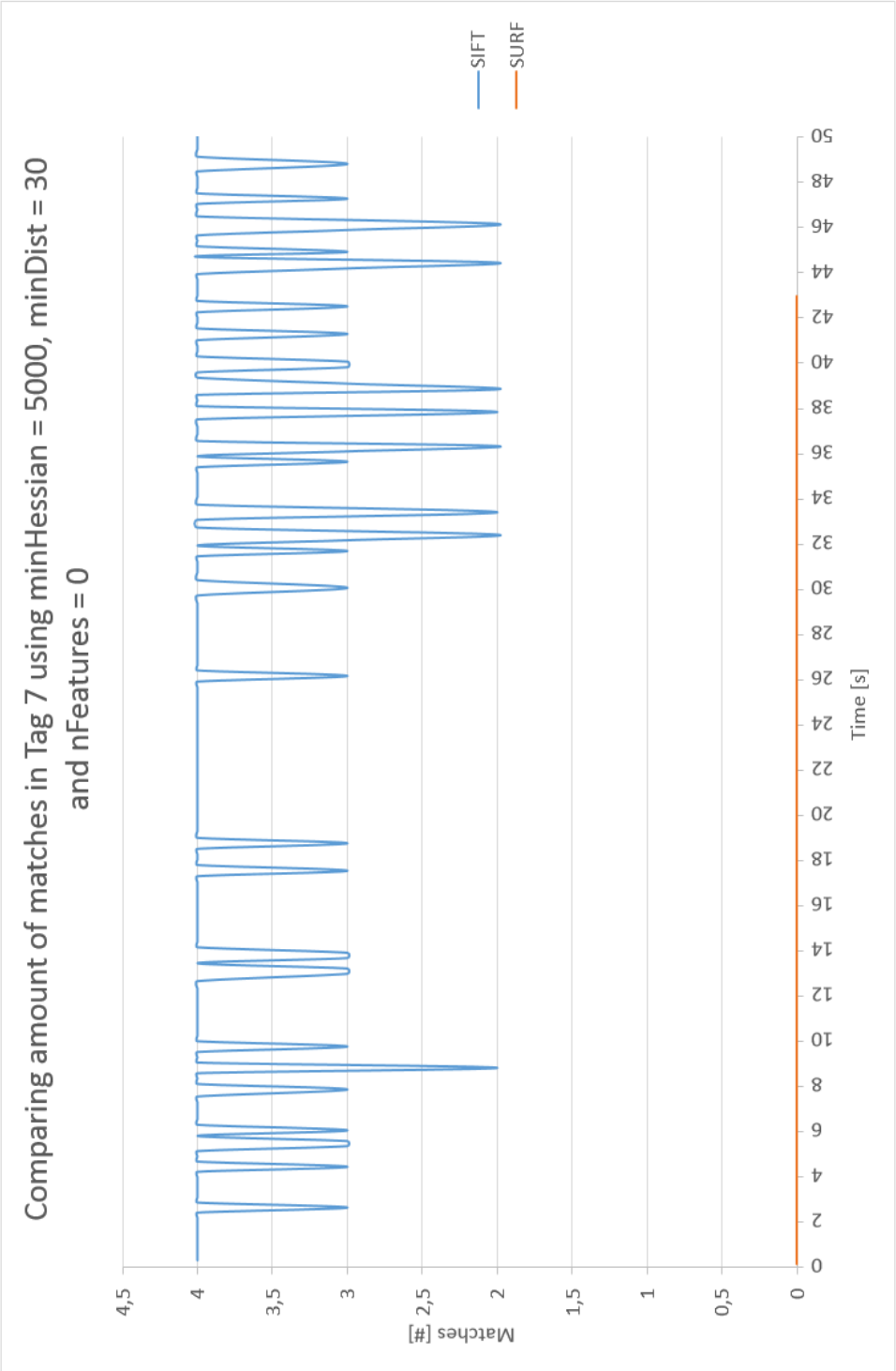


Figure A.6: Comparing matches results using tag 7. Image stream are analysed with SIFT and SURF.

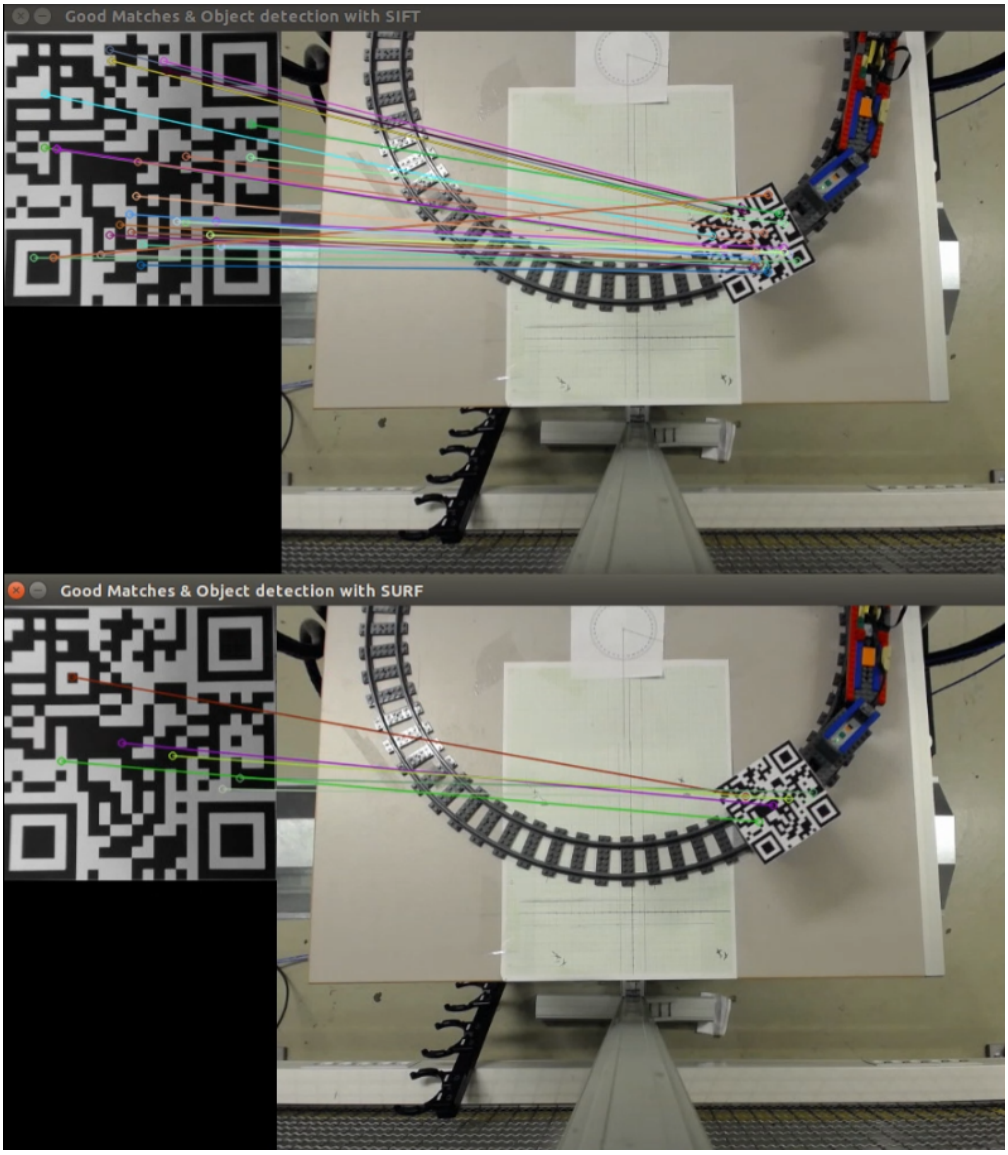


Figure A.7: Testing tag 8.

Tag eight tag appeared to be a very good tag. SIFT had up to 78 matches, SURF got 12 and none of them had mismatches, see Figure A.7. The tag was also tracked during a long interval.

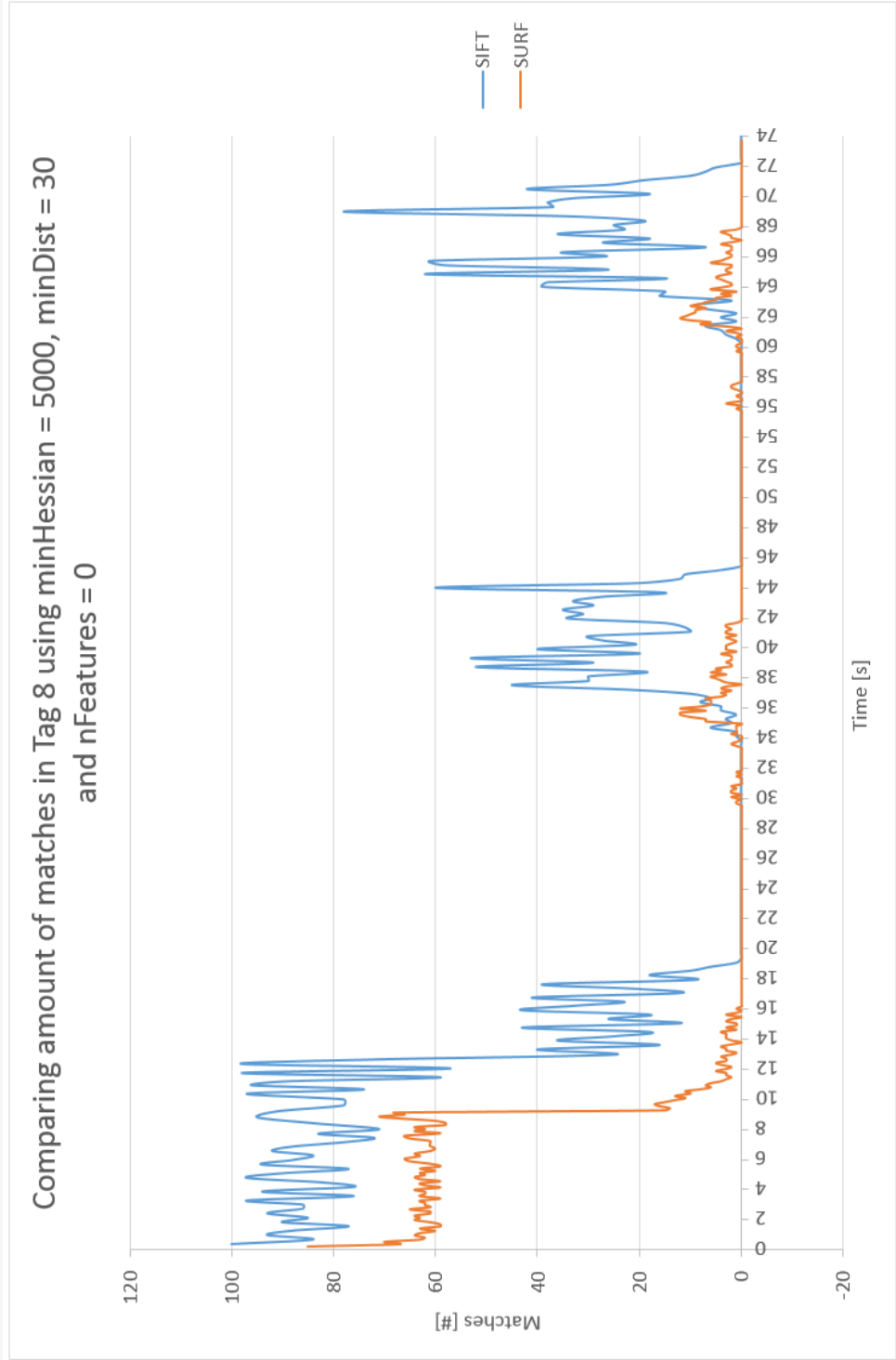


Figure A.8: Comparing matches results using tag 8. Image stream are analysed with SIFT and SURF.

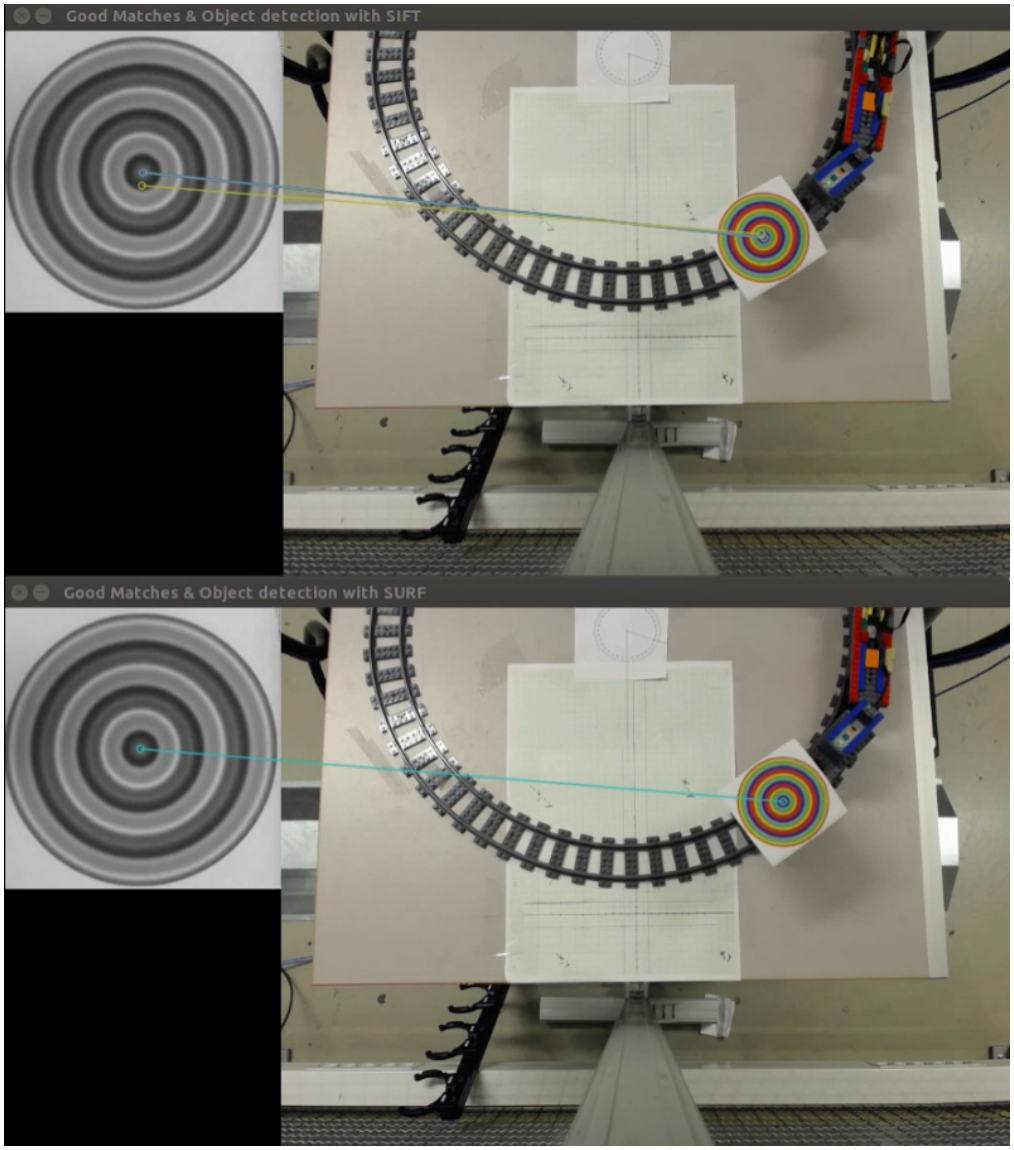


Figure A.9: Testing tag 10.

Using the tenth tag, SIFT had maximum seven matches, and SURF got maximum one.

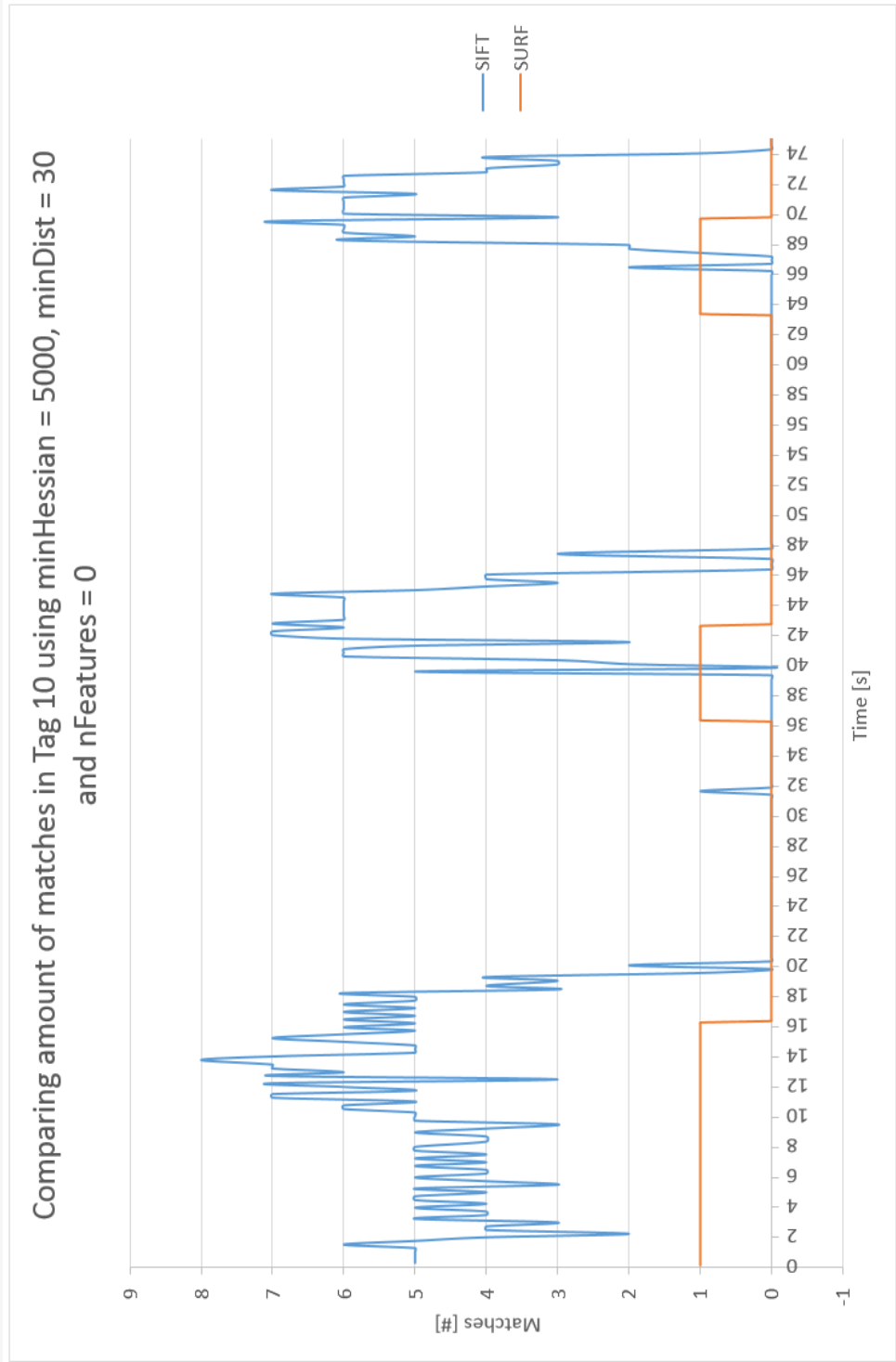


Figure A.10: Comparing matches results using tag 10. Image stream are analysed with SIFT and SURF.

A.2 Additional Graphs for Section: Computation Time

This chapter represents the additional graphs related to Chapter 5.1.2. Figure A.11 represents time used to detecting approximately 698 features in tag 1. Figure A.12 represents time used to create the descriptors, and Figure A.13 represents the time used for matching the features. Similar are represented for 166 features from tag 9 in Figure A.14,Figure A.15, Figure A.16 and for 916 features from tag 9 in Figure A.17, Figure A.18, Figure A.19.

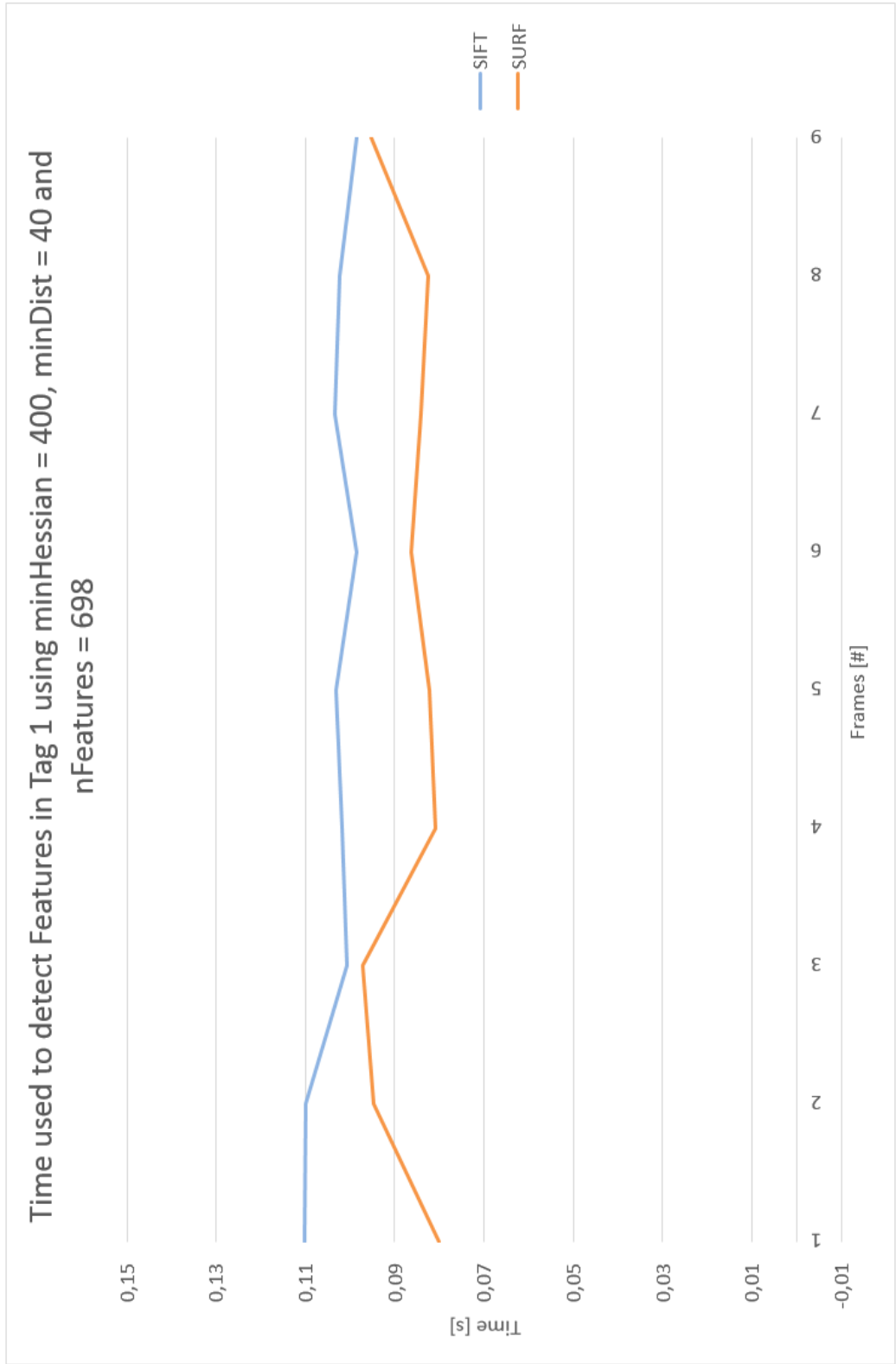


Figure A.11: Time used to detecting approximately 698 features in tag 1.

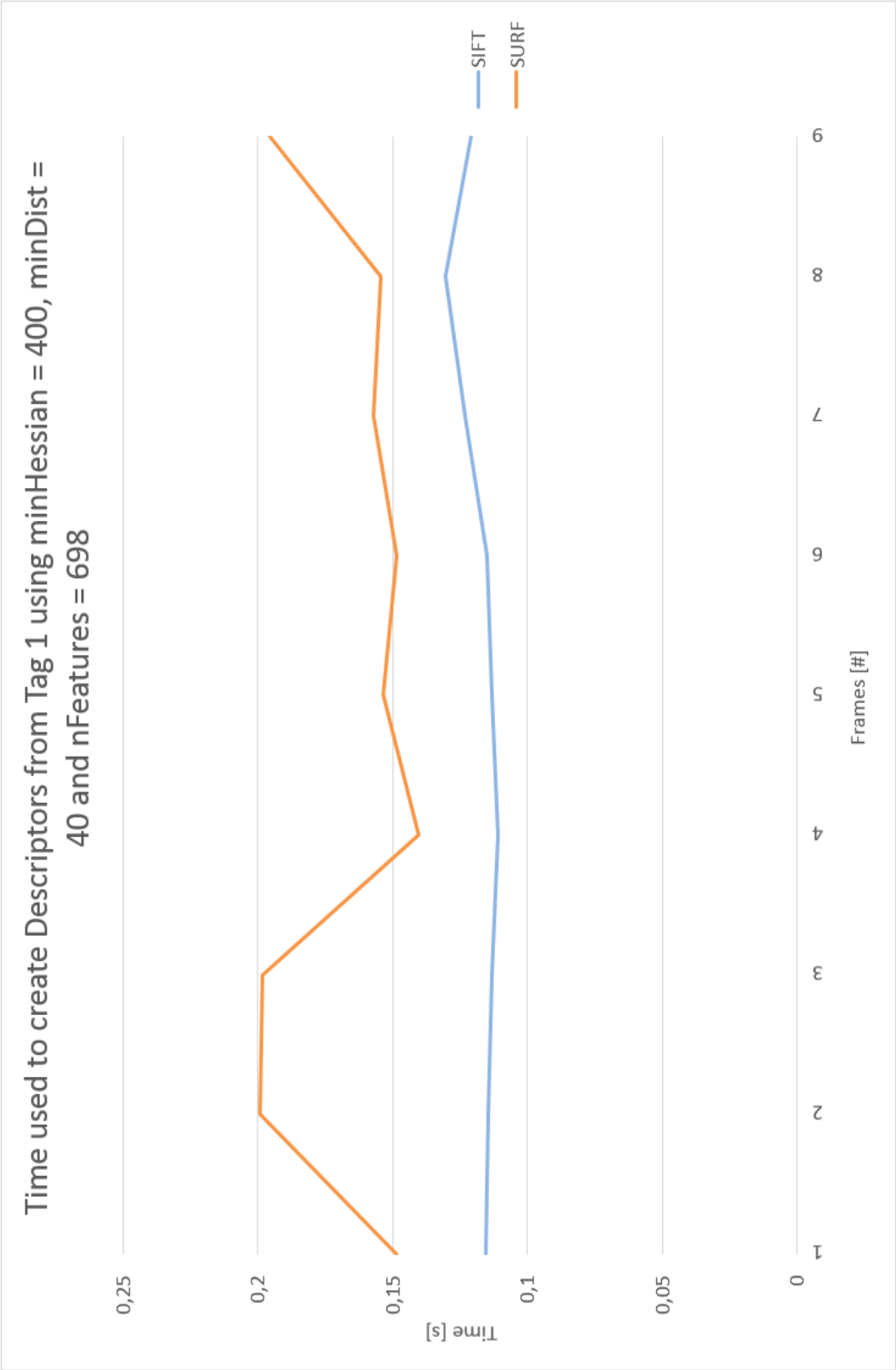


Figure A.12: Time used to extract descriptors for approximately 698 features from tag 1.

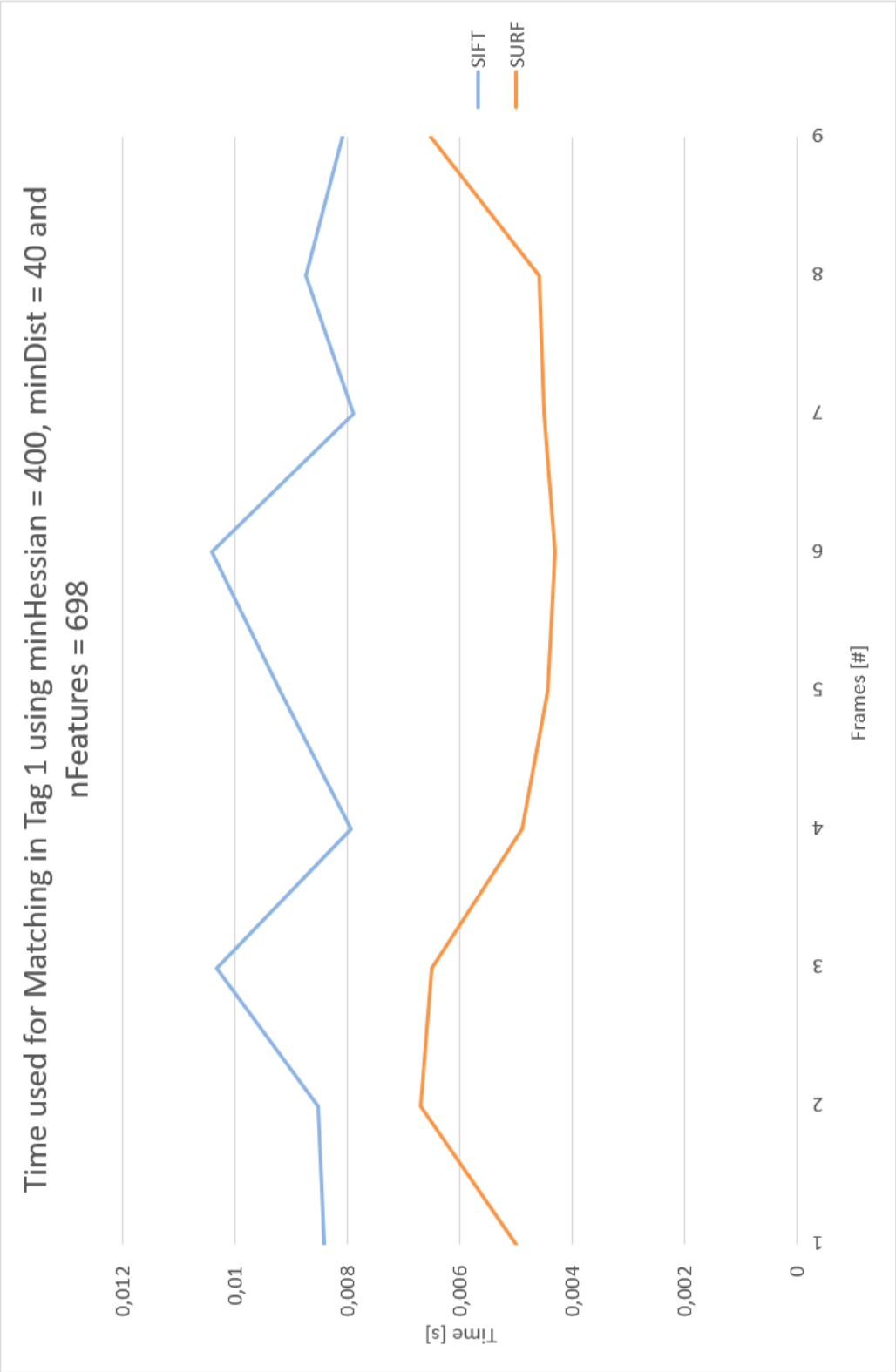


Figure A.13: Time used to match approximately 698 features in tag 1.

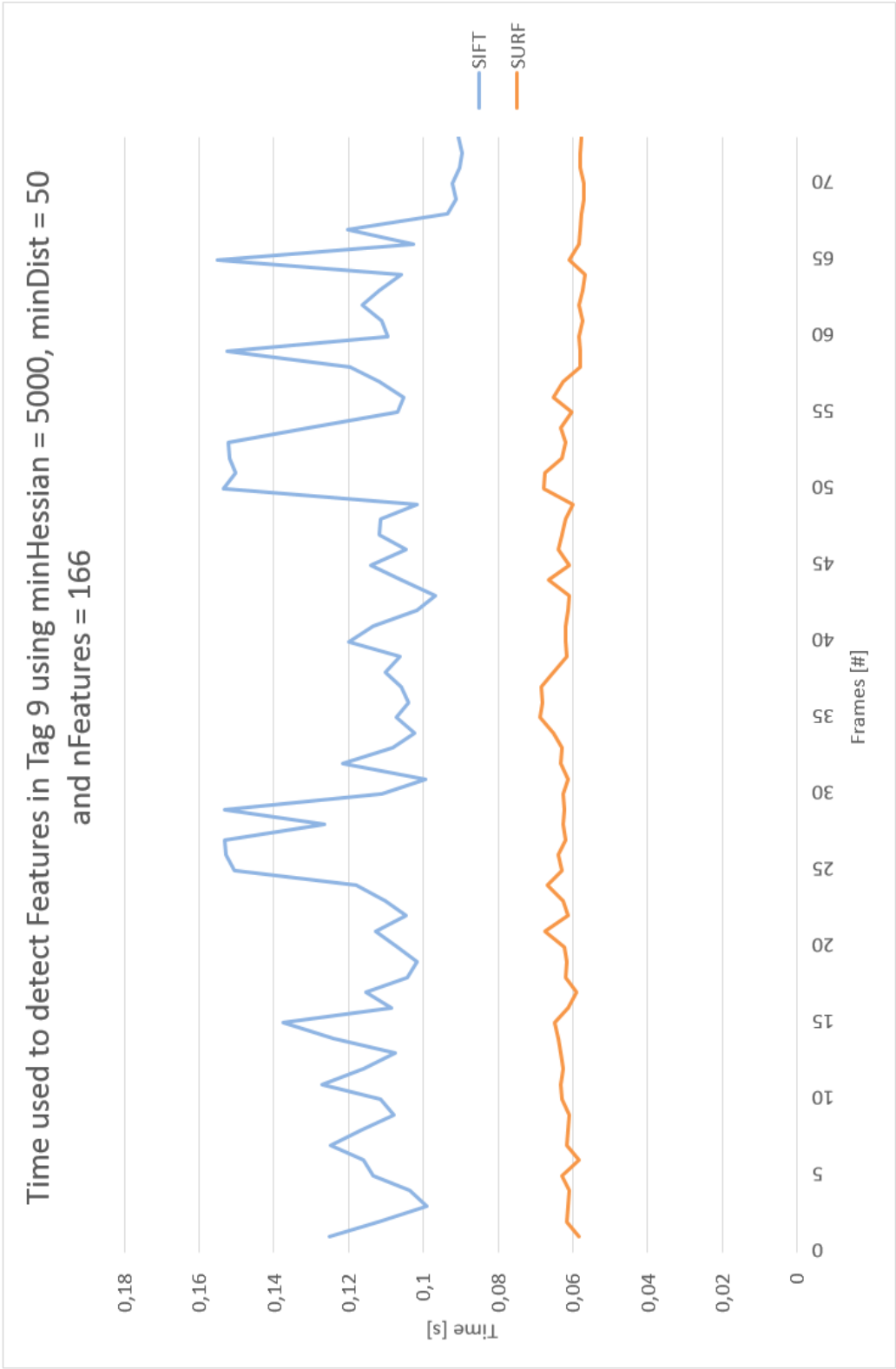


Figure A.14: Time used to detecting approximately 166 features in tag 9.

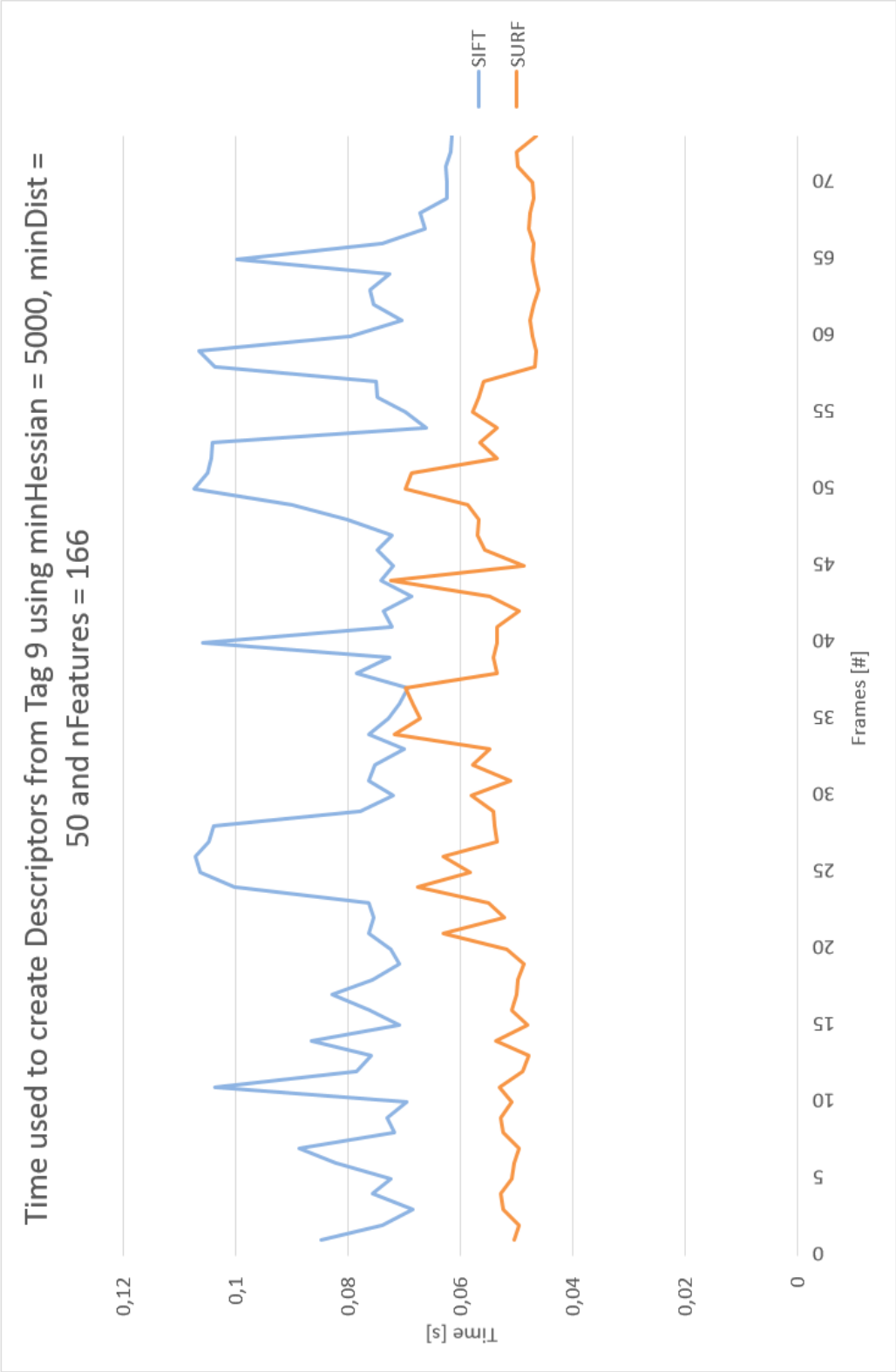


Figure A.15: Time used to extract descriptors for approximately 166 features from tag 9.

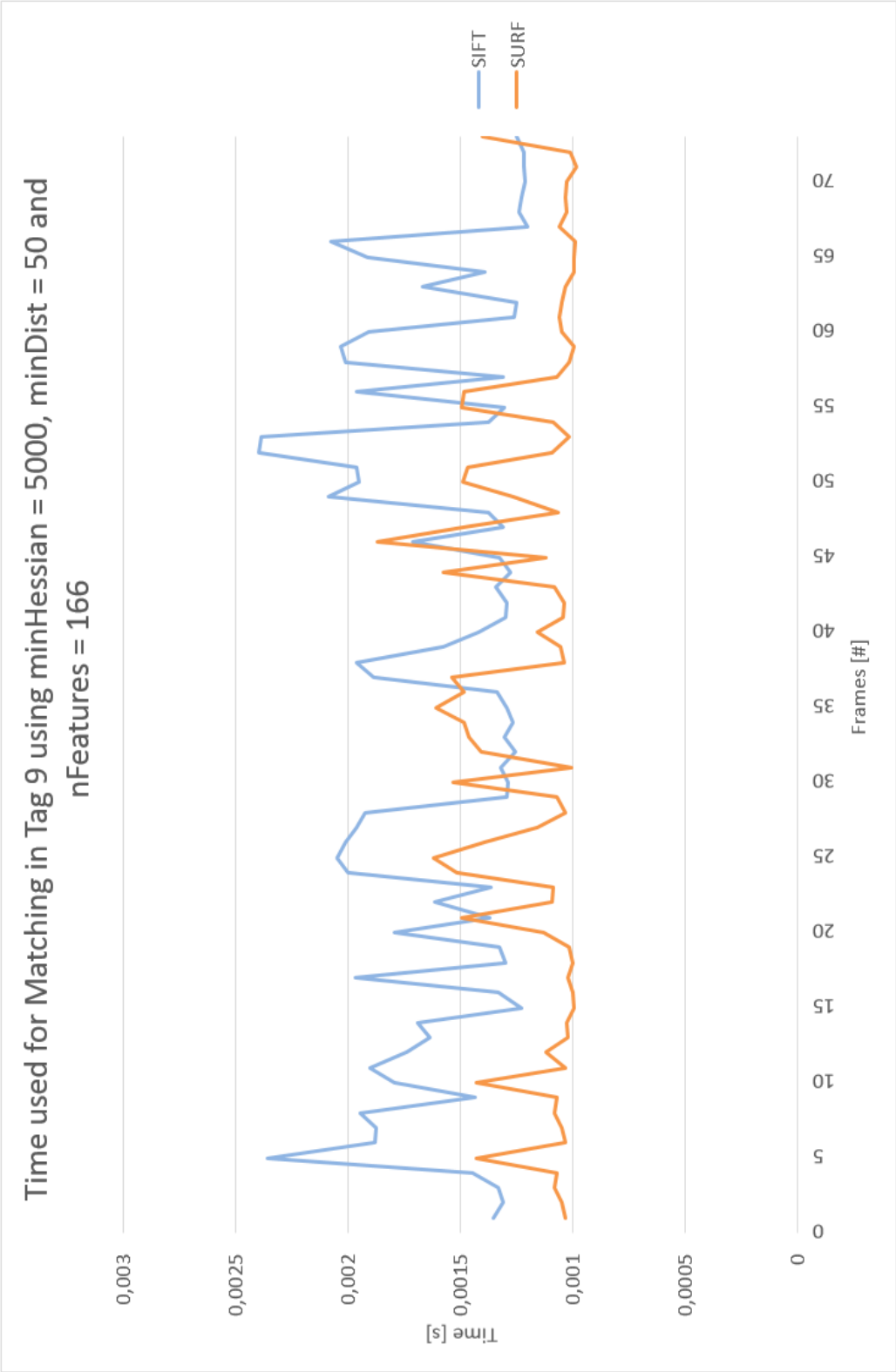


Figure A.16: Time used to match approximately 166 features in tag 9.

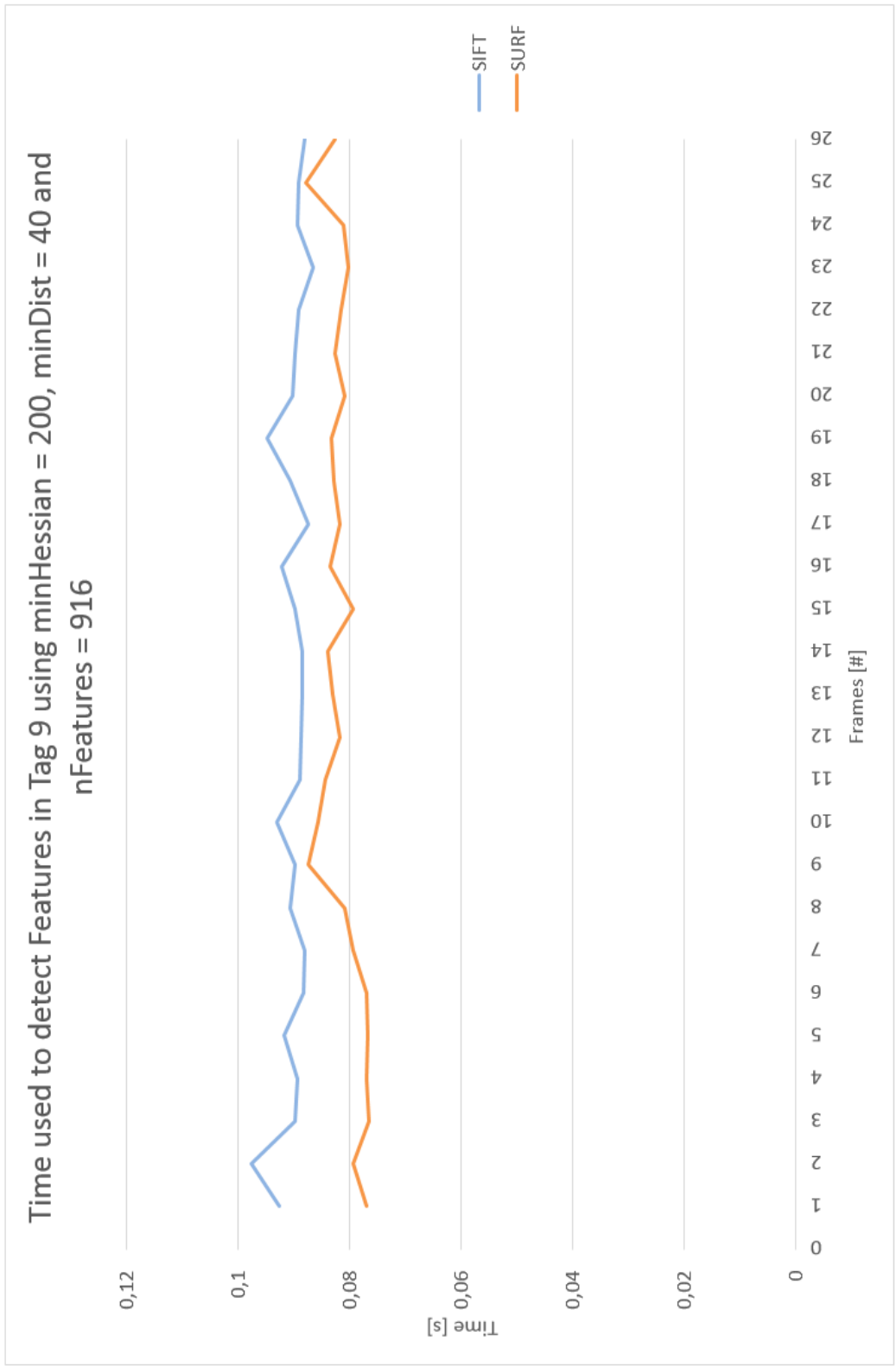


Figure A.17: Time used to detecting approximately 916 features in tag 9.

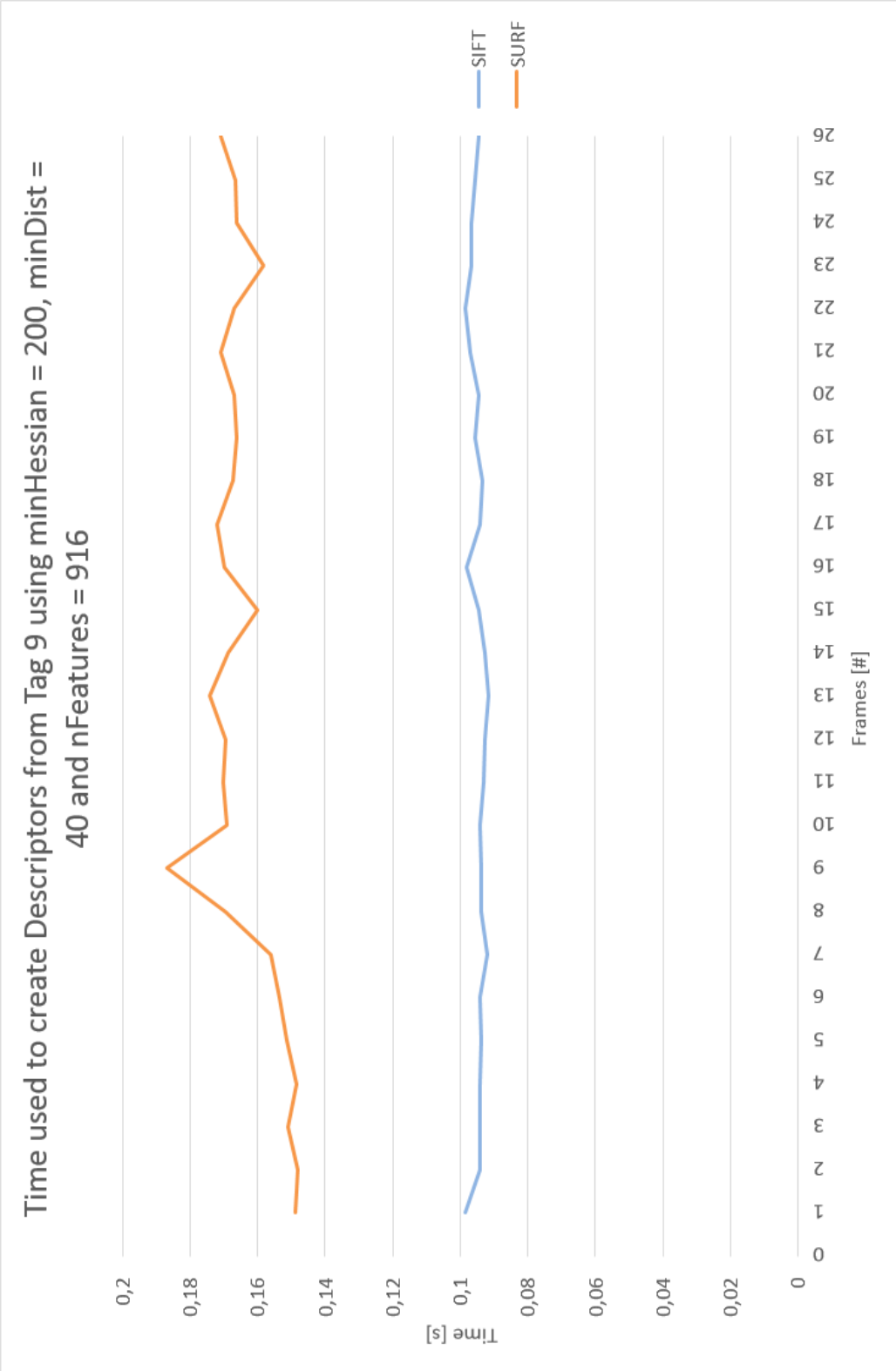


Figure A.18: Time used to extract descriptors for approximately 916 features from tag 9.

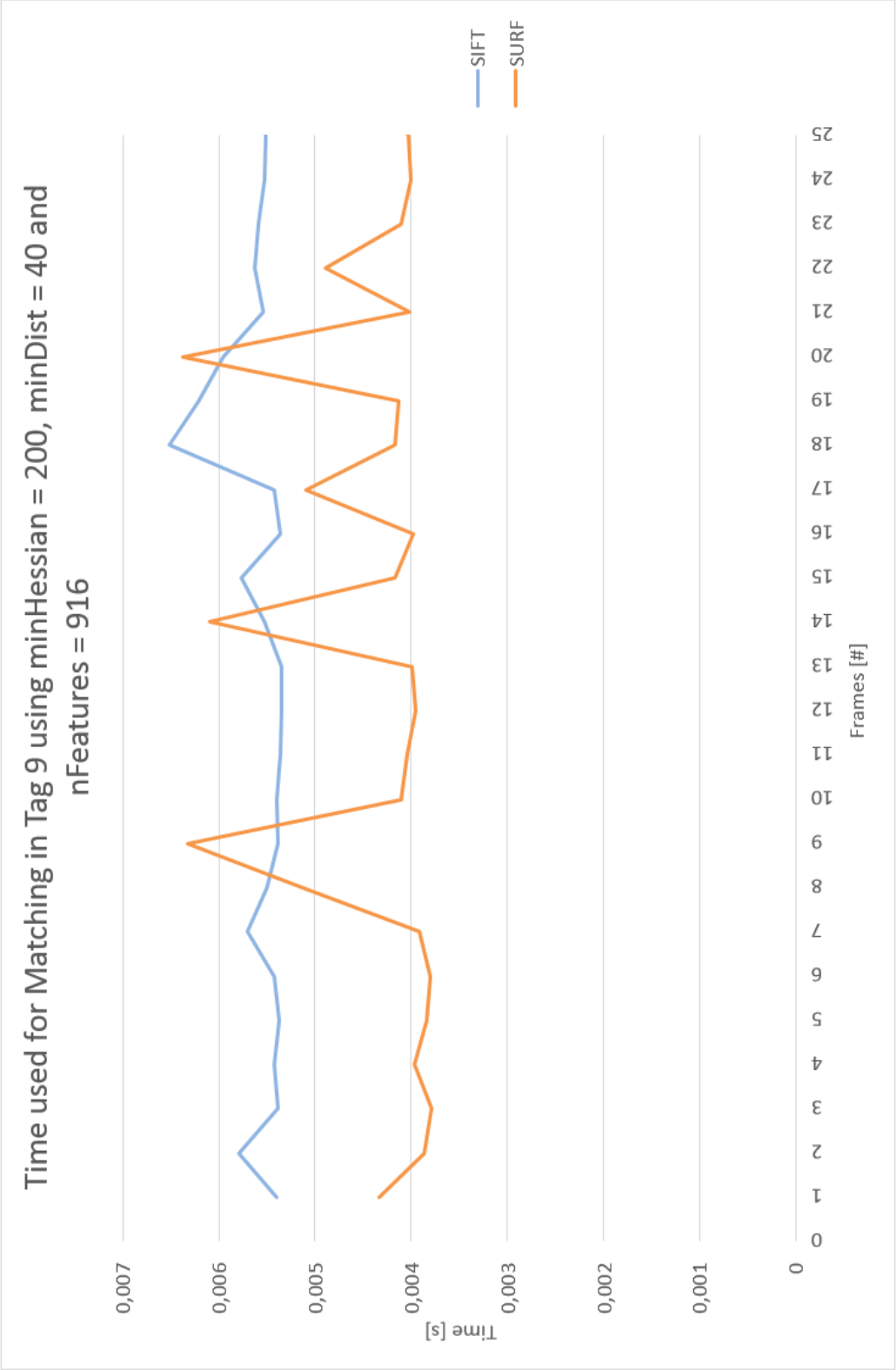


Figure A.19: Time used to match approximately 916 features in tag 9.

Appendix B

Additional Information for Chapter 6

B.1 Additional Graphs to Chapter: Results from Computation Time

The results in Chapter 6.1.2 are based on three more situations of feature detecting. Figure B.1 represents the total time used to detect features, descriptors extraction and matching keypoints for 689 features in tag 1. Figure B.2 represents the three similar steps for tracking 166 features in tag 9 and Figure B.3 represents 916 features in tag 9.

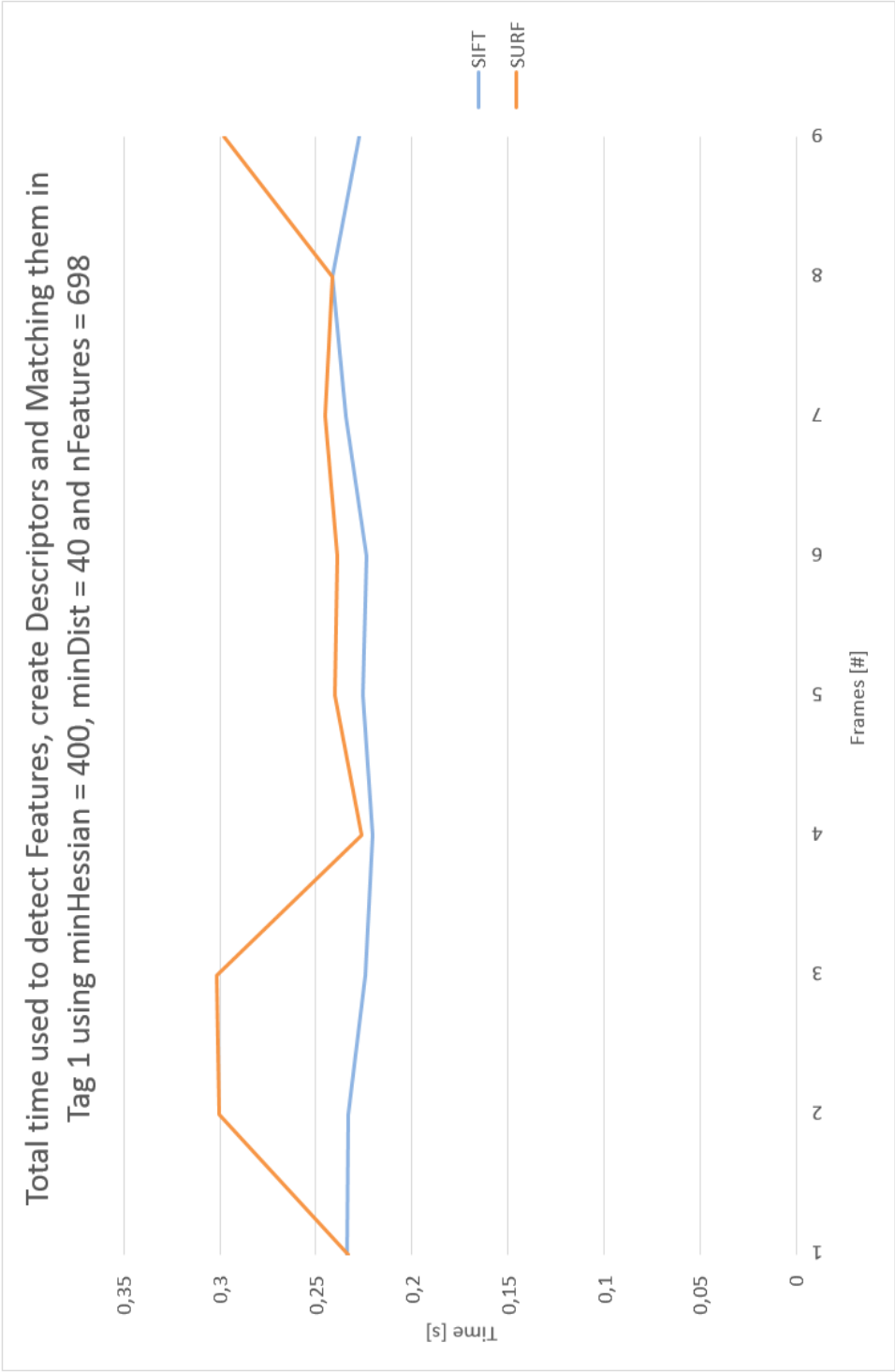


Figure B.1: Time used for detecting features, descriptor extraction and matching 698 keypoints using tag 1.

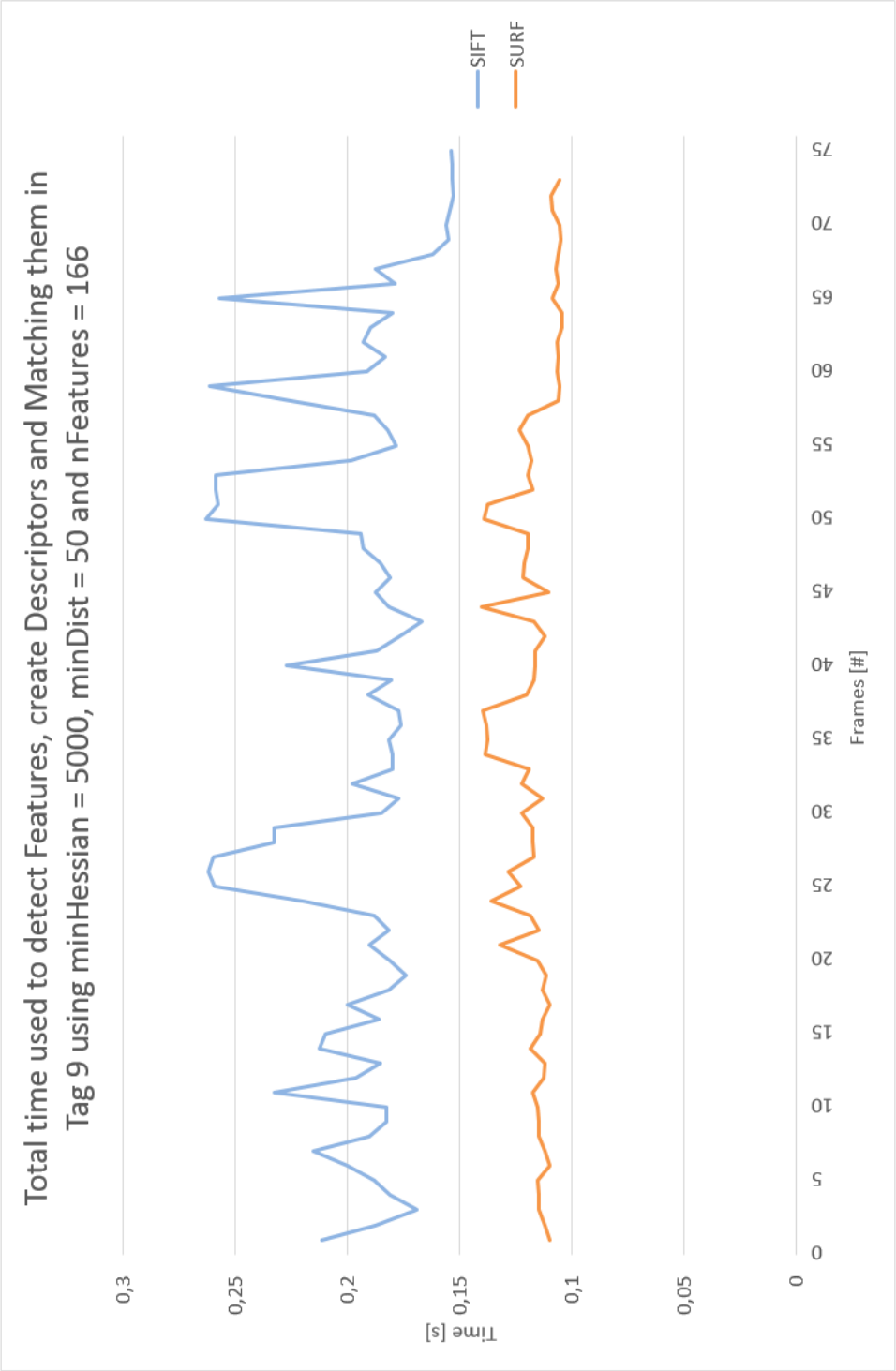


Figure B.2: Time used for detecting features, descriptor extraction and matching 166 keypoints using tag 9.

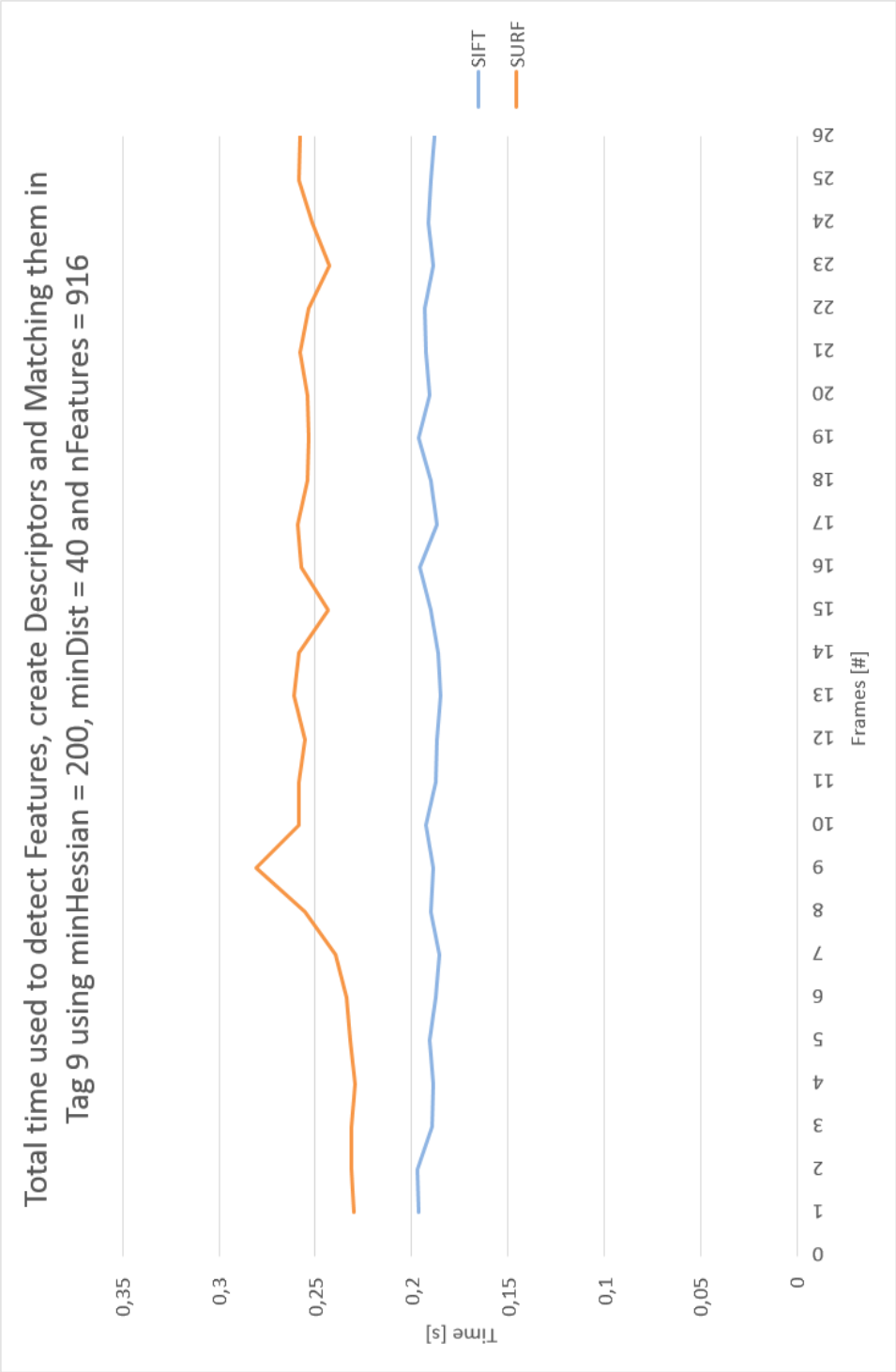


Figure B.3: Time used for detecting features, descriptor extraction and matching 916 keypoints using tag 9.

B.2 Additional Graphs for Section of Improvement to Computation Time

Additional graphs for section about improvement of computation time represented in Chapter 6.1.3. It was performed an experiment of combining the implementation of the SURF feature detector with the SIFT descriptor extractor, and comparing the result with a SIFT implementation. Figure B.4 represent the detection of 230 features in tag 1, Figure B.5 represent the detection of 185 features in tag 9 and Figure B.6 represent the detection of 230 features in tag 9.

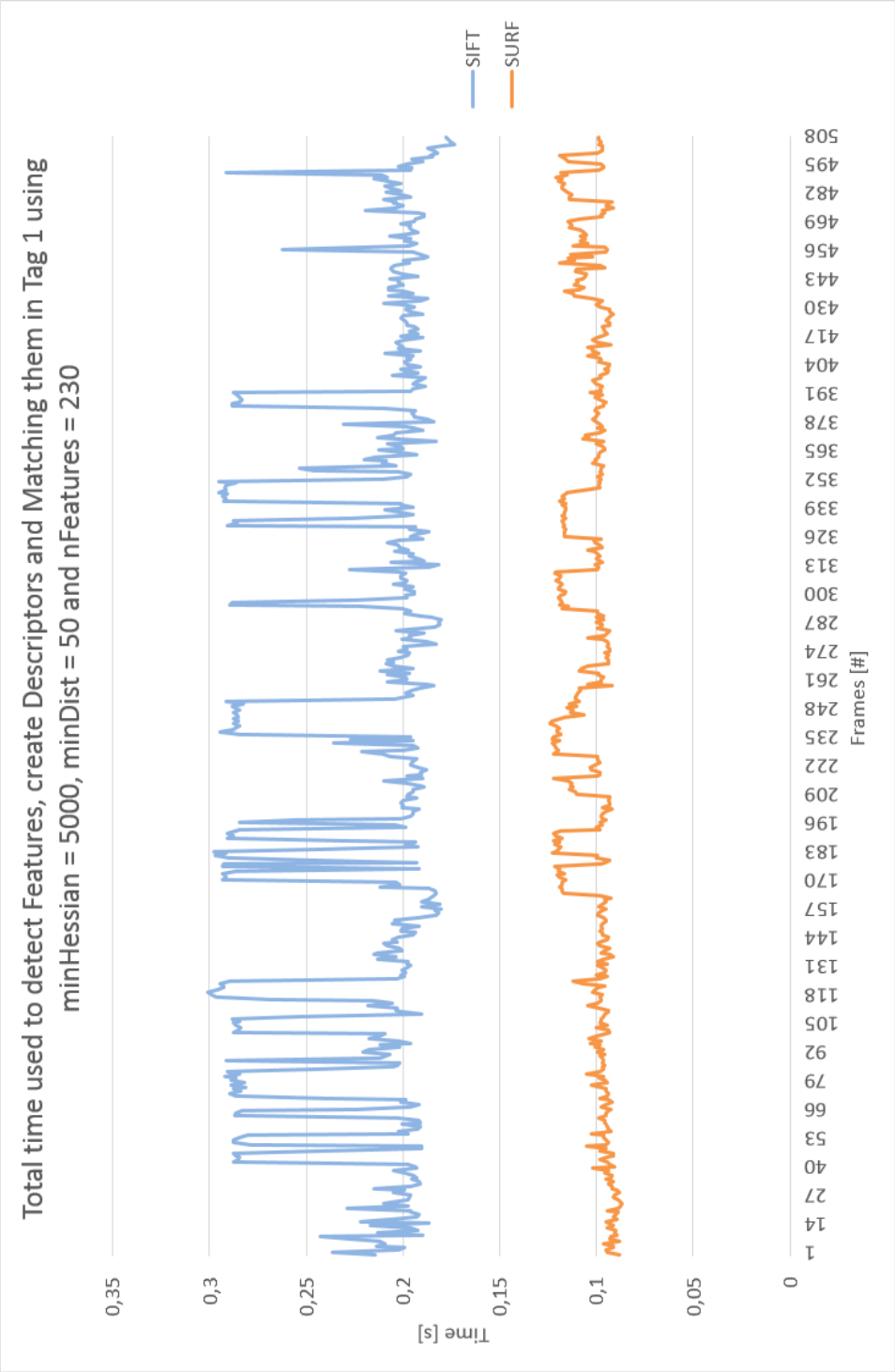


Figure B.4: Total time used to detect features, extract descriptors and matching 230 features in tag 1.

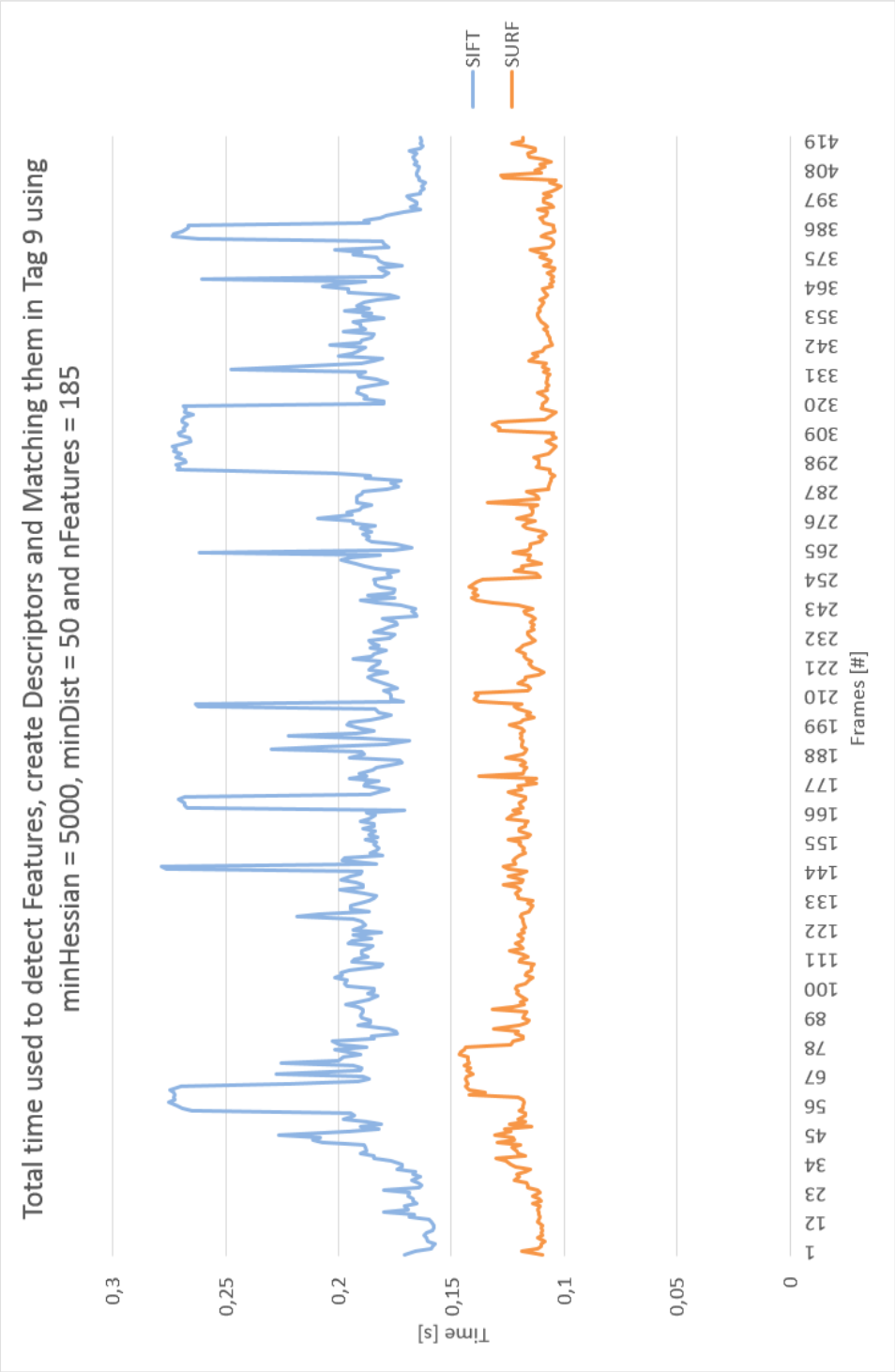


Figure B.5: Total time used to detect features, extract descriptors and matching 185 features in tag 9.

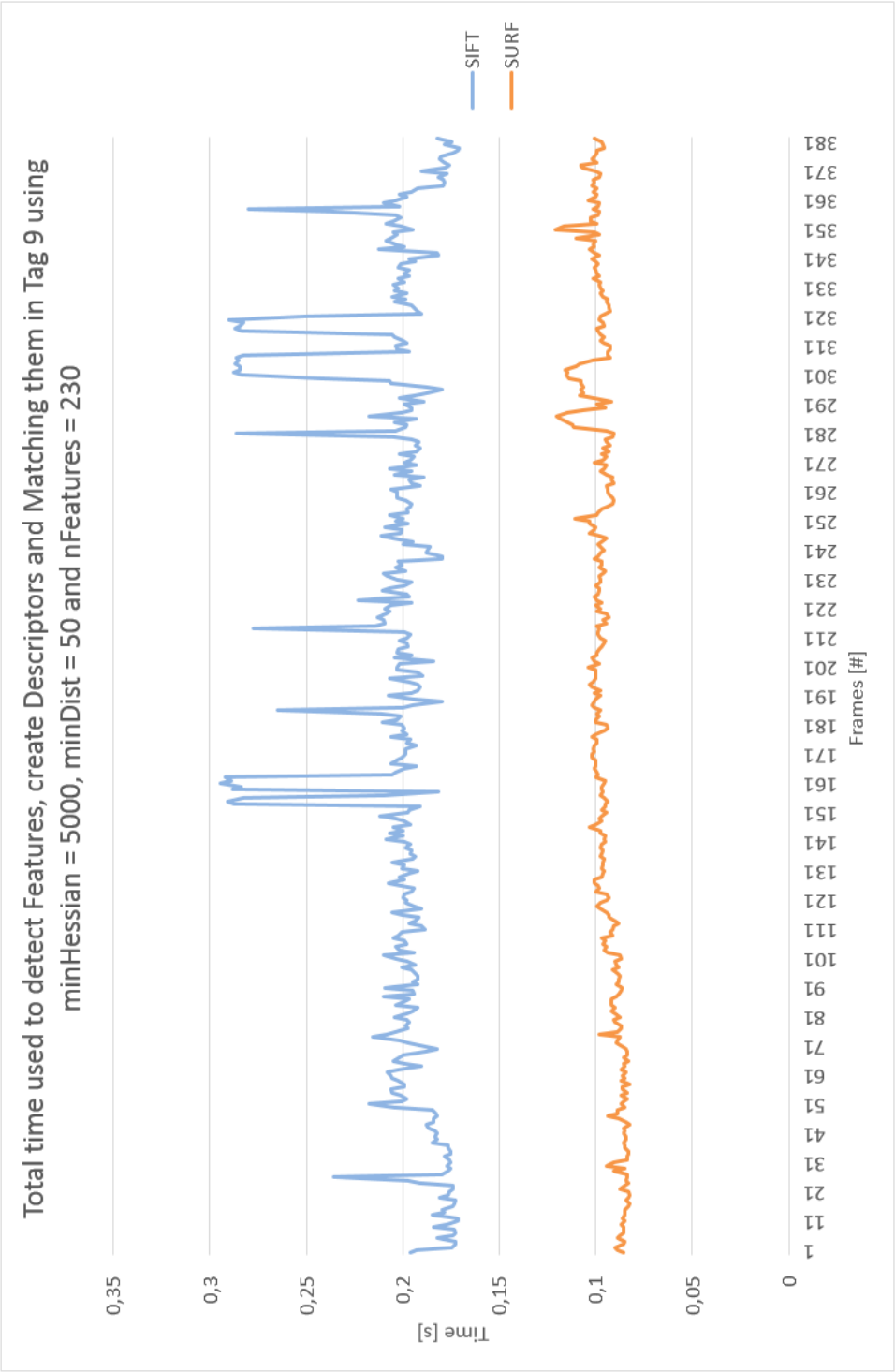


Figure B.6: Total time used to detect features, extract descriptors and matching 230 features in tag 9.

Appendix C

Source Code

```
/**
 * Tracking object in motion and logging behaviour of the algorithm SIFT.
 * The program is based on a tutorial from OpenCV:Features2D + Homography to find a known object
 *http://docs.opencv.org/doc/tutorials/features2d/feature_homography/feature_homography.html

 * Author: Jens Arne K. Engesaeter.
 * NTNU 2015
 */

//ROS
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include "std_msgs/Float64MultiArray.h"
#include <sensor_msgs/image_encodings.h>
#include <geometry_msgs/Pose.h>

//C++
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <fstream>
#include <cmath>
#include <vector>
#include <sstream>
#include <cstdio>
#include <iomanip>
#include <time.h>

//OpenCV
#include <opencv2/opencv.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/core/core.hpp"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/legacy/legacy.hpp>

//features - The number of best features to retain. The features are ranked by their scores (measured in SIFT algorithm as the local contrast)
//OctaveLayers - The number of layers in each octave. 3 is the value used in D. Lowe paper. The number of octaves is computed automatically from the image resolution.
//contrastThreshold - The contrast threshold used to filter out weak features in semi-uniform (low-contrast) regions. The larger the threshold, the less features are produced by the detector.
//edgeThreshold - The threshold used to filter out edge-like features. Note that the its meaning is different from the contrastThreshold, i.e. the larger the edgeThreshold, the less features are filtered out (more features are retained).
//sigma - The sigma of the Gaussian applied to the input image at the octave #0. If your image is captured with a weak camera with soft lenses, you might want to reduce the number.

//Default values
//double contrastThreshold = 0.04;
//double edgeThreshold = 10;
//double sigma = 1.6;

int  nFeatures = 0;
int nOctaveLayers = 5;
double contrastThreshold = 0.04;
```

```
double edgeThreshold = 10;
double sigma = 1.6;

std::clock_t total_time;
std::ofstream SIFTLogg;

class ImageConverter
{
private:
  ros::NodeHandle nh_;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;

public:

  ImageConverter()
  : it_(nh_)
  {
    // Subscribe to input video feed
    image_sub_ = it_.subscribe("usb_cam/image_raw", 1, &ImageConverter::imageCb, this);
  }

  ~ImageConverter()
  {
  }

  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {
    //Initalize cv bridge to operate images between ROS and OpenCV
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }

    //Initialize timers
    std::clock_t init_detect, init_descriptor, final;
    double final_detect, final_descriptor, final_time;

    //Initialize test image and convert it to gray scale
    cv::Mat img_object = cv::imread("/home/minions/workspaces/minion_ws/src/opencv_test/tag9.jpg", CV_LOAD_IMAGE_GRAYSCALE );
    cv::Mat image_scene = cv_ptr->image;

    std::vector<cv::KeyPoint> keypoints_object, keypoints_scene;
    std::vector< cv::DMatch > matches;

    // Step 1: Detect the keypoints using SIFT Detector

    cv::SiftFeatureDetector detector( nFeatures, nOctaveLayers, contrastThreshold, edgeThreshold, sigma );

    //Detect featuers and measure time
    init_detect=clock();
    detector.detect( img_object, keypoints_object );
    detector.detect( image_scene, keypoints_scene );
    final=clock()-init_detect;
    final_detect=(double)final / ((double)CLOCKS_PER_SEC) ;
    final =0;

```

```
    // Draw keypoints
    cv::Mat img_keypoints_1;
    cv::Mat img_keypoints_2;

    // Step 2: Calculate descriptors
    cv::SiftDescriptorExtractor extractor;

    cv::Mat descriptors_object, descriptors_scene;

    //Create descriptors and measure time
    init_descriptor=clock();
    extractor.compute( img_object, keypoints_object, descriptors_object );
    extractor.compute( image_scene, keypoints_scene, descriptors_scene );
    final=clock()-init_descriptor;
    final_descriptor=(double)final / ((double)CLOCKS_PER_SEC) ;
    final =0;

    //Draw keypoints
    drawKeypoints(img_object, keypoints_object, descriptors_object, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    drawKeypoints(image_scene, keypoints_scene, descriptors_scene, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

    //Write logged values to text file
    SIFTLogg << std::setw(25) << (double)final_detect <<";" <<
        std::setw(23) << keypoints_scene.size() <<";" <<
        std::setw(25) << final_descriptor <<";" <<
        std::setw(22) << final_time<< \n ;

    // Show detected keypoints
    imshow("Keypoints 1 SIFT", descriptors_object );
    imshow("Keypoints 2 SIFT", descriptors_scene );

    cv::waitKey(1);
  }
};

int main(int argc, char** argv)
{
  //Initalize a node
  ros::init(argc, argv, "sift");
  ros::NodeHandle n;
  total_time=clock();

  //Open logg file for writing
  SIFTLogg.open ("SIFT_Rapport_treshold.txt");

  //Add header to logg file
  SIFTLogg <<std::setw(20) << "nFeatures = " <<nFeatures<<"; nOctaveLayers = " <<nOctaveLayers<<"; contrastThreshold = "<<contrastThreshold<< "; edgeThreshold = "<<edgeThreshold<<"; sigma = "<<sigma<<";"<< \n << \n <<
        std::setw(20) << "Time to detect features[s];" <<
        std::setw(20) << "Ammount of features[#];" <<
        std::setw(20) << "Time to make descriptor[s];" <<
        std::setw(20) << "Total time [s];"<<  \n ;
  //Call object to use SIFT in a live stream
  ImageConverter ic;
  ros::spin();
  //close logg file
  SIFTLogg.close();

  return 0;
}
```

```
/**
 * Tracking object in motion and logging behaviour of the algorithm SURF.
 * The program is based on a tutorial from OpenCV:Features2D + Homography to find a known object
 * Author: Jens Arne K. Engesaeter.
 * NTNU 2015
 */

//ROS
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include "std_msgs/Float64MultiArray.h"
#include <sensor_msgs/image_encodings.h>
#include <geometry_msgs/Pose.h>

//C++
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <fstream>
#include <cmath>
#include <vector>
#include <sstream>
#include <cstdio>
#include <iomanip>
#include <time.h>

//OpenCV
#include <opencv2/opencv.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/core/core.hpp"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/legacy/legacy.hpp>


//int extended
//        0 means that the basic descriptors (64 elements each) shall be computed
//        1 means that the extended descriptors (128 elements each) shall be computed

//int upright
//        0 means that detector computes orientation of each feature.
//        1 means that the orientation is not computed (which is much, much faster). For example, if you match images from a stereo pair, or do image stitching, the matched features likely have very similar angles, and you can speed up feature extraction by setting upright=1.

//double hessianThreshold (minHessian)
//    Threshold for the keypoint detector. Only features, whose hessian is larger than hessianThreshold are retained by the detector. Therefore, the larger the value, the less keypoints you will get. A good default value could be from 300 to 500, depending from the image contrast.

//int nOctaves
//    The number of a gaussian pyramid octaves that the detector uses. It is set to 4 by default. If you want to get very large features, use the larger value. If you want just small features, decrease it.

//int nOctaveLayers
```

```
//    The number of images within each octave of a gaussian pyramid. It is set to 2 by default.

//Default values
//double minHessian = 350; // 300-400 is recomended
//int nOctaves=4;
//int nOctaveLayers=2;
//bool extended=true;
//bool upright=false;

double minHessian = 350;
int nOctaves=4;
int nOctaveLayers=5;
bool extended=true;
bool upright=false;

std::clock_t total_time;
std::ofstream SURFLogg;

class ImageConverter
{
private:
  ros::NodeHandle nh_;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;

public:

  ImageConverter()
  : it_(nh_)
  {
    // Subscribe to input video feed
    image_sub_ = it_.subscribe("usb_cam/image_raw", 1, &ImageConverter::imageCb, this);
  }

  ~ImageConverter()
  {
  }

  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {
    //Initalize cv bridge to operate images between ROS and OpenCV
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }

    //Initialize timers
    std::clock_t init_detect, init_descriptor, final;
    double final_detect, final_descriptor, final_time;

    //Initialize test image and convert it to gray scale
    cv::Mat img_object = cv::imread("/home/minions/workspaces/minion_ws/src/opencv_test/tag9.jpg", CV_LOAD_IMAGE_GRAYSCALE );
    cv::Mat image_scene = cv_ptr->image;

    std::vector<cv::KeyPoint> keypoints_object, keypoints_scene;
    std::vector< cv::DMatch > matches;

```

```
    // Step 1: Detect the keypoints using SURF Detector

    cv::SurfFeatureDetector detector( minHessian, nOctaves, nOctaveLayers, extended, upright );

    //Detect featuers and measure time
    init_detect=clock();
    detector.detect( img_object, keypoints_object );
    detector.detect( image_scene, keypoints_scene );
    final=clock()-init_detect;
    final_detect=(double)final / ((double)CLOCKS_PER_SEC) ;
    final =0;

    // Draw keypoints
    cv::Mat img_keypoints_1;
    cv::Mat img_keypoints_2;


    //-- Step 2: Calculate descriptors (feature vectors)
    cv::SurfDescriptorExtractor extractor;

    cv::Mat descriptors_object, descriptors_scene;

    //Create descriptors and measure time
    init_descriptor=clock();
    extractor.compute( img_object, keypoints_object, descriptors_object );
    extractor.compute( image_scene, keypoints_scene, descriptors_scene );
    final=clock()-init_descriptor;
    final_descriptor=(double)final / ((double)CLOCKS_PER_SEC) ;
    final =0;

    //Draw keypoints
    drawKeypoints(img_object, keypoints_object, descriptors_object, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    drawKeypoints(image_scene, keypoints_scene, descriptors_scene, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

    //Write logged values to text file

    SURFLogg << std::setw(25) << (double)final_detect <<";" <<
            std::setw(23) << keypoints_scene.size() <<";" <<
            std::setw(25) << final_descriptor <<";" <<
        std::setw(22) << final_time<< \n ;

    // Show detected keypoints
    imshow("Keypoints SURF 1", descriptors_object );
    imshow("Keypoints SURF 2", descriptors_scene );

    cv::waitKey(1);
  }
};

int main(int argc, char** argv)
{
  //Initalize a node
  ros::init(argc, argv, "surf");
  ros::NodeHandle n;
  total_time=clock();

  //Open logg file for writing
  SURFLogg.open ("SURF_Rapport_treshold.txt");

  //Add header to logg file
  SURFLogg <<std::setw(20) << "minHessian = " <<minHessian<<"; nOctaves = " <<nOctaves<<"; nOctaveLayers = "<<nOctaveLayers<< "; extended = "<<extended<<"; upright = "<<upright<<";"<< \n << \n <<
```

```
181          std::setw(20) << "Time to detect features[s];" <<
182          std::setw(20) << "Ammount of features[#];" <<
183          std::setw(20) << "Time to make descriptor[s];" <<
184          std::setw(20) << "Total time [s];"<<  \n ;
185
186   //Call object to use SIFT in a live stream
187   ImageConverter ic;
188   ros::spin();
189   //close logg file
190   SURFLogg.close();
191
192   return 0;
193 }
```

```
/**
Grasping a moving object using a combination of SIFT/SURF and a KUKA Agilus KR 6 R900 ↩
    sixx robot.
The program is based on a tutorial from OpenCV:Features2D + Homography to find a known↩
     object
Author: Jens Arne K. Engesaeter.
NTNU 2015
 */

//ROS
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include "std_msgs/Float64MultiArray.h"
#include <sensor_msgs/image_encodings.h>
#include <geometry_msgs/Pose.h>

//C++
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <fstream>
#include <cmath>
#include <vector>
#include <sstream>
#include <cstdio>
#include <iomanip>
#include <time.h>

//Eigen
#include <Eigen/Eigen>

//OpenCV
#include <opencv2/opencv.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/core/core.hpp"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/legacy/legacy.hpp>

//Robot
#include "robot_gripper.hpp"
#include "robot_planning_execution.hpp"
#include "robot_option_flag.hpp"

//Initilize gripper and KUKA Agilus robot
ih::RobotGripper gripper(ih::ROBOT_OPTION_VERBOSE_INFO);
ih::RobotPlanningExecution* ag1;

//Global values

//Matrix
Eigen::MatrixXd T_R_B(4,4);
Eigen::MatrixXd T_R_TAG(4,4);
Eigen::MatrixXd T_R_IC(4,4);

Eigen::MatrixXd T_IC_TC(4,4);
Eigen::MatrixXd T_IC_TAG(4,4);
Eigen::MatrixXd T_IC_TAG_start(4,4);
Eigen::MatrixXd T_IC_TAG_end(4,4);

```

```
Eigen::MatrixXd T_TAG_TC_start(4,4);
Eigen::MatrixXd T_TAG_TC_end(4,4);

//Average position after i_mages have been checked
double avg_X_pix_center = 0.0;
double avg_Y_pix_center = 0.0;

double X_pix_center_start = 0.0;
double Y_pix_center_start = 0.0;
double X_pix_center_end = 0.0;
double Y_pix_center_end = 0.0;

//Poses for movit
double movit_pose_x;
double movit_pose_y;
double movit_pose_z;

//Timer
double timeObjToPose =0;
double stop_watch =0;

//Flags
bool obj_detected = false;
bool object_grabed= false;
bool pose_given = false;
bool time_measured = false;
bool robot_pose_calculated = false;
bool timeOfangleVelosity=false;

//Radius of train track
const double r = 0.32;

static const std::string OPENCV_WINDOW = "Good Matches & Object detection";

double AngleVelosity(Eigen::MatrixXd T_IC_TAG_start, Eigen::MatrixXd T_IC_TAG_end, double stop_watch)
{
  //Initilize variables for estimation of angle velocity
  double angleVelosity =0;

  //Start pos
  double start_end_x =0;
  double start_end_y=0;

  //Vectors
  double AB=0;
  double AC=0;
  double BC=0;
  double pose_length=0;
  double L_start_end=0;

  //Circumference
  double L_tot=0;

  //Angle
  double theta =0;

  //Timers
  double lengthGripPos =0;
  double time_start_end =0;

  //Homogeneous transformations matrix from image center to tag center
  T_IC_TC << 1, 0, 0, -0.08,
         0, 1, 0, -0.385,
         0, 0, 1, 0,
         0, 0, 0, 1;
```

```
  //Homogeneous transformations matrix
  T_TAG_TC_start << T_IC_TAG_start.inverse()*T_IC_TC;
  T_TAG_TC_end << T_IC_TAG_end.inverse()*T_IC_TC;

  //Coordinates for grasping position
  start_end_x = T_TAG_TC_start(12) - T_TAG_TC_end(12);
  start_end_y = T_TAG_TC_start(13) - T_TAG_TC_end(13);

  //Vectors
  AB= std::sqrt(std::pow(T_TAG_TC_start(12),2)+std::pow(T_TAG_TC_start(13),2));
  AC= std::sqrt(std::pow(T_TAG_TC_end(12),2)+std::pow(T_TAG_TC_end(13),2));
  BC =  std::sqrt(std::pow(start_end_x,2)+std::pow(start_end_y,2));

  pose_length =  std::sqrt(std::pow(start_end_x,2)+std::pow(start_end_y,2));
  std::cout<< "Length: "<<pose_length<<std::endl;
  theta=2*asin((pose_length*0.5)/r);
  std::cout<<"Radians: "<< theta<<std::endl;

  L_tot = 2*M_PI*r;
  angleVelosity=L_tot/stop_watch;
  lengthGripPos=r*theta;
  timeObjToPose = lengthGripPos/angleVelosity;

  std::cout<<"Curve Length: "<<lengthGripPos<<std::endl;
  std::cout<<"Circumference: "<<L_tot<<std::endl;
  std::cout<<"Degrees: "<<(180*theta)/M_PI<<std::endl;
  std::cout<<"AngleVelosity: "<<angleVelosity<<std::endl;
  std::cout<<"timeObjToPose: "<< timeObjToPose<<std::endl;

  return timeObjToPose;
}

class ImageConverter
{
private:
  ros::NodeHandle nh_;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;
  ros::Publisher world_pos;

public:

  ImageConverter()
  : it_(nh_)
  {
    // Subscrive to input video feed and publish output video feed
    image_sub_ = it_.subscribe("usb_cam/image_raw", 1, &ImageConverter::imageCb, this);

    cv::namedWindow(OPENCV_WINDOW);
  }

  ~ImageConverter()
  {
    cv::destroyWindow(OPENCV_WINDOW);
  }

  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
```

```
    {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }

    //Distance from camera to tag in meters
    double Z_world = 0.656;

    //Poses
    double X_world_center_start;
    double Y_world_center_start;
    double X_world_center_end;
    double Y_world_center_end;

    double avg_X_world0 = 0.0;
    double avg_Y_world0 = 0.0;

    //Timers
    std::clock_t timer,grab_obj,final;
    double time;

    //Transformation from robot to image center
    T_R_IC << 0,-1, 0,-0.76938,
         -1, 0, 0, 0.59726,
          0, 0,-1, 0.17484,
          0, 0, 0, 1;

    //Camera matrix
    Eigen::Matrix3d camMatrix;
    camMatrix << 516.496569, 0.000000, 309.255248,
           0.000000, 516.477238, 268.673559,
           0.000000, 0.000000, 1.000000;

    Eigen::Matrix3d camMatrix_inv;
    camMatrix_inv = camMatrix.inverse();

    //Initalize test image
    cv::Mat img_object = cv::imread("/home/minions/workspaces/minion_ws/src/↩
        opencv_test/tag1.jpg", CV_LOAD_IMAGE_GRAYSCALE );

    cv::Mat image_scene = cv_ptr->image;

    int nFeatures = 5000;
    std::vector<cv::KeyPoint> keypoints_object, keypoints_scene, keypoints_scene2 ;
    std::vector< cv::DMatch > matches, matches2;


    //Step 1: Detect the keypoints using SIFT Detector
    cv::SiftFeatureDetector detector( nFeatures );
    cv::SiftFeatureDetector detector2( nFeatures );

    //Define mask in start position
    //Type of mask is CV_8U
    cv::Mat mask = cv::Mat::zeros(image_scene.size(), CV_8U);
    //Restriction of pixel values
    cv::Mat roi(mask, cv::Rect(290,200,40,40));
    roi = cv::Scalar(255, 255, 255);

    detector.detect(image_scene, keypoints_scene, mask);

    //Define mask in end position. The object will be grabed in this position.
    // type of mask is CV_8U
    cv::Mat mask2 = cv::Mat::zeros(image_scene.size(), CV_8U);
    //Restridtion of pixel values
    cv::Mat roi2(mask2, cv::Rect(530,30,40,40));
    roi2 = cv::Scalar(255, 255, 255);
```

```
    detector2.detect(image_scene, keypoints_scene2, mask2);

    //MASK FINISH

    detector.detect( img_object, keypoints_object );
    detector2.detect( img_object, keypoints_object );


    // Step 2: Calculate descriptors (feature vectors)
    cv::SiftDescriptorExtractor extractor;

    cv::Mat descriptors_object, descriptors_scene, descriptors_scene2;

    extractor.compute( img_object, keypoints_object, descriptors_object );
    extractor.compute( image_scene, keypoints_scene, descriptors_scene );
    extractor.compute( image_scene, keypoints_scene2, descriptors_scene2 );

    // Draw keypoints
    cv::Mat img_keypoints_1; cv::Mat img_keypoints_2; cv::Mat img_keypoints_3;
    //test = keypoints_object[1];

    drawKeypoints( img_object, keypoints_object, img_keypoints_1, cv::Scalar::all(-1),↩
        cv::DrawMatchesFlags::DEFAULT );
    drawKeypoints( image_scene, keypoints_scene, img_keypoints_2, cv::Scalar::all(-1),↩
        cv::DrawMatchesFlags::DEFAULT );
    drawKeypoints( image_scene, keypoints_scene2, img_keypoints_3, cv::Scalar::all(-1)↩
        , cv::DrawMatchesFlags::DEFAULT );

    // Show detected (drawn) keypoints
    //imshow("Keypoints 1", img_keypoints_1 );
    //imshow("Keypoints 2", img_keypoints_2 );
    //imshow("Keypoints 3", img_keypoints_3 );
    cv::circle(image_scene, cv::Point(0,0), 10, CV_RGB(255,0,0));
    cv::circle(image_scene, cv::Point(309.255248,268.673559), 10, CV_RGB(255,0,0));

    cv::waitKey(1);

    // Step 3: Matching descriptor vectors using FLANN matcher
    cv::FlannBasedMatcher matcher, matcher2;

    matcher.match( descriptors_object, descriptors_scene, matches );
    matcher2.match( descriptors_object, descriptors_scene2, matches2 );

    double max_dist = 0; double min_dist = 70;

    //Matches in first mash
    // Quick calculation of max and min distances between keypoints
    for( int i = 0; i < descriptors_object.rows; i++ )
    { double dist = matches[i].distance;
      if( dist < min_dist ) min_dist = dist;
      if( dist > max_dist ) max_dist = dist;
    }
    //-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
    std::vector< cv::DMatch > good_matches;

    for( int i = 0; i < descriptors_object.rows; i++ )
    { if( matches[i].distance < 3*min_dist )
      { good_matches.push_back( matches[i]); }
    }

    cv::Mat img_matches=cv_ptr->image;

    cv::drawMatches( img_object, keypoints_object, image_scene, keypoints_scene,
      good_matches, img_matches, cv::Scalar::all(-1), cv::Scalar::all(-1),
      cv::vector<char>(), cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

```

```
    //-- Localize the object from obj in scene
    std::vector<cv::Point2f> obj;
    std::vector<cv::Point2f> scene;

    for( size_t i = 0; i < good_matches.size(); i++ )
    {
      // Get the keypoints from the good matches
      obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
      scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
    }

    //Matches in second mask
    //-- Quick calculation of max and min distances between keypoints
    for( int i = 0; i < descriptors_object.rows; i++ )
    { double dist = matches2[i].distance;
      if( dist < min_dist ) min_dist = dist;
      if( dist > max_dist ) max_dist = dist;
    }

    // Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
    std::vector< cv::DMatch > good_matches2;

    for( int i = 0; i < descriptors_object.rows; i++ )
    { if( matches2[i].distance < 3*min_dist )
      { good_matches2.push_back( matches2[i]); }
    }
    cv::Mat img_matches2=cv_ptr->image;

    cv::drawMatches( img_object, keypoints_object, image_scene, keypoints_scene2,
      good_matches2, img_matches2, cv::Scalar::all(-1), cv::Scalar::all(-1),
      cv::vector<char>(), cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

    // Localize the object from obj in scene
    std::vector<cv::Point2f> obj2;
    std::vector<cv::Point2f> scene2;


    for( size_t i = 0; i < good_matches2.size(); i++ )
    {
      //-- Get the keypoints from the good matches
      obj2.push_back( keypoints_object[ good_matches2[i].queryIdx ].pt );
      scene2.push_back( keypoints_scene2[ good_matches2[i].trainIdx ].pt );
    }

    //If tag is detected start pos is defined and the timer is activated
    if(good_matches.size() > 0 && obj_detected == false){
      timer=clock();

      std::cout<<"Object is detected!"<<std::endl;
      std::cout<<"Time: "<<time<<std::endl;
      std::cout<<"Center position: 1"<<X_pix_center_start<<", "<<Y_pix_center_start<<std::endl;
      std::cout<<"Good matches 1: "<<good_matches.size()<<std::endl;

      //Estimates position when the tag was first detected
      for( int i_image = 0; i_image < good_matches.size(); i_image++ ){

    avg_X_pix_center = avg_X_pix_center + (double)keypoints_scene[ good_matches[i_image].trainIdx ].pt.x;
    avg_Y_pix_center = avg_Y_pix_center + (double)keypoints_scene[ good_matches[i_image].trainIdx ].pt.y;
    }
      X_pix_center_start = avg_X_pix_center/(good_matches.size());
      Y_pix_center_start = avg_Y_pix_center/(good_matches.size());

      avg_X_pix_center = 0.0;
```

```
386         avg_Y_pix_center = 0.0;
387
388         std::cout<<X_pix_center_start<<", "<<Y_pix_center_start<<std::endl;
389
390         //Convert pixel coordinates to Cartesian coordinates
391         Eigen::MatrixXd pixCor_center_start(3,1);
392         pixCor_center_start <<  X_pix_center_start,
393                     Y_pix_center_start,
394                     1.000000;
395
396         Eigen::MatrixXd norCor_center_start(3,1);
397         norCor_center_start << camMatrix_inv*pixCor_center_start;
398
399         norCor_center_start = norCor_center_start*Z_world;
400
401         X_world_center_start = (double)(norCor_center_start(0));
402         Y_world_center_start = (double)(norCor_center_start(1));
403
404         //Transformation from camera center to start position
405         T_IC_TAG_start <<  1,0,0, (X_world_center_start),
406                0,1,0, (Y_world_center_start),
407                0,0,1, -0.11,
408                0,0,0, 1;
409
410         std::cout<<"T_IC_TAG_start: \n"<<T_IC_TAG_start<<std::endl;
411
412         obj_detected = true;
413         }
414
415       //If tag is detected in second mask, the angle velocity and gashping position may ↩
            be estimated
416       else if(good_matches2.size() > 0 && robot_pose_calculated == false &&  ↩
            obj_detected == true){
417
418         //Estimates position to grab object
419         for( int i_image = 0; i_image < good_matches2.size(); i_image++ ){
420
421       avg_X_pix_center = avg_X_pix_center + (double)keypoints_scene2[ good_matches2[↩
            i_image].trainIdx ].pt.x;
422       avg_Y_pix_center = avg_Y_pix_center + (double)keypoints_scene2[ good_matches2[↩
            i_image].trainIdx ].pt.y;
423       }
424         X_pix_center_end = avg_X_pix_center/(good_matches2.size());
425         Y_pix_center_end = avg_Y_pix_center/(good_matches2.size());
426
427         avg_X_pix_center = 0.0;
428         avg_Y_pix_center = 0.0;
429
430
431         std::cout<<"Time: "<<time<<std::endl;
432         std::cout<<"Center position 2: "<<X_pix_center_end<<", "<<Y_pix_center_end<<std↩
              ::endl;
433         std::cout<<"Good matches 2: "<<good_matches2.size()<<std::endl;
434
435         //Convert pixel coordinates to Cartesian coordinates
436         Eigen::MatrixXd pixCor_center_end(3,1);
437         pixCor_center_end <<  X_pix_center_end,
438                   Y_pix_center_end,
439                   1.000000;
440
441         Eigen::MatrixXd norCor_center_end(3,1);
442         norCor_center_end << camMatrix_inv*pixCor_center_end;
443
444         norCor_center_end = norCor_center_end*Z_world;
445
446         X_world_center_end = (double)(norCor_center_end(0));
```

```
      Y_world_center_end = (double)(norCor_center_end(1));

      //Transformation from camera center to end position
      Eigen::MatrixXd T_IC_TAG_end(4,4);
      T_IC_TAG_end <<  1,0,0, (X_world_center_end),
               0,1,0, (Y_world_center_end),
               0,0,1, -0.11,
               0,0,0, 1;


      //Tag pose related to the robot base
      T_R_TAG <<  T_R_IC*T_IC_TAG_end;

      std::cout<<"T_IC_TAG_end: \n"<<T_IC_TAG_end<<std::endl;
      std::cout<<"Robot POSE: \n"<<T_R_TAG<<std::endl;


      //Adding offset related to Movit
      movit_pose_x = T_R_TAG(12) + 0.000509;
      movit_pose_y = T_R_TAG(13) -0.6025+0.0275;
      movit_pose_z=1.17;


      //Request robot to grabbing position
      ag1.goToPoseByXYZ(-movit_pose_x,movit_pose_y,movit_pose_z);

      std::cout<<"Movit!"<<movit_pose_x<<", "<<movit_pose_y<<", "<<movit_pose_z<<std::endl;
      robot_pose_calculated = true;
      }

    //Stop watch stops when object is observed in start position for a secound
    else if(good_matches.size() > 0 && obj_detected == true && time > 10 && time_measured==false){
      final=clock()-timer;
      stop_watch=(double)final / ((double)CLOCKS_PER_SEC);
      final=0;
      timer = clock();
      std::cout<<"Stop_watch!"<<stop_watch<<std::endl;

      //Move gripper down
      ag1.goToRelativePoseByXYZ(0,0,-0.13);

      time_measured=true;
      }
    //Estimates time to gab object
    else if(time_measured==true && robot_pose_calculated == true && object_grabed==false){
      if(timeOfangleVelosity==false){
    timeObjToPose = AngleVelosity(T_IC_TAG_start,T_IC_TAG_end, stop_watch);
    timeOfangleVelosity=true;
    }
      time==0;
      final=clock()-timer;
      time=(double)final / ((double)CLOCKS_PER_SEC);
      std::cout<<"Time: "<<time<<std::endl;

      if((time + 1.4)>timeObjToPose){
    std::cout<< "GRAB OBJECT!"<<std::endl;
    std::cout <<  "Gripping" << std::endl;

    //Gripper closes
    gripper.closeGripper();
    ros::Duration(0.5).sleep();

    //Lift object
```

```
    ag1.goToRelativePoseByXYZ(0,0,0.13);

    //Move to center of the train track and put down the object
    ag1.goToPoseByXYZ(-0.40,0,1.3);
    ag1.goToRelativePoseByXYZ(0,0,-0.19);
    ros::Duration(7).sleep();

    //Open gripper
    gripper.openGripper();

    //Robot goes to start position
    ag1.goToRelativePoseByXYZ(0,0,0.19);
    ag1.goToPoseByXYZ(0.05,0.05,1.3);

    //End loop
    object_grabed= true;
    }
      }

    else if(obj_detected == true && good_matches.size() == 0 && object_grabed== false)↩
        {
      //No object detected
      final=clock()-timer;
      time=(double)final / ((double)CLOCKS_PER_SEC);
      }
    else{}

    if( !world_pos ) {
      ROS_WARN("Publisher invalid!");
      }
    //-- Show detected matches
    cv::imshow( OPENCV_WINDOW, image_scene );
    cv::waitKey(1);
  }
};

int main(int argc, char** argv)
{
  ros::init(argc, argv, "image_converter");
  ros::NodeHandle n;
  //Initilize a robot node
  ag1 = new ih::RobotPlanningExecution("agilus1", 0.1, 1, 10);
  //Moves robot to start pose
  ag1.goToPoseByXYZ(0.05,0.05,1.3);

  ImageConverter ic;
  ros::spin();

  return 0;
}
```

Appendix D

Digital Appendix

A .zip called Digital_Appendix_JensArneEngesaeter.zip is included as a digital appendix. This file contains:

- The .mp4 file, Grasping_an_object_in_motion.mp4 is a additional movie to illustrate the experiment 2 in Chapter 5.2, where a KUKA Agilus grasping an object in motion according to observations from a Logitech Webcam C930e.
- The .cpp file, track_train_movie.cpp is the source code for implementation of experiment 2 in Chapter 5.2.
- The .mkv file, stor_tag1_mindist30_hassian5000_nfeatures0.mkv is a additional movie illustrate the tracking in experiment 1 in Chapter 5.1.1.
- The .cpp file, object_tracker_SIFT.cpp is the source code for implementation of experiment 1 in Chapter 5.1.2.
- The .cpp file, object_tracker_SIFT.cpp is the source code for implementation of experiment 1 in Chapter 5.1.2.
- The folder, Tags contains the ten different tags used in experiment 1 in Chapter 5.1.1 and Chapter 5.1.2.

Bibliography

[1] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346 – 359, 2008.

[2] Herbert Bay, Beat Fasel, and Luc Van Gool. Interactive museum guide: Fast and robust recognition of museum objects. *First International Workshop on Mobile Vision*, 2006.

[3] Jason Clemons. SIFT: Scale invariant feature transform by david lowe. Lecture, Opened 02.10.2014 .

[4] Peter Corke. *Robotics, Vision and Control*. Springe, 2013.

[5] OpenCV dev team. Feature Detection and Description. `http://docs.opencv.org/modules/nonfree/doc/feature_detection.html?highlight=sift`. Opened 16.05.2015.

[6] Christopher Evans. Notes on the OpenSURF library. Technical Report CSTR-09-001, University of Bristol, January 2009.

[7] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision, Second Edition*. Cambridge University Press, 2004.

[8] IFR. World Robotics 2014 Industrial Robots. `http://www.ifr.org/industrial-robots/statistics/`, 2014. Opened 20.10.2014.

[9] itseez. About opencv. `http://opencv.org/about.html`. Opened 16.05.2015.

[10] Luo Juan and Oubong Gwun. A comparison of sift, pca-sift and surf. *International Journal of Image Processing (IJIP)*, 3(4):143–152, 2009.

[11] Point Cloud Library. 3D SIFT in PCL. `http://docs.pointclouds.org/trunk/classpcl_1_1_s_i_f_t_keypoint.html#details`. Opened 21.05.2015.

[12] T. Lindeberg. Scale Invariant Feature Transform. *Scholarpedia*, 7(5):10491, 2012. revision 142692.

[13] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[14] David G Lowe. Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image, March 23 2004. US Patent 6,711,293.

[15] Alexander Mordvintsev and Abid K. Feature Matching. `http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html`. Opened 08.05.2015.

[16] Open Source MoveIt! Concepts. http://moveit.ros.org/documentation/concepts/#The_move_group_node. Opened 21.05.2015.

[17] Richard J Radke. *Computer Vision for Visual Effects*. Cambridge University Press, 2013.

[18] ROS. About ros. http://www.ros.org/about-ros/. Opened 01.05.2015.

[19] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[20] Wikipedia. Grayscale. http://en.wikipedia.org/wiki/Grayscale, 2 oktober 2014. Opened 05.12.2014.

[21] Andrew P Witkin. Scale-space filtering: A new approach to multi-scale description. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 150–153. IEEE, 1984.