

**ETH** zürich

Bachelor Thesis

# **Hand Tracking using Kalman Filter for Safe Human-Robot Interaction**

Hasret Gümgümü & Daniel Laumer

June 2, 2017

Institute of Geodesy and Photogrammetry  
ETH Zurich

Professorship  
Prof. Dr. Konrad Schindler

Supervisors  
Dr. Jan Dirk Wegner  
Dr. Mathias Rothermel  
Dr. Paul Beardsley (Disney Research)

---

## Abstract

The goal of this thesis is to develop an algorithm to detect human activity with the use of an Intel RealSense camera and determine if the activity intrudes into a specified safety zone. The Intel RealSense is a depth camera, which means that we know the distance from the camera to the objects in the scene for every pixel. Imagine a tabletop robot which interacts with people surrounding the table and a certain safety zone around the robot which is to ensure that it can shut down its movements when someone reaches too near. Since it is a tabletop setup, the main interest is on the hands. The Intel RealSense camera can deliver tracked coordinates of hand joints which can be used as a signal of human activity. In order to be able to keep the safety zone small and at the same time reduce false positives, we use a Kalman Filter with which we are able to solve two problems at once: With the prediction model of the filter we can calculate forecast coordinates and therefore detect the hand movements which are likely to enter the safety zone before they are too near to the robot. On the other hand the outputs of the filter (filtered and smoothed coordinates) can be used to make a nice visualization.

We implemented an application which takes the coordinates of the 22 tracked hand joints and filters them individually with a Kalman Filter. The prediction works smoothly and we are now able to foresee quick hand movements and signal those in direction of the camera as potentially hazardous. The time of how much the prediction goes into the future can be adjusted freely. The filter smooths the raw coordinates as desired and gives a nice visualization of the hand. To ensure an overall security also other objects besides the hand can be recognized. We added another module which uses the 3D coordinates of the depth information to also check them against the security zone. This part does not work with prediction, it is more like a safety net to make the application more reliable. The main result, the software, can be a ground source for further research purposes and a potential starting base to implement a profound safety zone for a real tabletop robot.

---

## Acknowledgement

We would like to thank the following people:

- Prof. Dr. Konrad Schindler for making this thesis and the cooperation with Disney Research possible.
- Dr. Jan Dirk Wegner for the constant support, valuable inputs during our meeting sessions and guiding us to the right direction.
- Dr. Mathias Rothermel for the technical support, the presence for questions and valuable feedback.
- Dr. Paul Beardsley for the whole organization at Disney Research and being always a contact person for questions and giving new ideas for improvement.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theoretical Principles</b>	<b>3</b>
2.1	Computer Vision . . . . .	3
2.1.1	Introduction . . . . .	3
2.1.2	History of Computer Vision . . . . .	3
2.1.3	Projective Geometry . . . . .	4
2.2	Tracking . . . . .	6
2.2.1	Interest point detection . . . . .	7
2.2.2	Feature Tracking . . . . .	8
2.2.3	Face Tracking . . . . .	9
2.2.4	Hand Tracking . . . . .	13
2.3	Low Cost Depth Cameras . . . . .	15
2.3.1	Structured Light Method . . . . .	15
2.3.2	Intel RealSense SR300 . . . . .	16
2.3.3	Kinect . . . . .	19
2.4	Filtering Problems . . . . .	20
2.4.1	Introduction . . . . .	20
2.4.2	Bayes Filters . . . . .	22
2.4.3	Gauss-Markov Estimation . . . . .	23
2.4.4	Sequential Gauss-Markov resp. Static Kalman Filter . . . . .	24
2.4.5	Kalman Filter and EKF . . . . .	25
2.4.6	Monte-Carlo Simulations . . . . .	27
2.4.7	Particle Filter . . . . .	28
2.4.8	Unscented Kalman Filter UKF . . . . .	29
<b>3</b>	<b>Methodology</b>	<b>30</b>
3.1	General Idea . . . . .	30
3.2	SARI . . . . .	32
3.2.1	Safety Realization . . . . .	32
3.2.2	Safety Zone Setup . . . . .	32
3.3	Visualization . . . . .	34
3.4	Applied Kalman Filter . . . . .	38
3.4.1	Motivation . . . . .	38

3.4.2	General Principle . . . . .	38
3.4.3	Prediction Step . . . . .	39
3.4.4	Correction Step . . . . .	41
3.4.5	Iterative Calculation . . . . .	42
3.4.6	Building of the Matrices . . . . .	42
3.4.7	Prediction . . . . .	45
3.5	Implementation . . . . .	46
<b>4</b>	<b>Results and Discussion</b>	<b>50</b>
4.1	Initialization of the Filter . . . . .	50
4.2	Filtering by the Kalman Filter . . . . .	51
4.3	Gesture Recognition . . . . .	53
4.4	Hand Tracking . . . . .	55
4.5	Discussion . . . . .	56
<b>5</b>	<b>Conclusion</b>	<b>57</b>
<b>6</b>	<b>Outlook</b>	<b>58</b>
<b>References</b>		<b>59</b>
<b>Appendix</b>		<b>63</b>

# 1 Introduction

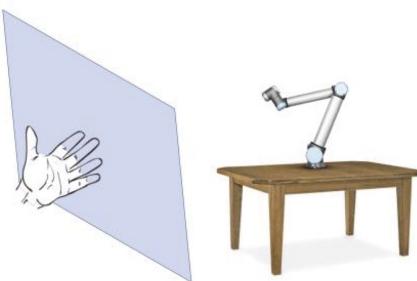
Nowadays there is a huge change in the use and importance of vision based applications especially with automation in many big markets. Many autonomous processes need to have the ability to 'see', to conceive and understand the surroundings in order to react properly. However, there are still unsolved problems and questions in this area. Especially when we look at the research in robotics and suchlike topics, which depend on reliable vision based methods for the automation process to get along with the environment, a question arises: How can we ensure the safety of such human-robot interactions? As long as the person reacts as imagined by the programmer everything is fine, but a human is not a robot and can act unexpectedly. In this thesis we drill down on this safety problem and develop a security algorithm which protects interacting people of being hurt from a movement of the robot and at the same time the robot of getting broken.

The initial setup for our thesis is the following: There is a robot on top of a table which is supposed to be able to interact with humans. This interaction could be for example playing board games. The robot has an Intel RealSense camera mounted on its top which is a so called depth camera that can deliver 3D coordinates for each pixel of the image.

The goal of this thesis is now the following:

**Design an algorithm which uses the functionalities of the Intel RealSense and filtering methods to establish a safety zone around the robot and issues a warning when something enters that zone.**

**Easy Solution:** Large safety zone



**Our Solution:** Small zone, with prediction

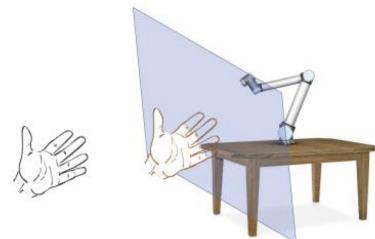


Figure 1: The two possible solutions to solve the safety problem

Since it is a tabletop interaction, the main focus is on the hands. The Intel RealSense has a hand tracking option included, so it is possible to get the raw coordinates of 22 joints at any time a hand is in the field of view. An option would be to use this coordinates and check, if they are behind the safety zone. But then safety zone would have to start far away from the robot in order to be safe, because if a hand is coming with high velocity, there should be enough time to shut down. But this produces many false negatives, since many hand movements will not reach the robot. In this thesis we want to go a step further. The algorithm should distinguish if the hand is just near the robot but does not move much, or if it is likely to enter the zone. This calls for a prediction of the movement. This is achieved by using the Kalman Filter (KF), with which we are able to solve two problems at once: The prediction can be made using the prediction model integrated into the KF and at the same time we are able to filter the raw coordinates and use them to generate a nice smooth visualization.

This thesis therefore combines topics from Computer Vision like tracking and recognition of humans with filtering problems used in many areas as for example navigation.

## 2 Theoretical Principles

### 2.1 Computer Vision

#### 2.1.1 Introduction

As described in Szeliski (2010, pp. 3 – 10), with the automation, Computer Vision became an important research area in recent years. For new technologies, for example autonomous systems and robotics, the information of the environment is essential for the robot in order to be able to interact with it. Computer Vision works with mathematical techniques to generate three dimensional objects from the information of two dimensional image data. Stereo matching makes this possible. With several pictures from the same object but from different perspective it is possible to reconstruct the 3D object.

The big dream of computer vision is to achieve algorithms which operate at the same level as a small child observes his environment and processes information from images. This may be achieved in the distant future. Nowadays, the research is still far away from reaching this goal, so that it remains a dream.

What a human brain handles effortlessly, like human intuition, is the catchiest for computer algorithms. For that reason, the complexity of computer vision is underestimated by laymen. It is tricky though because it must deal with inverse problematics like reconstructing the reality from images with less information (from 2D to 3D).

Statistical approaches with applicable probability distribution are common for modeling and detecting noise afflicted pictures. It is important to provide algorithms which are resilient towards noise and variations from the models and efficient in memory space and run-time. For this, Bayesian techniques can be used, for example a Bayes Filter (see section 2.4.2).

#### 2.1.2 History of Computer Vision

A short insight into the history of computer vision with information from Szeliski (*ibid.*, pp. 10 – 19) is provided in this section. Its beginning is in the early 70's. What at that time separated computer vision clearly from the already existing image processing was to regain the real 3D structure of an object from images and as a result to open new research areas with 3D objects, like the edge detection which was one of the first. Later, in the 80's the focus was more on complex mathematical techniques for quantitative analyses of images. It was possible to reconstruct a 3D object from light, texture and orientation.

Merging from different image data and reconstruction of surfaces were important topics and also improved edge and shape recognition.

Projective invariants for recognition were used for eliminating motion from structure. The development of factorization techniques for improved efficiency could be used for orthographic mapping and later for perspective cases. The notion of full global optimization<sup>1</sup> was introduced in the community but later it was ascertained that it is the same as the bundle adjustment already known in photometrically applications. The optical flow method<sup>2</sup> and dense stereo correspondence algorithms (important for photogrammetry especially) were improved. At that time, also tracking and intensity based algorithms (for human face and body) progressed. A stronger interaction between computer graphics and computer vision was possible in those days with revolutionary image morphing algorithms (flowing of multi-images into each other) which could be used for 3D animations with images from the real world.

Post-millennially, the research grew with a fast speed. There were four important trends:

- Strong connection between computer vision and computer graphics.
- New knowledge for digital photography like illumination transition with morphing.
- Development of feature-based techniques and more efficient global optimization algorithms.
- Complex machine learning technique connected with computer vision, which opened new branch of research until today.

### 2.1.3 Projective Geometry

In Computer Vision, one of the main tools is the camera. It provides the knowledge about the environment. The camera makes a perspective of the scene. To work with such kind of data, Cartesian coordinates are not ideal. This is why projective coordinates are often used instead to simplify the calculations. The whole knowledge of projective geometry is from the lecture documents of K.Schindler (2016).

---

<sup>1</sup>Full Global Optimization: A subdomain of applied mathematics and numerical analysis which has the goal to find optimal elements  $\mathbf{x}$  from a set of  $\mathbf{X}$  elements according to a set of criteria  $F = \{f_1, f_2, \dots, f_n\}$ . These criteria (mathematical functions) are called objective functions. An objective function  $f : \mathbf{X} \mapsto Y$  with  $Y \subseteq \mathbb{R}$  is a mathematical function which is subject to optimization.(Weise 2009, pp. 21)

<sup>2</sup>Optical Flow Method: It detects velocity of real world objects (e.g. cars on the highway) from the image through comparing changes of pixel values in specific time interval. Usually, objects near the camera have longer motion vector than further afar objects with same absolute speed. It can be also used for situation where the camera is moving and the objects are stationary. This method is especially used in computer vision to detect movements in video streams oder tracking systems.(MathWorks 2017)

Its origin comes from Pappus von Alexandria which mentioned the first projective coherences 340 A.D. It is an alternative algebraic representation of geometric objects (e.g. point, lane, plane, ...) beside the usually known euclidean representation. The main use is for optics because of the essential simplification of equations. It should be mentioned that the choice of the algebraic representation does not influence the geometric relations. All calculations can be done similarly with euclidean representations. It serves only for the sake of convenience.

**Homogeneous coordinates Def:** The representation  $\mathbf{q}$  of a geometric object is homogen if  $\mathbf{q}$  and  $\lambda\mathbf{q}$  represent the same object. Homogeneous coordinates are used in projective geometry similar as Cartesian coordinates are used in euclidean geometry.

A 2D point  $\mathbf{x} = (x, y)$  in homogeneous coordinates is represented as followed:

$$\mathbf{x} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} \quad (1)$$

The definition of a homogeneous line  $\mathbf{l}$  2D looks like this:

$$\mathbf{l} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad \text{with } ax + by + c = 0 \quad (2)$$

**Geometric Operations** For some specific computations the homogeneous coordinates have big advantage over the Cartesian coordinates.

To determine if a point is on a line we only need to calculate the dot product and check if it is 0:

$$\mathbf{x}^T \mathbf{l} = \mathbf{l}^T \mathbf{x} = \mathbf{x} \cdot \mathbf{l} = 0 \quad (3)$$

or

$$au + bv + cw = w(ax + by + c) = 0 \quad (4)$$

The intersection of two lines is their cross product.

$$\mathbf{x} = \mathbf{l} \times \mathbf{m} \quad (5)$$

It is also an easy way to determine the line between two points. It is their cross product as well:

$$\mathbf{l} = \mathbf{x} \times \mathbf{y} \quad (6)$$

Something inconvenient in Cartesian coordinates is representing infinite points which are commonly used in homogeneous coordinates. An infinite point  $\mathbf{x}_\infty$  and infinite line  $\mathbf{l}_\infty$  have this form:

$$\mathbf{x}_\infty = \begin{bmatrix} u \\ v \\ 0 \end{bmatrix} \quad \mathbf{l}_\infty = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (7)$$

With 7 we can see that all far points lie on a far line

$$\mathbf{x}_\infty^T \mathbf{l}_\infty = 0 \quad (8)$$

and the intersection of parallel lines

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \times \begin{bmatrix} a \\ b \\ d \end{bmatrix} = \begin{bmatrix} bd - bc \\ ac - ad \\ ab - ab \end{bmatrix} = \begin{bmatrix} u \\ v \\ 0 \end{bmatrix} \quad (9)$$

is a far point.

For a point in 3D space all vectors are extended with a z-value.

**Transformation** Projective transformation (projection, homography) is a reversible linear mapping

$$\mathbf{x}'_n = M_{n \times n} \mathbf{x}_n . \quad (10)$$

**Law of projective geometry:** Every unambiguous line-preserving representation of a projective space  $\mathbb{P}^n$  on itself is a homography, for  $2 \leq n < \infty$ .

All unique line-preserving transformations are linear in homogeneous coordinates.

## 2.2 Tracking

Since the beginning of Computer Vision big progress has been made for tracking algorithms. One easy possibility for tracking human body parts is extracting the shape with the use of edge detection and doing that in every frame. But this is already an outdated method. Another technique is using interest point extraction to get some actual coordinates of some landmarks of the tracked object. This is further explained in section 2.2.1. To make this kind of algorithm more reliable, we need scale invariant transforms like SIFT (Scale Invariant Feature Transform) or SURF (Speeded-Up Robust Features) (Herbert Bay et al. 2006; Li Deng 2014).

Nowadays there is a whole new area of methods which contains Machine Learning and

Deep Learning as explained in Li Deng (2014, p.230-234). The problems of the previous mentioned methods are that they only get the low level information of the pictures. More interesting is to know mid- or even high-level edge information like edge intersection or object parts. This can be achieved using Deep Learning techniques by letting the algorithm learns hierarchies of visual features by itself. Using Pose Estimation, the tracking can be improved significantly and make it to the state of art solution (Ali Erol et al. 2007).

In the next sections we introduce the interest point detection as a basis algorithm and focus then on the tracking and detection of body parts like hands and face.

### 2.2.1 Interest point detection

The knowledge of interest point detection and all information given in this section is based on J.Wegner (2016). Interest point detection is used in image processing for finding distinctive points in an image and giving its parameter. Distinctive points in general are unique in its narrow environment. Additional requirements are for example to be invariant towards image variations like radiometric and geometric contortions (e.g. rotation, scaling). Two of the most known operators for interest point detection are Harris and Förstner. We will give a short introduction and show the process of each operator. Both only work with gray value images.

**Autocorrelation** The autocorrelation matrix is the basis for the following operators. We consider  $g(x, y)$  as the gray value of the image at position  $x$  and  $y$ . When we take the square sum of the gray level differences by shifting of the window  $W$  about  $(\Delta x, \Delta y)$  we get

$$c(x, y, \Delta x, \Delta y) = \sum_{x, y \in W} p(x, y)[g(x + \Delta x, y + \Delta y) - g(x, y)]^2 \quad (11)$$

with  $p(x, y)$  as weight function (e.g. for smoothing). An approximation with Taylor expansion gives us the following form:

$$c(x, y, \Delta x, \Delta y) \approx (\Delta x, \Delta y) \cdot M \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \quad (12)$$

with

$$M = \begin{bmatrix} g_x^2 & g_x g_y \\ g_y g_x & g_y^2 \end{bmatrix}. \quad (13)$$

The diagonal elements of  $M$  are smoothed quadratic first derivation and its non-diagonal elements are mixed product of the first derivation.  $M$  is also known as *2nd moment matrix* or *autocorrelation matrix*.

**Harris-Operator** The Harris operator compares a cornerness  $r$  with a threshold value to decide if it is an interest point or not.

$$r = \det(M) - \kappa \cdot \text{trace}(M)^2 = \lambda_1 \cdot \lambda_2 - \kappa \cdot (\lambda_1 + \lambda_2)^2 \quad (14)$$

In (14) we can see that the eigenvalues  $\lambda_1$  and  $\lambda_2$  of  $M$  must be high to increase  $r$  respectively the cornerness.

**Förstner-Operator** This algorithm uses a slightly different autocorrelation matrix. The only difference is that the gradients are smoothed with an additional derivation scale  $\sigma_\Delta$ :

$$g_x = \frac{\partial G(x, y, \sigma_\Delta)}{\partial x} \cdot g \quad (15)$$

$$g_y = \frac{\partial G(x, y, \sigma_\Delta)}{\partial y} \cdot g \quad (16)$$

The elements of  $M$  are smoothed with a Gauss-Filter  $G(x, y, \sigma_I)$  with integration-scale  $\sigma_I$ :

$$M = \begin{bmatrix} G(x, y, \sigma_I) \cdot g_x^2 & G(x, y, \sigma_I) \cdot g_x \cdot g_y \\ G(x, y, \sigma_I) \cdot g_y \cdot g_x & G(x, y, \sigma_I) \cdot g_y^2 \end{bmatrix} = M(x, y, \sigma_\Delta, \sigma_I) \quad (17)$$

We exploit the fact that  $M^{-1}$  is the covariance respectively it says how precisely the point can be determined. The axis of the ellipse of uncertainty corresponds with the eigenvalues  $\lambda_1$  and  $\lambda_2$  of the matrix  $M^{-1}$ .

To find out if it is a point of interest its ellipse of uncertainty has to be as small as possible and as round as possible. The size of the ellipse of uncertainty can be judged with the weight

$$\omega = \frac{1}{\lambda_1 + \lambda_2} = \frac{1}{\text{trace}(M^{-1})} = \frac{\det(M)}{\text{trace}(M)} \quad (18)$$

and roundness

$$q = 1 - \left(\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}\right)^2 = 4 \cdot \frac{\det(M)}{[\text{trace}(M)]^2} \quad (19)$$

### 2.2.2 Feature Tracking

This section is especially undertaken from Szeliski (2010, pp. 235 – 237).

Tracking is mostly used in video applications. The usual workflow is to initialize the object at first and afterwards track it by searching the same object in the following frame. To avoid big computational time, the searching algorithm will not be over the whole picture but rather in the same area as the object was in the last frame. This is possible because

the time step between the frames are normally a few milliseconds. The hierarchical search strategy searches first in a low resolution picture and after finding the right area there is a further search in high resolution in the same area. For that reason, it is more difficult to track objects with high speed. The methods vary depending on the purpose. To solve the problem of different illuminations, normalized cross-correlation can be used. Tie points should be in positions with high gradients for reliable detecting. In homogeneous (e.g. clear sky, house wall) and totally heterogeneous (e.g. meadow) areas it is almost impossible to get reliable tie points.

To find the new position on the current picture, we compare it to a certain reference image which shows the static situation. We need this static background to detect the motion by overlapping the images and finding the differences. For the reference image we have two possibilities:

- Always use the same (first) image and track the whole movement from beginning to the current state.
- Just use two consecutive pictures and determine the relative movement. The problem here is that it is not robust against outliers since we do not have the whole information.

The affine motion model combines these two methods: We use the translation between two following frames and lead it back to the first reference image. This way we get the absolute approximative translation vector. In the end we make an affine transformation to get the right shape and orientation of the current state.

Machine learning for computer vision advanced tracking in the last few years rapidly. Algorithms can be trained and become more stable and robust. Important features can be learned and used.

### 2.2.3 Face Tracking

Face tracking is an important but challenging task for computer vision, since the faces change constantly dependent on the orientation, lightning and shadows. Additionally, every person has a different face, so the tracking algorithm cannot just look for a fixed reference object like in feature tracking. But before a face can be tracked it has to be recognized:

**Eigenfaces** Matthew Turk (1991) developed an efficient way for recognition of human faces, so called eigenface approach. Relevant information from the image will be extracted

and compared one face encoding with all other from the database to detect any characteristics for the specific face. For that reason, enough data from one face is needed in the beginning for finding all relevant information to encode it.

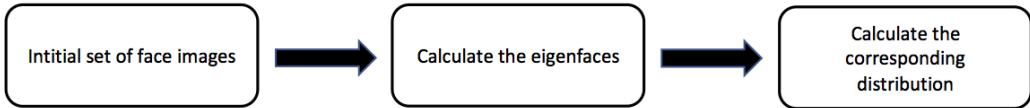


Figure 2: Initialization approach to face recognition with eigenfaces

This means mathematically, that the eigenvectors of the covariance matrix of the set of face images is needed. Each eigenvector shows the difference of the face images in an ordered way. They can be seen as a set of features which characterize the variation between face images because each position of the image is connected to each eigenvector. All eigenvectors can be displayed and are called *eigenfaces*. Each image can be reconstructed with a linear combination of the eigenfaces. The eigenfaces with the largest eigenvalues are the "best" because they provide the biggest variance in the face images. Therefore, the best  $M$  eigenfaces span a  $M$ -dimensional subspace.

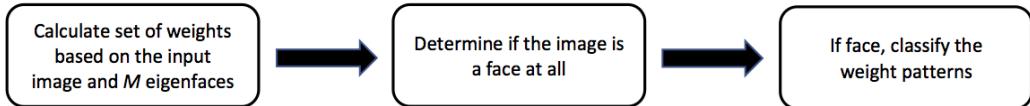


Figure 3: Recognition of new face images after initialization

For recognition of new face images we have optional features like updating the eigenfaces and weight patterns or if the same unknown face is seen many times, its characteristic weight patterns can be calculated and included into the known faces. For the process we need in the beginning  $\Gamma_1, \Gamma_2, \dots, \Gamma_M$  images of the face. We create an average face  $\Psi$  from the training set:

$$\Psi = \frac{1}{M} \sum_{i=1}^M \Gamma_i \quad (20)$$

With this average face we can create faces of difference  $\Phi_i$ :

$$\Phi_i = \Gamma_i - \Psi \quad (21)$$

With all faces of difference  $\Phi_i$  the covariance matrix  $C$  can be created:

$$C = \frac{1}{M} \sum_{i=1}^M \Phi_i \Phi_i^T = AA^T \quad \text{where } A = [\Phi_1 \Phi_2 \cdots \Phi_M] \quad (22)$$

For the reason that the eigenvectors of  $C$  looks similar to faces, Turk and Pentland gave them the name *eigenfaces*. For efficiency reason a new matrix  $L$  will be calculated as followed:

$$L = A^T A \quad (23)$$

Because of the smaller dimension of  $L$  its eigenvectors  $v_l$  can be calculated easier. It is only an interim step to get the eigenvectors  $u_l$  of  $C$ .

$$u_l = \sum_{k=1}^M v_{lk} \Phi_k \quad \text{with } l = \{1, \dots, M\} \quad (24)$$

or

$$u_l = Av_l \quad (25)$$

From the eigenvectors  $u_l$  of  $C$  are only  $M'$  (with  $M' < M$ )  $u'_l$  with the highest eigenvalues of importance. For the last step we have to normalize the eigenvectors  $u'_l$ .

This introduction to face tracking was undertaken from the publications by Fei Yang et al. (2012) about face tracking with a consumer depth camera.

The existing face tracking algorithms with a normal camera without depth information, can be divided into two classes:

- *Appearance based methods:* Use of 3D Models and deform them according to the detected face. This means that the whole face gets tracked, not only some points. This technique needs a training phase with a data bank of faces. The modeling includes geometry and texture. This technique is often performed with the use of eigenfaces (see section 2.2.3)
  - Because of different lightning and deformation it is not easy to generalize this method, good training data is needed.
- *Feature based methods:* Tracking distinctive points of the face, so called landmarks which should be local, unique and robust against change. They are found with interest point extraction like explained in section 2.2.1. To avoid errors, these landmarks have to be held in constraints, which are realized by a global shape model. This technique uses only geometric data and there is no training needed. → This method is better generalizable for other faces but is not that stable.

This techniques can now be improved by integrating depth data. We will mention two possibilities here:

In addition to the landmarks (feature based method), the orientation of the head can be calculated (pose estimation) with depth data. Orientation is often very inaccurate in classical methods because there is only frontal 2D data available. With the depth data, the head pose can now be estimated as an regression problem. One possibility to do this are Regression Forests<sup>3</sup>.

A second improvement can be done in the finding of the landmarks. This is normally done by template matching. The idea of template matching is to always check, if there is a point which matches better. This can be done by minimizing the Mahalonobis distance  $d$ :

$$d^2 = (g - \bar{g})^T \cdot S^{-1} \cdot (g - \bar{g}) \quad (26)$$

$g$  is the gradient of the current position and  $\bar{g}$  and  $S$  are mean and covariance matrix from the training phase. This method can sometimes deliver a bad performance in differencing between landmarks due to poor lightning or complex background. This can be optimized by using edge detection in the depth map. So the new distance to be minimized reads as following:

$$d^2 = (g - \bar{g})^T \cdot S^{-1} \cdot (g - \bar{g}) + \|\nabla l\|^2 \quad (27)$$

$\|\nabla l\|^2$  is the additional term which describes the edge detection in the depth image. This is done especially for the landmarks on the edge of the face because there the spatial gradient is the highest.

These techniques can be also used for interactive deformation of a 3D graphics model based on tracking a user's motion as described in Szeliski (2010, pp. 237 – 238). In the initial phase, landmarks will be set to specific points. From this data a morphing with an avatar will connect the real human face with a virtual 3D avatar. The landmarks can be tracked in real-time with fast feature tracking and the animation can be fitted to the real human face, so that the orientation and facial features looks similar.

This was only an example of using tracking data from a human face. There are many other possibilities to work with tracking data which will not be mentioned in this work.

---

<sup>3</sup>Regression Forests is a regression method based on Random Forests. Random Forests is a segmentation algorithm using several search trees in order to split the dataset according to the homogeneity. This can also be used for regression problems since the model which is fitted in a regression analysis is also a kind of a border between two regions, which can be determined. This is then called Regression Forests. (Ho 1995)

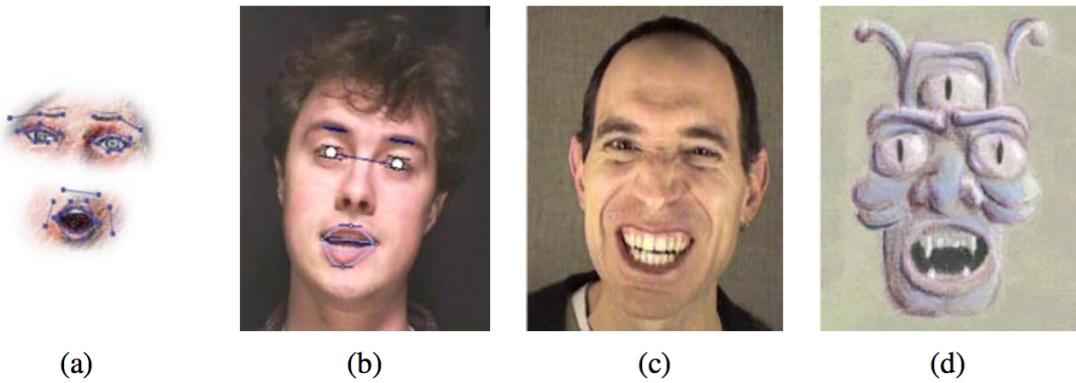


Figure 4: Animation example of a real human face tracking joint to an avatar (Szeliski 2010, pp.238).

#### 2.2.4 Hand Tracking

There are lots of approaches in research and industry how hand tracking from image data can be executed. A human hand is not one of the easiest parts of the human body to recognize and track. One main reason is the high degree of freedom, which is for a realistic model at least 26 (Matthieu Bray et al. 2004). In figure 5 we can see one of the possibilities how a human hand can be represented with 22 joints. Although free hand movement is still a big challenge, the research goes deeper and focus on hands holding and manipulating objects because in many applications this will be the case (Henning Hamer et al. 2009).

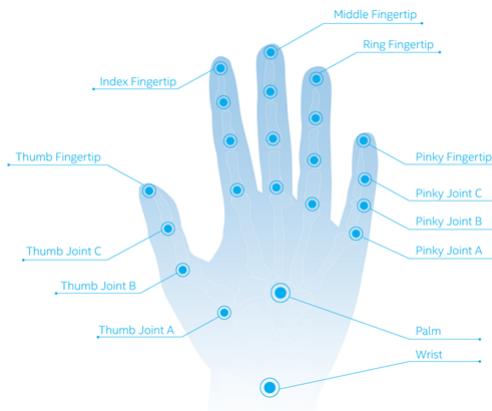


Figure 5: All detectable joints with the Intel RealSense camera (Intel 2017)

As Zhou Ren et al. (2013) described, these low cost 3D cameras have satisfying algorithms implemented for bigger objects like human body or human face. With the hand, the big problem is its relative small size. In general, 3D low cost cameras have a resolution of  $640 \times 480$  Pixel (Nowadays even higher). This is one of the main points why hand tracking comes to its limits. Due to the low resolution, the noise is relatively big. An additional problem is the computational cost which hinders to increase the resolution.

From our own experience with the Intel RealSense we learned that the camera operates satisfactorily after it recognizes and initializes a hand but for Human-Computer-Interaction (HCI) is only the hand tracking not enough but gesture recognition is essential.

**Gesture** In the future, applications with gesture recognition will become more and more important. Especially research and development in virtual reality (VR) promotes this. When VR becomes more common in the industry there will be a huge need in reliable and robust gesture recognition algorithms. 3D cameras like Intel RealSense (see section 2.3.2) and Kinect (see section 2.3.3) have already such functions implemented which however reach the limit of usage. On one hand the number of recognizable gestures is limited, on the other hand the recognition is in some cases not reliable. It also differs between the gestures: Gestures without overlapping objects (v-sign, hand wave, etc.) are more reliable than overlapping gestures (e.g. fist, grab, thumbs up etc.).

According to Zhou Ren et al. (ibid.) there are two main methods for gesture recognition from depth data:

- For the pose of the palm and the angles of the joints colored markers are used to extract high-level features, such as the fingertip, joint locations or some anchor points on the palm.
- Representation of the hand region by edges or an ellipse using skin color mode.

However, both cases still perform inefficiently in cluttered environments.

## 2.3 Low Cost Depth Cameras

Not long ago, the first industrial usable low cost depth cameras were introduced and sold in the market. The procedure is for the most low cost depth cameras roughly the same. There are three main components: A low resolution RGB color camera, a low resolution IR camera and an IR laser projector to collect the data provided from the IR camera. The difference to common 2D color cameras are the additional integrated IR components which makes it possible to get the 3<sup>rd</sup> dimension, the depth data. The process with structured light will be explained in the following section.

These cameras are easy to handle and can connect with different devices. Despite the fact that they only cost a few hundred dollars and are affordable for almost everyone they provide more than enough functionalities and satisfying accuracy for a private person to use them in a useful way.

### 2.3.1 Structured Light Method

There are several possibilities to gain depth by a camera,

- Stereo triangulation: Use more than one camera, find corresponding points and perform a triangulation. This is an application of stereo-photogrammetry.
- Structured light: Illuminate the scene with a pattern of light and analyze the deformation of the pattern.
- Time-of-Flight: Radar-like method (e.g. LiDAR).
- Interferometry: Measure the phase shift of the reflected to the source light and deduce the depth.

(Wikipedia 2017)

The Intel RealSense camera uses the method of structured light. There, in addition to the normal camera, we need a light projector and a second camera. The light projector beams some kind of structure, barcode or pattern onto the scene. The second camera records the reflected pattern. Often the projected light is infrared (IR) which implies that the second camera has to be a IR camera. With the distortion of the incoming pattern, the distance to every pixel in the field of view can be calculated.

To make such kind of calculations easier, homogeneous coordinates as explained in section 2.1.3 can be used since they are specifically designed for projective computations.

The pattern has to be unique, and this can be achieved by different colors, gray values or shapes of the pattern. Another method is to vary the projected light over time, so it can have both spatial and temporal component. This is called multiplexing which can be divided into four categories:

- Wavelength multiplexing: Use different wavelengths, which means different colors.
- Range multiplexing: Use just one wavelength, but multiple gray values (intensity).
- Temporal multiplexing: Change the projected pattern over time.
- Spatial multiplexing: Project a distinctive pattern with different shapes and sizes.

(P. Zanuttigh et al. 2016, p.54)

A big advantage of the structured light method for example to scanning systems is, that it is much faster, because the whole scene can be acquired at once and not pixel by pixel. A disadvantage to stereo triangulation is, that it relies on reflectance of the IR light. When dark objects are in the scene, they absorb most of the light and for that reason the object is not visible in the depth map. But an advantage is, that no corresponding points have to be searched, which makes it much more stable and faster.

### 2.3.2 Intel RealSense SR300

Intel provides three different cameras with RealSense technology (F200, SR300 and R200). SR300 is the next generation of F200 which adds new features and has some improvements over the F200 model. The focus of both cameras are the users. Both cameras use coded light depth technology to create a 3D depth video stream at close range. The SR300 provides a faster depth mode which allows a reduction in exposure time and allows motions up to 2 m/s. (Nhuy L. (Intel) 2016) The newest camera is the R200. Its focus is on the world, not the user, it is a rear facing camera. (Colleen C. (Intel) 2015)

The choice of the camera depends on the purpose. In this thesis only user activity is important. For that reason the Intel RealSense front-facing camera SR300 is appropriate. The color camera works with 720p with 60fps or when using the higher resolution of 1080p there are 30fps (Nhuy L. (Intel) 2016).

The information of the following section is from Intel SR300 Datasheet (2016).

The device can be used for many different applications:

- Face Analytics and Tracking

- Scanning and Mapping
- Scene Segmentation
- Hand and Finger Tracking
- Augmented Reality

There are also other integrated functionalities like voice recognition and more but these are not of big importance for this thesis. Its user friendly SDK provides a huge toolbox for development and the possibility to use it for many different applications. Also the first laptops already came to the market with an integrated Intel RealSense camera. In future 3D cameras like Intel RealSense will be more common than today for the simple reason that there will be much more applications in the market available for their usage and the development in HCI is getting better from day to day.



Figure 6: SR300 camera model (Nhuy L. (Intel) 2016)

## Components

- Infrared Camera: 640 x 480 pixel. It has a sensor aspect ratio of 4:3 with a wide field of view (vertical:  $55^\circ \pm 2^\circ$ ; horizontal:  $71.5^\circ \pm 2^\circ$ ; diagonal:  $88^\circ \pm 3^\circ$ ).
- Color Camera: 2MP (1920 x 1080 pixel) color camera with integrated ISP<sup>4</sup> and a sensor aspect ratio of 16:9. The field of view (vertical:  $41.5^\circ \pm 2^\circ$ ; horizontal:  $68^\circ \pm 2^\circ$ ; diagonal:  $75.2^\circ \pm 4^\circ$ ) is more limited than the infrared camera.

---

<sup>4</sup>Image Signal Processor

- Infrared Laser Projector: Coded light with a nominal laser wavelength of 860 nm and a field of projection of  $60^\circ \pm 4^\circ$  vertical and  $5^\circ \pm 2^\circ$  horizontal.

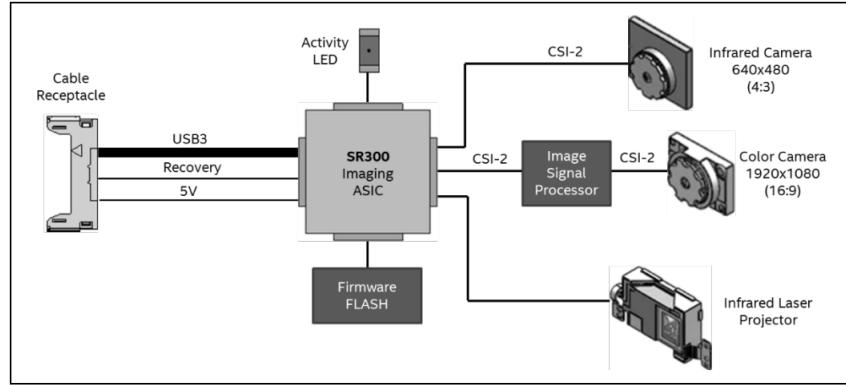


Figure 7: Embedded 3D imaging system of the Intel RealSense SR300 camera (Intel SR300 Datasheet 2016)

**Functionality** As mentioned before, the SR300 has two different systems embedded. A depth and/or infrared video frame can be generated with the implemented IR camera and IR projector which are processed by the imaging ASIC<sup>5</sup>. The IR projector illuminates the scene with a set of vertical bar patterns. The depth information can be computed with the data from the IR camera. It captures the reflected patterns which are warped by the scene.

The color camera on the other side, is made up of a chromatic sensor (for the color information of each pixel) and an image signal processor (to process these information). Color video frames for the client can be provided also with the transmission to the imaging ASIC. It is possible to use both imaging systems together or independently which are connected via USB3.

---

<sup>5</sup>Application Specific Integrated Circuit

### 2.3.3 Kinect

Kinect is one of the most used low price 3D camera in the market. It was launched a few years ago first for the Xbox for gaming purpose. Nowadays it is widely used in research and has a integration for Windows as well.

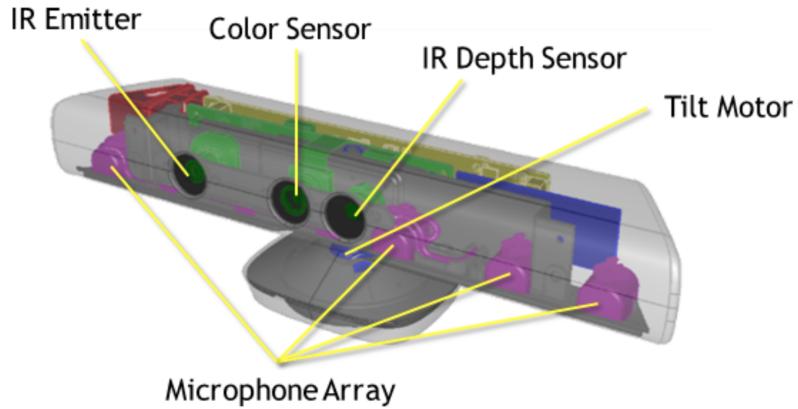


Figure 8: Kinect camera model (Network 2017)

The setup and components of the Kinect camera are similar to Intel RealSense (see figure 8). The newest Kinect has a resolution of 1280 x 960 for the color camera. There are differences in the field of view of Kinect in comparison to the Intel RealSense. The vertical viewing angle is with  $43^\circ$  bigger but the horizontal is with  $57^\circ$  distinctly more constrained than the Intel RealSense. A nice feature, which is integrated into the Intel RealSense as well (and which is not important for this thesis but should be mentioned anyway) is the integrated multi-array microphone which contains four microphones for capturing sound. This makes it moreover possible to find the location of the sound source and the direction of the audio wave which can be an interesting feature for many research areas. Another special function is a 3-axis accelerometer to determine the current orientation of the device. The resolution an the depth stream depends on the frame rate.

## 2.4 Filtering Problems

### 2.4.1 Introduction

Filtering problems are always of the same nature: There are noisy measurements and the task is to find the estimate for the variables, which satisfies a given functional model best. Simultaneously the uncertainty of these variables, the variance, has to be propagated from the noise of the measurements. Or in other words the task is to find the a posteriori probability density function (pdf) of the desired variables using all of the given information. When we have the pdf, we can calculate the state variables and their variances. This is called a Bayesian estimation. This filters can also be time-variant, which means that this estimation has to be done while the system changes over time. Often this algorithms are recursive, which means they only use the last state to determine the new adjustment. Such filters are called Bayes Filters.

Proposed by R.E. Kalman 1960, the Kalman Filter is now widely used as a powerful tool to solve such linear filtering problems. The Kalman Filter is a implementation of a Bayes Filter, which is based on a deterministic estimation technique like the Gauss-Markov Estimation. One of its advantages is that it can estimate the current and even a predicted state of a system without having knowledge of the processes inside the system. We chose to use this filter for this thesis. In this section we are going to outline related models and methods for filtering and thus also explain how the Kalman Filter works.

As already mentioned, the Kalman Filter can be derived from the Gauss-Markov Model for parameter estimation (see figure 9). The maximum likelihood estimation is achieved by minimizing the squares of the residuals, also called Least Squares Method. The recursive version of it is called Sequential Gauss-Markov or Static Kalman Filter. This model enables to add new data into an existing adjustment without computing the whole parameter estimation new. The Kalman Filter is a dynamic, real time version of this Sequential Gauss-Markov Model. In comparison to the Static Kalman Filter, the new estimate is a weighted mean of the impact of the new data and a prediction of the state of the system. So the Kalman Filter is actually a dynamic fitting of a function to new data which is obtained at every time step. Because of the adjustment of the data it is at the same time a process of noise filtering.

The Kalman Filter works only for linear models. When a system has non-linear processes, the Extended Kalman Filter (EKF) should be used. The matrices of the functional models are replaced by the Jacobi-Matrices. It is like an extension for the Kalman Filter, which means that the most formulas also apply in the EKF (G. B. Greg Welch 2006).

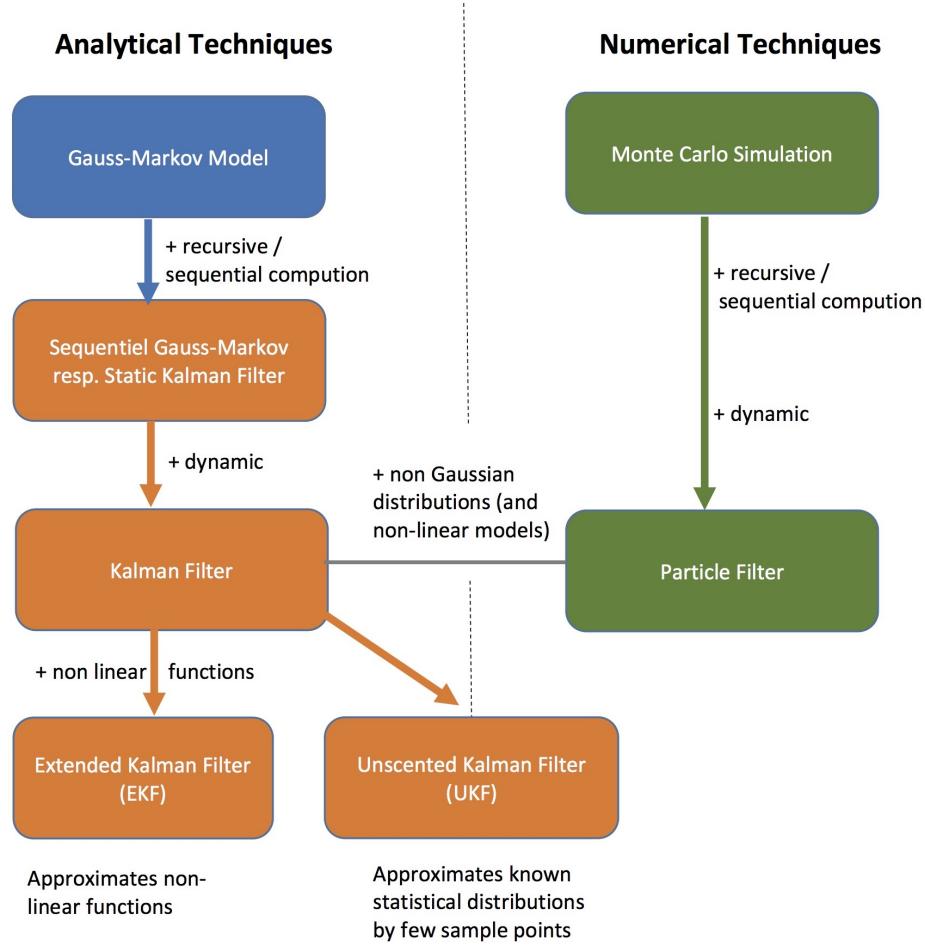


Figure 9: Overview of the different filters and the connection to the Gauss Markov Model

When it comes to filtering problems, there is a whole different section of techniques, the numeric methods, which could be used. The methods prior explained are analytic techniques where one expects always the same output for one input into the functional model. But since there are many random factors involved when tracking a hand, looking at the random sampling methods makes sense.

There the filtering problem is normally solved by generating a big number of random realizations of the model and calculating the most probable solution. These practices are also called Monte - Carlo Simulations. Like in the analytic part, there also exists an implementation of a Bayes filter which is called Particle Filter. It is the stochastic counterpart to the Kalman Filter.

When the system is highly non-linear, the linearization in the EKF can be a bad approximation and can lead to bad performance. That's why another non-linear version of the Kalman Filter, the Unscented Kalman Filter (UKF), was introduced by Simon J. Julier (1997). This filter works with the underlying distribution instead of linearizing the model. This is similar to the general idea of the Monte-Carlo method. This is why, in figure 9, the UKF is placed between the analytic and stochastic part.

### 2.4.2 Bayes Filters

The following section is based on the publication by David Salmond (2005).

A Bayes Filter, or also called Recursive Bayesian Estimation is a general approach to filtering problems. Considering a state  $x$  and observations  $z$ . As described in section 2.4.1 the goal is to find the a posteriori pdf  $p(x_n|Z_n)$  of the state  $x$  using all information of  $z$ . The core idea is now to divide the filtering process into two substeps:

- Predict (Time Update): Calculation of the a priori state  $p(x_n|Z_{n-1})$
- Correct (Measurement Update): Calculation of the a posteriori state  $p(x_n|Z_n)$

This means that first, the a priori solution  $p(x_n|Z_{n-1})$  is calculated which is a prediction of  $x_n$  based on the previous data  $Z_{n-1}$ . In the second step the prediction gets corrected with the new measurements  $z_n$ . This is then called the a posteriori state  $p(x_n|Z_n)$ .

Note:  $Z_n$  denotes the set of all observations up to  $n$ :  $Z_n = \{z_1, z_2, \dots, z_n\}$ .

The first step, the prediction, can be described by the Chapman-Kolmogorov equation:

$$\underbrace{p(x_n|Z_{n-1})}_{\text{A priori state}} = \int \underbrace{p(x_n|x_{n-1})}_{\text{Prediction model}} \cdot \underbrace{p(x_{n-1}|Z_{n-1})}_{\text{Last a posteriori state}} dx_{n-1} \quad (28)$$

So the prediction is calculated from the last a posteriori state with a model for the dynamics of the system, the prediction model. As we will see in section 2.4.5, this is later called prediction matrix  $A$ .

For the second step, the correction, we need the Bayes Rule, which allow us to calculate the inverse conditional probability for two events  $A$  and  $B$ .

$$p(A|B) = \frac{p(B|A) \cdot p(A)}{p(B)} \quad (29)$$

This equation allows us to calculate the inverse conditional probability. The correction is now an application of this rule:

$$\underbrace{p(x_n|Z_n)}_{\text{A posteriori state}} = \underbrace{p(z_n|x_n)}_{\text{Conversion model}} \cdot \underbrace{p(x_n|Z_{n-1})}_{\text{A priori state}} / \underbrace{p(z_n|Z_{n-1})}_{\text{Normalizing denominator}} \quad (30)$$

The conversion model describes the connection between the state and the measurements. It contains the likelihood of the measurements and serves as kind of a weighting of the particles according to the new inputs.

The Kalman Filter and the Particle Filter are both different implementation of this general idea of a Bayes Filter. But first lets look at a popular parameter estimation algorithm in the following section in order to get the bigger picture.

#### 2.4.3 Gauss-Markov Estimation

The following paragraphs are based on the course material by Guillaume (2016).

The Gauss-Markov Model is a mathematical procedure to estimate parameters to given measurements which are afflicted with errors. The underlying principle is the calculation of the maximum likelihood solution for the unknown parameters. This happens by using Least Squares Adjustment.

For this we first need a functional and stochastic model. The functional model describes the relationship between the parameters  $\mathbf{x}$  and the measurements  $\mathbf{z}$  and is in the form of

$$\mathbf{z} = h(\mathbf{x}). \quad (31)$$

We consider  $\mathbf{z}$  a vector with length  $n$  and  $\mathbf{x}$  a vector with length  $u$ .

Note that this means that every equation can only have one measurement in it, which however can be dependent on several parameters. Also note that in Kalman-Filter-Terms this model is called conversion model. The equation can be linearized and written in matrix notation:

$$\delta\mathbf{z} + \mathbf{v} = H \cdot \delta\mathbf{x} \quad (32)$$

with  $\delta\mathbf{z} = \mathbf{z} - h(\hat{\mathbf{x}})$  and  $\mathbf{v}$  being the residuals.

The stochastic model on the other hand serves to describe the errors and correlations between the parameters, so it models their probabilistic behavior. They are represented by the matrix  $P$ :

$$P = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \dots & \sigma_{xv_z} \\ \sigma_{yx} & \sigma_y^2 & \dots & \sigma_{yv_z} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{v_xz} & \sigma_{v_zy} & \dots & \sigma_{v_z}^2 \end{bmatrix} \quad (33)$$

With the use of the functional model, the Least Squares Adjustment (LSQ) now tries to fit the parameter as close to the observations as possible. An indicator for how good the fit is, are the residuals. They describe the deviation from the estimated parameters from the actual measurements. They have to be as small as possible. One method to achieve that is to minimize the square sum of the residuals:

$$\mathbf{v}^T \cdot K^{-1} \cdot \mathbf{v} \rightarrow \min \quad (34)$$

Estimation problems can be divided into three categories:

- $n = u$ : When there are exactly as many observations as parameters, the problem is solvable with one unique solution. But the problem is that we do not have any control over the reliability because of lacking redundancy.
- $n > u$ : In this case there is more information available than necessary and the system is overdetermined. This means that we have a more reliable solution but it also means that there is no unique solution which fulfills the functional model. That is why we calculate the most likely model with the maximum likelihood method.
- $n < u$ : This system is not determined, there is not sufficient information present. Therefore there have to be added some constraints or other conditions which means that there is a infinite number of solutions, dependent on which constraints are added.

#### 2.4.4 Sequential Gauss-Markov resp. Static Kalman Filter

The first step from the Gauss-Markov Model to the Extended Kalman Filter is to make the estimation a recursive procedure. The reason for that is that, when we want to have a real time application with high frequencies, the computation of the whole adjustment is inconvenient and takes too long. So we need to find a possibility to add some new observations to an existing estimation and find out how this new data changes the previous result.

This is achieved by the Sequential Gauss-Markov Model or also called Static Kalman Filter. The following explanation is taken from the course material by A. Geiger (2015) and from

T. B. Greg Welch C. R. (2001).

The idea is to use the results from the last estimation and calculate how much difference the new observations bring. The last result is represented by the state vector  $x_{old}$  and the variance matrix  $P_{old}$ . The new observations are  $x_{new}$  with their errors stored in  $R_{new}$ . The accuracy of the state with only the new measurements would be (by the law of variance propagation):

$$P_{new} = H^T \cdot R_{new} \cdot H \quad (35)$$

The new solution is now a weighed mean between the old and the new data. Generally said, the weights are formed from the accuracy of the new state  $P_{new}$  in respect to the accuracy of the existing adjustment  $P$ :

$$K = \frac{P_{new}}{P_{old} + P_{new}} \quad (36)$$

The matrix K is called Gain Matrix and is very similar to the Kalman Gain Matrix in the KF (see next section and section 3.4 on page 38).

#### 2.4.5 Kalman Filter and EKF

For this section the publication from Brown and Hwang (1997) and G. B. Greg Welch (2006) were used.

The next step from the static to the normal Kalman Filter is to add dynamic. Up until now the state did not change over time, it was fix. Of course, when we put new measurements into the model, the state variables change but they are always fit to the same desired state. In other words the system was up until now time-invariant:

$$\mathbf{x}_n = \mathbf{x}_{n-1} \quad (37)$$

with  $n$  being the current number of timesteps.

As mentioned before, the Kalman Filter is an implementation of a Bayesian filter, so we want to add a functionality which describes how the state will change over time. This is a function of recursive nature which is called prediction model and delivers an a priori guess of the future state (prediction):

$$\mathbf{x}_{n,apriori} = f(\mathbf{x}_{n-1}) \quad (38)$$

This function can then be represented by a matrix which is called Prediction Matrix A:

$$\mathbf{x}_{n,apriori} = A \cdot (\mathbf{x}_{n-1}) \quad (39)$$

with  $\mathbf{x}_{n,apriori}$  being the predicted state.

The model can also be extended with input parameters. If we have some kind of influence over the system, some ability to change the system with controls, we can add this to our equation. The input variables are represented by the control vector  $u$ :

$$\mathbf{x}_{n,apriori} = f(\mathbf{x}_{n-1}, \mathbf{u}_{n-1}) \quad (40)$$

or in matrix notation:

$$\mathbf{x}_{n,apriori} = A \cdot \mathbf{x}_{n-1} + B \cdot \mathbf{u}_{n-1} \quad (41)$$

The model for the influence of the controls is represented by the control matrix B.

If there was no prediction model  $f$  (which means if we would use the Static Kalman Filter) the filter would assume a fixated, non-varying state and always try to fit the measurements, which of course do not match the model anymore. This would result in a big delay of the estimate to the actual observations and quite bad performance.

However, if we only take the prediction model and omit the functional (or here called conversion model)  $h$ , which integrates the observations to the system, the filter would also not work because the filter would just go in one direction and would not be able to adapt to changes.

This leads us the core idea of the Kalman Filter and to the second step of a Bayes Filter: The new state is calculated as a weighted mean between the prediction and the measurements. The weights depend on the accuracy of the state, the measurements and the whole process. They are stored in the three covariance matrices  $P$ ,  $R$  and  $Q$ . The matrices  $R$  and  $Q$  are normally determined before executing the filter, so we can choose if we want the filter to stay close to the observations by having a small variance in  $R$  or, if they have much noise, perform more filtering and stay closer to the prediction by decreasing the value of the variance in the Q matrix. With this three matrices, the weight matrix  $K$  is calculated, which is also called Kalman Gain Matrix. The weighted mean, the a posteriori estimated guess of the state is calculated like that:

$$\mathbf{x}_{n,apost} = \mathbf{x}_{n,apriori} + K \cdot (\mathbf{z}_n - H \cdot \mathbf{x}_{n,apriori}) \quad (42)$$

Now this only works if all models are linear, because only then a direct transformation from the equation to the matrix can take place. If there are non-linear functions, the EKF comes into play. The main difference between the normal Kalman Filter and the EKF is, that the functions  $f$  and  $h$  are approximated with a first degree Taylor polynomial, which means that the matrices  $A, B$  and  $H$  are the Jacobi-Matrices :

$$A_{i,j} = \frac{\partial f_i}{\partial x_j}, \quad B_{i,j} = \frac{\partial f_i}{\partial u_j}, \quad H_{i,j} = \frac{\partial h_i}{\partial x_j} \quad (43)$$

The premises of the normal Kalman Filter is, that the system is linear and that the pdf of  $x$  is Gaussian. Otherwise the propagation of variance does not work. As pointed out by G. B. Greg Welch (2006) and Simon J. Julier (1997), when we use the EKF for non-linear models, the distribution of  $x$  cannot be assumed Gaussian anymore. This is because the Jacobi-Matrices are only approximations since it is assumed that the second and higher order terms of the Taylor Series can be neglected. This is a big flaw of the EKF and means that it is also only an approximation to the Bayes Filter.

This was a quick overview over the Kalman Filter and EKF, because we used this filter there is a more detailed and more mathematical explanation in section 3.4 on page 38.

#### 2.4.6 Monte-Carlo Simulations

As described in the book by Kalos and Whitlock (2009) and the course material by Guillaume (2016), the Monte-Carlo Methods try a quite different approach to finding the best estimate of a states variables and propagating the variances than the Gauss-Markov Model. First we need a functional model as always. What we also need to know is the distributions of the input variables. Then a big number of random input variables with the given distribution are generated. Every set of input variables is evaluated with the functional model. Like this, many solutions are simulated which we can analyze using empirical determination of the underlying distribution. Since there are random processes involved, the results are subject to the law of chance. This means we never get an exact, unique solution. And with every conduction of the calculation we get different results. This does not sound ideal, but the good thing is that we can calculate how accurate the result is going to be and always get a more precise result with adding more repetitions.

In figure 10 you can see an overview. The diagrams on the left stand for the distributions of the random input variables, here called  $x$ . In this case we have 3 input variables. When the model is evaluated with the random realizations, we get the output  $y$ . This is repeated many times and so we get a time series represented by the diagrams in the middle, here we have 2 outputs. This time series get analyzed by histograms, correlation functions etc.

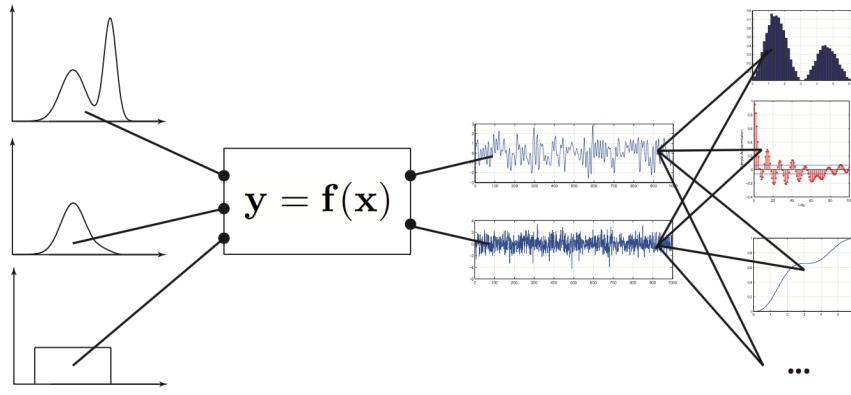


Figure 10: Overview of the concept of the Monte-Carlo Method

One of the main strengths of the Monte Carlo Methods is that the input variables do not have to be Gaussian distributed. They can be arbitrary distributed, the distribution just has to be known.

Another advantage is that any functional model can be used. It does not matter how many inputs and outputs there are or if it is linear or not. The calculations are also very trivial and yield yet to powerful results. This leads us to a big disadvantage of this kind of methods: In order to get an accurate result, we have to perform many calculations for each estimation, which takes up a lot of computing resources.

#### 2.4.7 Particle Filter

The name Particle Filter was first used by Moral (1996) and is another implementation of a Bayesian filter, based on the stochastic principles used for the Monte-Carlo Simulations. This is why it sometimes is also called Sequential Monte-Carlo Method.

The following explanation was taken from the publication by David Salmond (2005) and the course material by D'Andrea (2016). The first step to a Particle Filter is to generate the particles. The particles are random samples drawn from the distribution of the last a posteriori state. When there are enough samples generated, the locations of these particles should represent the underlying distribution. Where many particles are, the pdf takes large values and vice versa.

For the first step, the prediction, the particles get simple propagated through the prediction model, which gives us another set of particles, which represent the a priori, the predicted

state. In the second step, the particles get weighted according to the likelihood of the measurements in respect to the given a priori sample. As a last step we do the inverse of what we did in the beginning, we calculate an empirical distribution from the given samples. Note that the Particle Filter, like all the Monte-Carlo Simulation, is just an approximation of the ideal solution. But due to the large number of samples, the approximation is good enough and we are able to specify how accurate the approximation is.

The main strengths of the Particle Filter are the same as the ones of the Monte-Carlo Method. When the models are linear or mildly non-linear, the Kalman Filter is a good choice to tackle the problem. But if it gets highly non-linear and/or if errors cannot anymore be modeled as Gaussian, the Particle Filter is the better or even the only option to solve it.

#### 2.4.8 Unscented Kalman Filter UKF

As already pointed out in section 2.4.5, when dealing with a non-linear model, the EKF is only an approximation to the Bayes Filter, because the second and higher order terms of the Taylor-Series are neglected when performing the linearization. This can lead to significant errors in the a posteriori variance and respectively in the new distribution. One problem which is often an issue is for example the transformation from polar to Cartesian coordinates.

This is why a new version of the EKF was introduced by Simon J. Julier (1997). The main idea is, instead of approximating a non-linear function, to approximate an unknown Gaussian distribution. First, a number of points is chosen, which are chosen so that they represent the underlying distribution best. They are called sigma points. They are then propagated through the model which gives us a new set of points from which the new distribution is empirically calculated. This method for variance propagation for non-linear functions is called the Unscented Transform. The core idea, namely to propagate samples through the model and calculate the distribution empirically, is the same as in the Particle Filter. But note that there is a fundamental difference, which is that the points are not chosen randomly but are carefully chosen by a deterministic algorithm. This allows us to reduce the number of samples significantly thus reduce the computation expenses.

Although the Unscented Kalman Filter calculates the solutions to a higher order than the Extended Kalman Filter, it is not less computationally efficient. This makes it to an interesting alternative to the EKF, especially since there are no significant downsides as there are for example in the Particle Filter.

## 3 Methodology

### 3.1 General Idea

As explained in the introduction, the goal of the thesis was to use the Intel RealSense camera to implement a safety zone for a robot. The purpose of the safety zone is to protect both the user and the robot by detecting if a person comes to near to the robot. When this happens, the robot should know that and be able to shut down or just stop moving so that no one gets harmed.

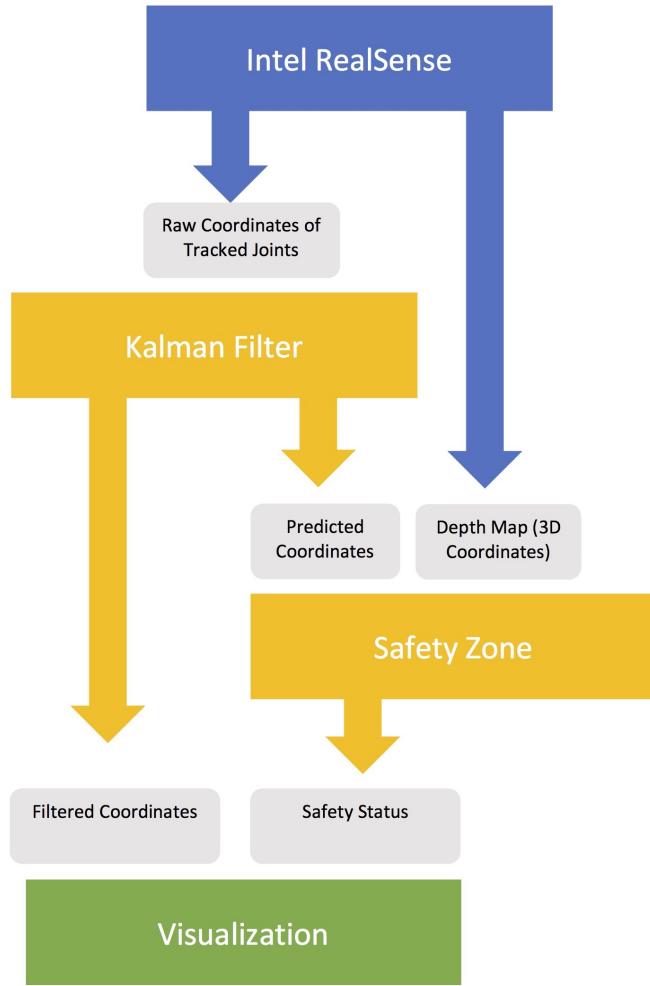


Figure 11: Overview of the main process

This can be realized by using the predicted hand coordinates of the Kalman Filter (KF) and check them against the safety zone. But in order to make it a safe and usable application, the software should also detect anything else apart from hands which comes too near to the robot. This is achieved by using the whole depth map delivered by the Intel RealSense which delivers 3D coordinates for each pixel of the picture.

As shown in figure 11, the proposed work flow is as follows: First the camera delivers the raw hand coordinates and depth data. The hand coordinates are fed to the KF, which returns filtered and predicted coordinates. The filtered coordinates are directly passed on to the visualization. Together with the 3D coordinates of the depth map, the predicted coordinates are used to determine if something has invaded the safety zone. This results in a safety status for each joint and for each pixel, which is handed to the visualization as well, so that the intruded joints can be marked and an invasion can be announced.

Note that the visualization will not be needed in the end application of our implementation since the user will not have to know how and how good his hands are tracked by the robot. The visualization is just to show what our algorithm does and will be important for the development and research phase of the robot, since you get a real time feedback of what is happening behind the scene.

## 3.2 SARI

We labeled all of our own implementation with SARI, which stands for **Safety for Robot Interaction**. These implementations include mainly the build and execution of the KF, the prediction, the setup of the safety plane and the calculation if an object is inside the safety zone. Because the KF and the prediction are such an important part, they are explained in a separate section (3.4).

### 3.2.1 Safety Realization

The main idea is to define a region where no other objects than the robot should be. When anything else enters that zone, a warning should be issued which could tell the robot to stop moving. So if we have any coordinates given, we can check them against the safety region and determine if they are outside or inside, we call that safety status. We do that with the predicted hand coordinates. This makes sense, because since we are dealing with a tabletop robot, the main body parts for interaction are the hands. But as one can imagine, there are many other things which could enter the safety zone like a head, an elbow or also non-human objects. In addition to that the tracking algorithm does not always recognize the hand, particularly when the hand is a fist or is oriented perpendicular to the camera. This is why we added an additional module. The Intel RealSense camera delivers also a depth map, which we use to calculate 3D coordinates for each pixel. We can feed the function, which we already use for the hand, with the depth coordinates and determine the safety status per pixel. This part of the safety implementations does not work with prediction it just makes the application more safe and usable. It is some kind of safety net to make the whole application more reliable.

### 3.2.2 Safety Zone Setup

The safety zone could be represented by many different geometrical primitives. We thought about a cube or sphere around the camera. But since the field of view of one camera is limited to one direction, we decided that a simple plane is sufficient. The side where the camera is, is the inside, the other side we call outside.

One possibility to describe a plane in a 3D space is to define it by three points  $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ . Then the normal vector of the plane can be calculated with the cross product of two difference vectors between points:

$$\mathbf{n} = (\mathbf{p}_3 - \mathbf{p}_1) \times (\mathbf{p}_2 - \mathbf{p}_1) \quad (44)$$

With this vector and one of the plane points, the plane is defined in a more convenient way which we will see in the next paragraph.

**Safety Zone Calculation** To determine whether a point  $\mathbf{p}$  is inside or outside of the safety zone, we have to compare the given coordinates with the safety plane. This can be done using the dot product between the normal vector and the vector from one plane point (for example  $\mathbf{p}_1$ ) to the given point:

$$\text{Safety Status} = \begin{cases} 0, & \text{for } (\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} > 0 \\ 1, & \text{for } (\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} < 0 \end{cases} \quad (45)$$

With 0 meaning outside and 1 meaning inside of the safety zone.

But now we encounter a problem of direction. The normal vector, which the inside and outside, should always look in the same direction, for example to the outside. But dependent on the order and arrangements of the plane points, this vector points to a different side.

We solved this problem by taking a point  $\mathbf{p}_\infty$  which is far away of the origin on the z axis:

$$\mathbf{p}_\infty = \begin{bmatrix} 0 \\ 0 \\ M \end{bmatrix} \quad (46)$$

where  $M$  is a finite big value. Then we calculate the dot product of the position vector of  $\mathbf{p}_\infty$  and  $\mathbf{n}$ . When the dot product is positive, they look into the same direction and we can leave  $n$ , and if it is negative we have to invert the vector  $n$ .

### 3.3 Visualization

As it is mostly the case in Computer Vision projects the calculation and the visualization should be separated. The importance of the visualization part is in Computer Vision even higher than in most Computer Science areas.

As described in section 3.1 and shown in figure 11 our algorithm is divided into three main parts: Getting the data from the Intel RealSense, making the calculation and lastly visualizing it. In section 3.2 the calculation process was explained roughly and in this section we focus on the visualization part in detail.

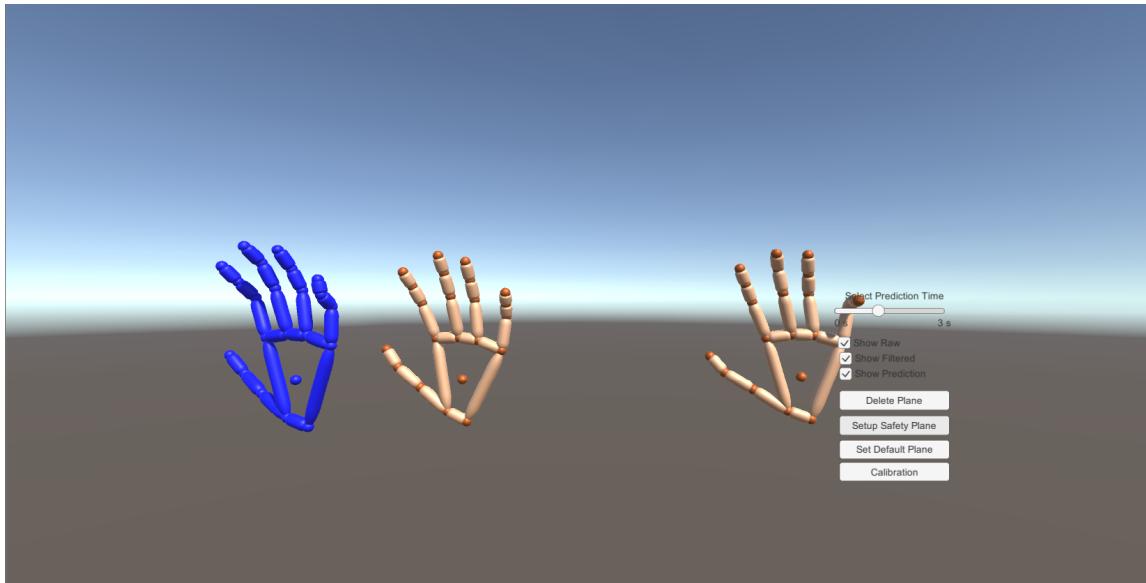


Figure 12: Visualization of the hands, from left to right: Predicted, Filtered, Raw

**Hands** In figure 12 we can see three hands visualized in Unity and except of the color they look similar in appearance. They are made up of spheric representations of the joints and cylindric representations of the bones connecting two joints.

To illustrate a human hand we get information of 22 joints  $\mathbf{x}_j$  from the human hand (see figure 5). We define sphere objects with a constant radius to display each joint and set their position to the current positions. Because it is a sphere, the orientation is not a problem.

It is a different for the bones. There we need some small calculations to set it to the right

position. First of all, we represent all bones with a cylinder. The radius is constant as well but to determine the length  $\mathbf{l}_i$  with  $i = \{1, 2, \dots, 21\}$  of each bone we use the length of the vector between two adjacent joints:

$$\mathbf{l}_i = |\mathbf{x}_{i+1} - \mathbf{x}_i| \quad (47)$$

After we determine the size of each bone we set the bone center  $\mathbf{b}_i$  to the right position. This is exactly the middle of the difference vector between two adjacent joints:

$$\mathbf{b}_i = \mathbf{x}_i + \frac{1}{2} \cdot (\mathbf{x}_{i+1} - \mathbf{x}_i) \quad (48)$$

The last step is to find the right rotation of the bone to connect the both ends with the two joints. For the sake of convenience we set all orientations from the beginning to a reference orientation vector

$$\mathbf{r}_{REF} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad (49)$$

and from this reference orientation we transform each bone to the final orientation vector  $\mathbf{r}_{Bi}$  which can be determined as followed:

$$\mathbf{r}_{Bi} = \frac{\mathbf{x}_{i+1} - \mathbf{x}_i}{|\mathbf{x}_{i+1} - \mathbf{x}_i|} = \frac{\mathbf{x}_{i+1} - \mathbf{x}_i}{\mathbf{l}_i} \quad (50)$$

For the angle transformation we use for each bone a quaternion

$$\mathbf{q}_i = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = \begin{bmatrix} s \\ v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} \cos \frac{\alpha_i}{2} \\ n_{xi} \sin \frac{\alpha_i}{2} \\ n_{yi} \sin \frac{\alpha_i}{2} \\ n_{zi} \sin \frac{\alpha_i}{2} \end{bmatrix} \quad (51)$$

where  $\alpha_i$  is the rotation angle from the scalar product of  $\mathbf{o}_{REF}$  and  $\mathbf{r}_{Bi}$ :

$$\alpha_i = \arccos(\mathbf{r}_{REF} \cdot \mathbf{r}_{Bi}) \quad (52)$$

and  $\mathbf{n}_i$  the rotation axis respectively the cross product of them:

$$\mathbf{n}_i = \mathbf{r}_{REF} \times \mathbf{r}_{Bi} . \quad (53)$$

Before a transformation can be done with any point  $\mathbf{p}$  of the cylinder we must bring it in a quaternion compatible shape  $\mathbf{p}_q$ :

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \mathbf{p}_q = \begin{bmatrix} 0 \\ x \\ y \\ z \end{bmatrix} \quad (54)$$

with this form and the fact that the inversion of the quaternion can be written like this:

$$\mathbf{q}^{-1} = \begin{bmatrix} s \\ -v_x \\ -v_y \\ -v_z \end{bmatrix} \quad (55)$$

We can transform the points of the cylinder

$$\tilde{\mathbf{p}}_q = \mathbf{q}\mathbf{p}_q\mathbf{q}^{-1} = \begin{bmatrix} k \\ \tilde{p}_x \\ \tilde{p}_y \\ \tilde{p}_z \end{bmatrix} \quad (56)$$

and bring it to the format of a 3D point by taking the three last elements.

In our case Unity (the visualization program provided a function called *quaternion()* which takes the two unit vectors  $\mathbf{r}_{REF}$  and  $\mathbf{r}_{Bi}$  as input and the new orientation of a cylinder object is done internally.



Figure 13: Visualization of the safety status

**Safety Zone** The safety zone is represented as a blue transparent plane, so the hand can also be seen when outside of the safety zone. When the predicted hand enters the zone, the respective joints and bone change the color as shown in figure 13. Thus can be verified if the algorithm is working and if the safety is ensured.

**Depth Map** The safety status of the depth map is displayed in the upper left corner as a small image stream as you also can see in figure 13. It appears white when nothing enters the zone and the respective pixels change to red when an intrusion is happening. We can see the shape an approximate position of any objects inside the zone.

## 3.4 Applied Kalman Filter

### 3.4.1 Motivation

One part of this thesis is to filter and predict tracked hand coordinates. A Bayes Filter very suitable for such tasks, because the first step, the prediction, can be used to propagate the movement further into the future. The question that now arises is, which implementation of the Bayes Filter should be used, because there are several different versions available. Four of them were described in section 2.4:

- Normal Kalman Filter (KF)
- Extended Kalman Filter (EKF)
- Unscented Kalman Filter (UKF)
- Particle Filter (PF)

As we will see in the next sections, our filtering problem is linear and has quite a high frequency (30-50 Hz). Since it is linear, it is not necessary to use the Particle Filter, because its advantage is dealing with highly non-linear systems and it has a downside of not being very computationally efficient, which is especially an issue when having a high frequency rate like in our case. It makes no sense to use the Extended Kalman Filter since this is just the extension to the normal Kalman Filter for non-linear models. The Unscented Kalman Filter would have been a possibility to use, since it is superior or equal to the EKF in the most areas. However, for the same reasons as with the EKF, there is no direct need for the UKF. This is why we decided to use the normal Kalman Filter.

### 3.4.2 General Principle

The following subsections are mainly based on the publications by Babb (2016), Czerniak (2017) and Brown and Hwang (1997).

Since the Kalman Filter is a recursive process, the new state of the system ( $x_n$ ) is calculated only from the very last state of the system ( $x_{n-1}$ ). The Kalman Filter is an implementation of a general Bayes Filter, so process is divided into the two substeps prediction and correction.

The first step to build a Kalman Filter, is to determine the variables by which the state of the system should be described. These are the variables which one wants to know, filter or predict. In our case of hand tracking these variables are position and velocity, so we

consider the following state - vector:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix} \quad (57)$$

These variables are assumed to be random and Gaussian distributed. Therefore, in addition to the mean value  $x$ , we consider an uncertainty  $\sigma_x^2$  which is also called the variance. The variables might also be correlated, so there could be a dependency for example between the position and the velocity. All this information is stored in a matrix  $P$  containing the variances on the diagonal and correlations on the other positions:

$$P = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \dots & \sigma_{xv_z} \\ \sigma_{yx} & \sigma_y^2 & \dots & \sigma_{yv_z} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{v_xz} & \sigma_{v_zy} & \dots & \sigma_{v_z}^2 \end{bmatrix} \quad (58)$$

So the whole information we need to know to describe the current state consists of  $x$  and  $P$ . This is also what makes the Kalman Filter very useful for dynamic filtering, it does not take up much memory space because the only data to be stored from  $x$  and  $P$ .

### 3.4.3 Prediction Step

The first step is the prediction. There the Kalman Filter makes a kind of educated guess of how the state of the system is going to be in the near future. For that we need the actual state and the prediction model. The prediction model is an equation which describes the nature of how the state of the system changes over time. It is a recursive function which is of the form:

$$\mathbf{x}_n = f(\mathbf{x}_{n-1}) \quad (59)$$

For hand tracking, the prediction model is a normal equation of motion by Newton:

$$x_n = x_{n-1} + v \cdot \Delta t + \frac{1}{2} \cdot a \cdot \Delta t^2 \quad (60)$$

$$v_n = v_{n-1} + a \cdot \Delta t \quad (61)$$

As proposed by (Kohler 1997), when tracking hands the movement between two time steps is often assumed as linear and the acceleration is omitted and modeled as noise. Because the frequency of the camera is so high, is the time interval between two frames so small which means that the acceleration is very small as well. So the final equations in vector form are

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{n,apriori} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{n-1} + \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}_{n-1} \cdot \Delta t \quad (62)$$

and

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}_{n,apriori} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}_{n-1} \quad (63)$$

Linear models like this can be directly transformed into matrix notation:

$$\mathbf{x}_{n,apriori} = A \cdot \mathbf{x}_{n-1} \quad (64)$$

For the predicted  $P$  matrix we perform a propagation of variance. Since the matrix  $A$  is already calculated, this is an easy task:

$$P_{n,apriori} = A \cdot P_{n-1} \cdot A^T \quad (65)$$

To make the system more general, it is also possible to add a control vector  $\mathbf{u}_{n-1}$  to the equation. This vector describes the external influences on the system, like for example a motor or just any kind of input which adds into the system and changes its state too. Since this functions can also be non-linear we use the same procedure as before and calculate the Jacobi-Matrix which is then called control matrix  $B$ .

In every process there are unknown occurrences which we do not know about. This uncertainties are called process errors and are stored in the  $Q$  matrix. They also describe how good we expect our prediction to be. This information can easily be integrated into the equations, so we finally end up with

$$\boxed{\mathbf{x}_{n,apriori} = A \cdot \mathbf{x}_{n-1} + B \cdot \mathbf{u}_{n-1}} \quad (66)$$

$$\boxed{P_{n,apriori} = A \cdot P_{n-1} \cdot A^T + Q} \quad (67)$$

### 3.4.4 Correction Step

In the second step, this prediction gets refined with new measurements. These measurements can be basically anything which tells us something about the state of the system. For example when the Kalman Filter is used for navigation, which is a common application, the measurements are GNSS coordinates. In our case the measurements are the tracked coordinates of the hand joints and their speed. The measurements are stored in the vector  $\mathbf{z}_n$ . Similar to the process errors, the measured data can also be afflicted with errors and noise. We model these parameters with the matrix  $R$ .

Since these measurements are not necessarily in the same units or scale as the state of the system, we have to transform the predicted (a priori) state from state units to measurement units in order to compare them. For this we use a conversion matrix  $H$ . So the equation for the expected measurements  $\mathbf{z}_{n,apriori}$  and their covariance matrix  $\Sigma_{n,apriori}$  is the following:

$$\mathbf{z}_{n,apriori} = H \cdot \mathbf{x}_n \quad (68)$$

$$\Sigma_{n,apriori} = H \cdot P_n \cdot H^T \quad (69)$$

while the actual measurements are:

$$\mathbf{z}_{meas} = \mathbf{z}_n \quad (70)$$

$$\Sigma_{meas} = R \quad (71)$$

The last important step is the weighted mean of the predicted and the measured state. This makes the filter more smooth and robust. We first calculate the mean in measurement units, knowing that we can always transform to the units of the state with the Matrix  $H$ . A possibility to calculate a weighted mean  $\bar{\mathbf{a}}$  between two vectors  $\mathbf{a}_1$  and  $\mathbf{a}_2$  is to compute the difference vector  $\mathbf{c} = \mathbf{a}_2 - \mathbf{a}_1$  and then add  $\mathbf{c}$  to  $\mathbf{a}_1$  with a factor between 0 and 1 in front of it:

$$\bar{\mathbf{a}} = \mathbf{a}_1 + W \cdot (\mathbf{a}_2 - \mathbf{a}_1) = \mathbf{a}_1 + W \cdot \mathbf{c} \quad (72)$$

with  $W$  being the weight matrix.

Here we do the same. The two vectors are  $\mathbf{z}_{n,apriori}$  and  $\mathbf{z}_{meas}$ . So the weighted mean (in measured units) is:

$$\mathbf{z}_{n,apost} = \mathbf{z}_{n,apriori} + K' \cdot (\mathbf{z}_{meas} - \mathbf{z}_{n,apriori}) \quad (73)$$

We substitute  $\mathbf{z}_{n,apriori}$  with  $H \cdot \mathbf{x}_{n,apriori}$  and  $\mathbf{z}_{meas}$  with  $\mathbf{z}_n$ :

$$H \cdot \mathbf{x}_{n,apriori} = H \cdot \mathbf{x}_{n,apriori} + K' \cdot (\mathbf{z}_n - H \cdot \mathbf{x}_{n,apriori}) \quad (74)$$

The weight matrix  $K'$  is the Kalman Gain Matrix. It is actually a kind of inverse of the ratio between the two covariance matrices  $\Sigma_{n,apriori}$  and  $\Sigma_{measured}$  and is calculated as follows:

$$K' = \frac{\Sigma_{n,apriori}}{\Sigma_{n,apriori} + \Sigma_{measured}} \quad (75)$$

$$= \frac{H \cdot P_n \cdot H^T}{H \cdot P_n \cdot H^T + R} \quad (76)$$

Now we can convert back to the scale and unit of the system state by multiplying  $H^{-1}$  from the left to both sides of the equation which gives us the final result:

$$\mathbf{x}_{n,apost} = \mathbf{x}_{n,apriori} + K \cdot (\mathbf{z}_n - H \cdot \mathbf{x}_{n,apriori}) \quad (77)$$

$$P_{n,apost} = P_{n,apriori} - K \cdot H \cdot P_{n,apriori} \quad (78)$$

Because of the conversion back to state units the Kalman Gain Matrix changes from  $K'$  to  $K$ :

$$K = \frac{P_n \cdot H^T}{H \cdot P_n \cdot H^T + R} \quad (79)$$

The derivation of this formula for  $P_{n,apost}$  is omitted in this report but can be looked up in the publication by Babb (2016).

### 3.4.5 Iterative Calculation

The calculations explained in the last two sections are repeated as long as the filtering should continue. In practice all matrices could change along the process but most of them are assumed constant as proposed in G. B. Greg Welch (2006). The only variables which change are the inputs  $\mathbf{u}_n$  and  $\mathbf{z}_n$  and the outputs which are  $\mathbf{x}_n$  and  $P_n$ . So the matrices  $A, B, H, Q$  and  $R$  can all be determined before the filter is used. In addition to that, approximate values  $x_0$  and  $P_0$  for the outputs need to be defined. This all takes part in the initialization of the filter:

### 3.4.6 Building of the Matrices

For the initialization the matrices are defined and for that the values for the estimated errors need to be determined.

**A Matrix** Since the prediction model for hand tracking is linear as you can see in equation (62) and (63), the functions can be easily transformed into matrix form like this:

$$A = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (80)$$

**B Matrix** When the Kalman Filter is used for just hand tracking, the user has no control over the system so there are no inputs. Therefore it does not matter what the  $B$  Matrix looks like. We just set it to the 6x6 identity:

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (81)$$

**H Matrix** In the case of hand tracking the measured data is already in the same unit and scale since we directly get position and speed from the camera. So the matrix  $H$  is the identity.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (82)$$

**Q Matrix** As proposed by Kohler (1997) and discussed in section 3.4.3, the Q matrix should contain the white noise model for the acceleration. They proposed the following form of the matrix:

$$Q = \frac{a^2 \Delta t}{6} \begin{bmatrix} 2\mathbf{I}(\Delta t)^2, 3\mathbf{I}\Delta t \\ 3\mathbf{I}\Delta t, 6\mathbf{I} \end{bmatrix} \quad (83)$$

$a$  is the acceleration ,  $\Delta t = \frac{1}{60}s$  the time step and  $\mathbf{I}$  is the 3x3 identity matrix.

We set the acceleration  $a = 11\frac{m}{s^2}$  as suggested by Kohler (ibid.).

**R Matrix** The  $R$  Matrix describes the measurement errors and noise. For this we need to know how accurate the camera measures the hand joints. We determine that empirically as a calibration act:

For that one needs to hold the hand 180 time steps  $\Delta t$ , which is approximately 3 seconds, still in front of the camera and during that time the detected coordinates are recorded. Since we know that the hand did not move for that time we can just calculate the standard deviation from the recorded points and get directly a dimension for how good the camera is. The result for one of the 22 joints  $j$  and one observed variable, for example  $x$ , is calculated as a normal empirical standard deviation:

$$\sigma_{x_j} = \sqrt{\frac{\sum_{i=1}^{180} (\bar{x}_j - x_{j,i})^2}{180 - 1}} \quad (84)$$

This is done for each joint and for each of the 6 observed variables, meaning position and velocity. Then the resulting standard deviation for  $x$   $\sigma_x$  is the mean over all joints:

$$\sigma_x = \frac{\sum_{j=1}^2 2\sigma_{x_j}}{22} \quad (85)$$

$R$  itself is a quadratic matrix with these variances on the diagonal. The other matrix items are covariances. We assumed the measurements as uncorrelated as done by Kohler (1997):

$$R = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_y^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_z^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{v_x}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{v_y}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{v_z}^2 \end{bmatrix} \quad (86)$$

The resulting deviation values are normally around 0.1 cm for the position and 0.7  $\frac{\text{cm}}{\text{s}^2}$  for the velocity.

**Approximate values** Since the Kalman Filter is a recursive algorithm we need a first value for  $x$  and  $P$ . These values can be chosen more or less randomly. But it can be that, if they are chosen badly, the algorithm diverges. Especially if  $x$  and  $P$  are both 0, then the state variables will always stay 0, as explained by G. B. Greg Welch (2006, p.12).

Our approximate values are

$$\mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (87)$$

and

$$P_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (88)$$

### 3.4.7 Prediction

The normal prediction step propagates the state variable  $\mathbf{x}$  only one time step  $\Delta t$  into the future.  $\Delta t$  is normally quite small since it should correspond to the rate of the observations, but in order to be able to detect a critical quick hand early enough we need to have a prediction which goes further into the future. This is achieved by iteratively performing the first step, the prediction step  $n$  times:

for i = 1 to n : (89)

$$\mathbf{x}_{i,predicted} = A \cdot \mathbf{x}_{i-1,predicted} + B \cdot \mathbf{u} \quad (90)$$

end (91)

(pseudo code)

Note:  $\mathbf{x}_{0,predicted} = \mathbf{x}_{n,a posteriori}$ .

### 3.5 Implementation

The realization of our algorithm was made using three software components: In order to use the functionalities of the Intel RealSense camera we used the SDK provided by Intel. For the visualization we had the choice between the two game engines Unity and Unreal. We chose Unity, mainly because of better liaison with the Intel RealSense SDK. The coding we did with Visual Studios using C#.

For the implementation our goal was to separate the code parts which interacts with the software components by putting them in different classes. The main three modules are:

- The Intel RealSense camera and the inherent SDK
- Unity, the program for the visualization
- SARI (Safety for Robot Interaction), the functions for our own algorithm.

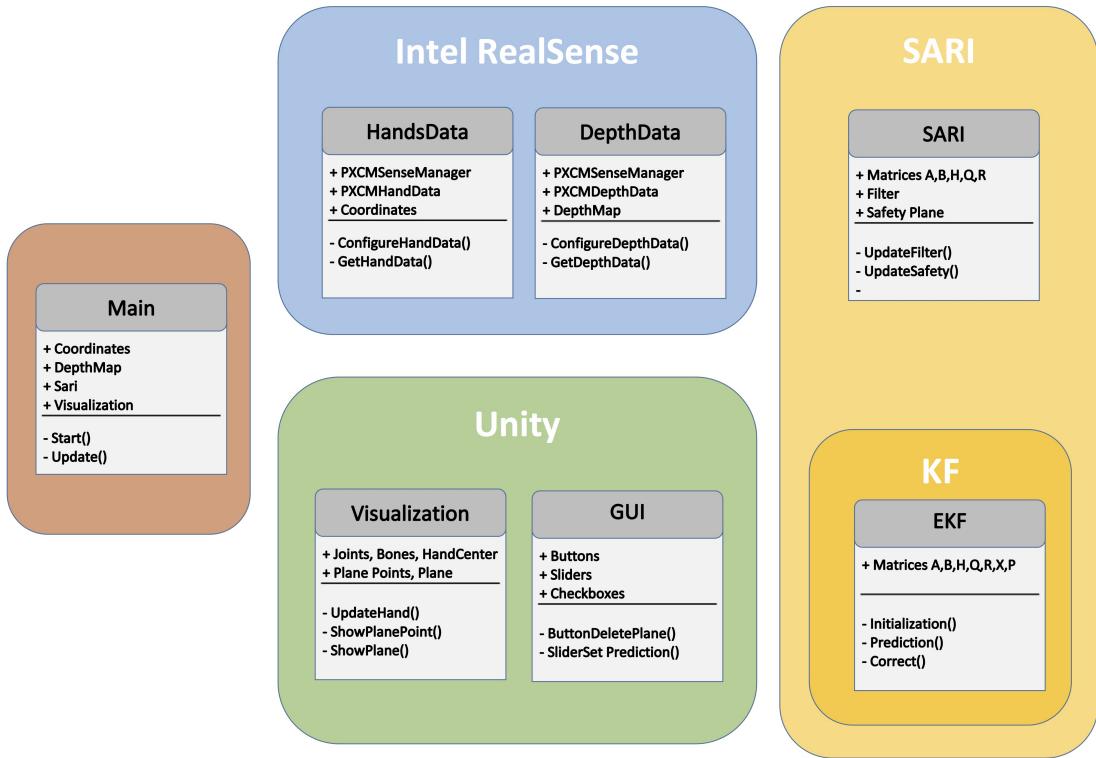


Figure 14: Overview of the code and the different classes

We made different classes for each module. For example everything that has to do with the Intel RealSense, is in the classes *HandData* and *DepthData*. The same is for the Unity part, which is all in the class *Visualization*. All of the SARI code, which is our own contribution, is in its own class and independent from the camera or the visualization. The reason for that is, that it makes it easier for later users of the software to change parts of it. For example if someone wants to use a different program for the visualization, he only has to replace the class *Visualization* and *GUI* and can still use the rest.

There are 7 classes:

**Main** This class is the one which holds all instances of the other classes and is the one which is run by Unity. For this it has to be assigned to a Game Object in Unity. In order that it works, the two functions *Start()* and *Update()* are mandatory. So for every class an instance is generated in the code except for the *Main* and the *GUI* class. The instances of these two classes are made when running the Unity project.

**HandData and Depth Data** Here are all the functionalities stored to get the hand joints coordinates and the depth map out of the Intel RealSense camera. With the function *GetHandData()* the coordinates can be fetched by the *Main* class.

**SARI** This is the class which contains all the features which are associated with the Safety Zone and the filtering of the data, except the KF. Also the functions for checking if any points are inside the safety zone are in this class.

**KF** For the KF there was a separate class made, which is held totally general, so that it could be used for any other application.

**Visualization** Here the variables for Unity are stored, as well as the functionalities to show the hands and the safety plane/status.

**GUI** The functions which are called by GUI elements in the interface of Unity, are stored in a different class. GUI elements are objects like buttons, sliders or checkboxes which allow the user to interact with the programm. These functions call the Unity-instance of the *Main* class and change some variables.

The classes interact with each other and exchange data. As it is with objective programming, to use the functionalities inside of a class, an instance of this class has to be made.

Then the functions and class variables can be accessed over this instance. The instances of the classes *SARI*, *Visualization*, *HandData*, *DepthData* are generated in the *Main* class, the one of *KF* is made in the class *SARI*. This means that data transfer can only happen between two classes where one has an instance of the other. This is why most the data has to go through the main class in order to pass for example from *SARI* to *Visualization*.

This leads to the following implementation of the work flow proposed in figure 11:

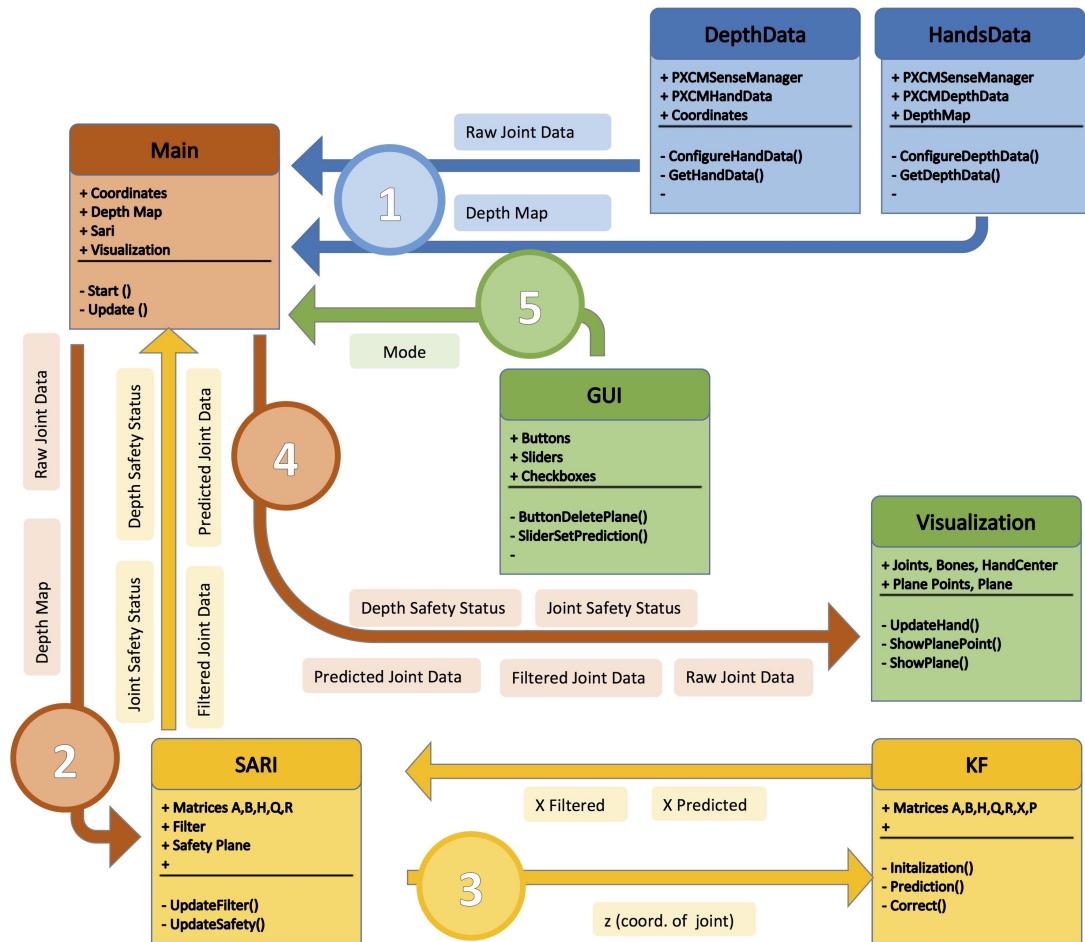


Figure 15: Detailed overview of the code and the different classes and the work flow

1. First the *Main* class fetches the joint coordinates and the depth data from the classes *HandsData* and *DepthData*.

2. This data is then passed on to the *SARI* class. There the coordinates get filtered with the KF joint by joint (observation  $z$ ), so there are actually 22 Kalman Filter working simultaneously.
3. The Kalman Filter uses the observation to update the state variables  $x_{Filtered}$  and generate the predicted state  $X_{Predicted}$ . The prediction is done 20 time steps ahead. This data is gathered together in *SARI* for all joints as filtered and predicted joint data which are given back to the main class. The next step is to calculate if any of the coordinates is behind the safety plane or not. This results in a boolean array of length 22. The same is done for each pixel of the depth map.
4. With this information the visualization can be started. All "three hands" (raw, filtered and predicted) can be displayed, so their coordinates get passed on to the *Visualization* class as well as the safety statuses (as boolean vectors). This data can then be used to change the position of the Game Objects in Unity, the spheres and cylinders which represent the hand, and change their color according to if they are inside or outside of the safety zone.
5. This step is not really the last one, the *GUI* class always intervenes whenever the user changes the GUI controls of the interface. This could be for example the Mode, which is a variable which determines whether the program should run the part with the safety zone or if a new safety zone should be constructed.

## 4 Results and Discussion

The main result of our thesis is actually the roughly 1000 lines of code that we wrote to get that application working. This code can be the ground source for further research in that area or it could even be a starting base to implement a profound safety zone for a real robot. We tried to keep the code clean and easy to read. The modularity of the code, which means the dividing into several classes should also help for the code to be for much further use.

Nevertheless we wanted to get some quantitative results on how our algorithm works but also on how well the camera can track hands and recognize gestures. This is what the next sections are about.

### 4.1 Initialization of the Filter

In the section 3.4.6 the theoretical form and content of the matrices is described. In this section we will explain what values we actually used.

**Timestep  $\Delta t$**  In the  $A$  matrix there is the value  $\Delta t$  which denotes how far the prediction should go, what the difference between two time steps is. Since the prediction should be around where the next measurement should take place, we chose

$$\Delta t = \frac{1}{60} \text{ s} \quad (92)$$

because the frequency rate of the update step is around is around 30-60 Hz. The update interval

**R values** We estimated the values of R as already described with the empirical standard deviation of a still hand. We do that and take the mean over all joints, which usually results in a standard deviation of 0.1 cm for the position and 0.7  $\frac{\text{cm}}{\text{s}}$  for the velocity.

**Q Values** The values of Q should represent the modeling of the acceleration as white noise. This can be achieved as explained by Kohler (1997) in such a form:

$$Q = \frac{a^2 \Delta t}{6} \begin{bmatrix} 2\mathbf{I}(\Delta t)^2, 3\mathbf{I}\Delta t \\ 3\mathbf{I}\Delta t, 6\mathbf{I} \end{bmatrix} \quad (93)$$

and  $a$  around 11  $\frac{\text{cm}}{\text{s}^2}$ . We experienced a much worse behavior of the filter with this setup, the filtered coordinates were very jittery and inexact. For that reason we switched to an

easier model, which is just a diagonal model with a standard deviation of around  $0.045\text{ cm}$  bzw.  $\frac{\text{cm}}{\text{s}}$ . We just found this value to be the best after trying out several others.

## 4.2 Filtering by the Kalman Filter

To assess how good the filtering of the coordinates takes place, we chose a qualitative approach of recording some seconds and plotting the distance from the camera to one joint against the time. This way we can easily see how much noise the raw coordinates contain and what the Kalman Filter makes of this data. Such a plot can be seen in figure 16. The blue line is from the tracked coordinates without any filtering. The red line is the Kalman Filter and we can observe that the noise gets filtered nicely and the Kalman Filter makes the movement much smoother.

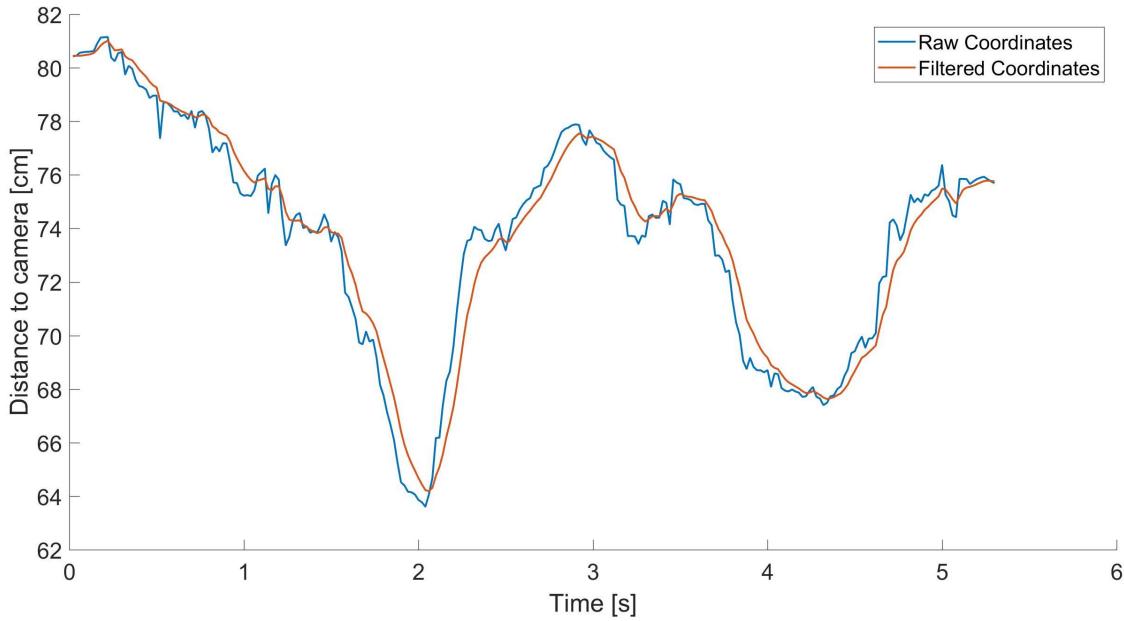


Figure 16: Distance from the camera to the index fingertip over time, raw and filtered coordinates

The amount and type of filtering depends on the values of the error matrices. One example is the R matrix, which contains the estimated accuracy of the observations, the tracked coordinates. In figure 17 we can see how variation of these values affects the filtering process. If the values in the R matrix are small, the filter stays close to the observations, so there is virtually no filtering happening. The reason for that lies in the correction step

of the KF as described in section 3.4. The Kalman Filter makes a weighted mean between the observation and the prediction and chooses the weights according to their estimated accuracy, expressed in standard deviations. When the observed data has a small standard deviation, it gets more weight. As bigger as the values in R get, the more the filter smooths the data and sticks to the predictions.

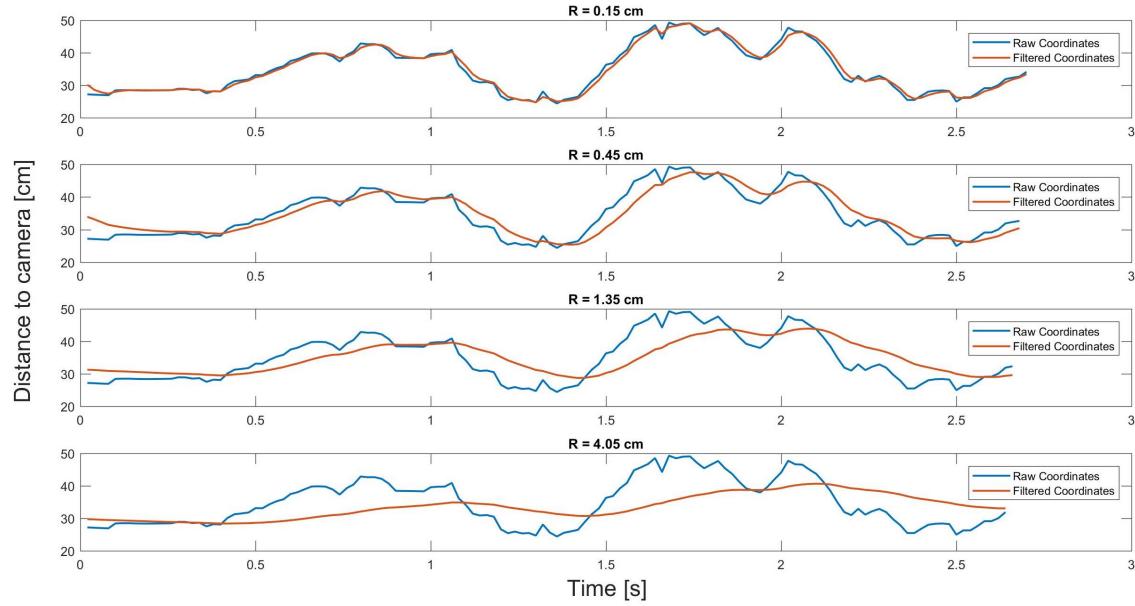


Figure 17: Influence of the R value on the filtering process (Given values are the mean over the matrix)

The same can be done with the Q matrix, which describes the system errors. When we increase the values in the Q matrix, exactly the opposite happens. The Filter assumes a big error of the process and therefore gives the prediction a smaller weight than the observation and the filter moves nearer to the measurements as it can be seen in figure 18.

We had to find a compromise between the values of R and Q. We chose them so that the values are smoothed but not to much, as it can be seen in figure 16.

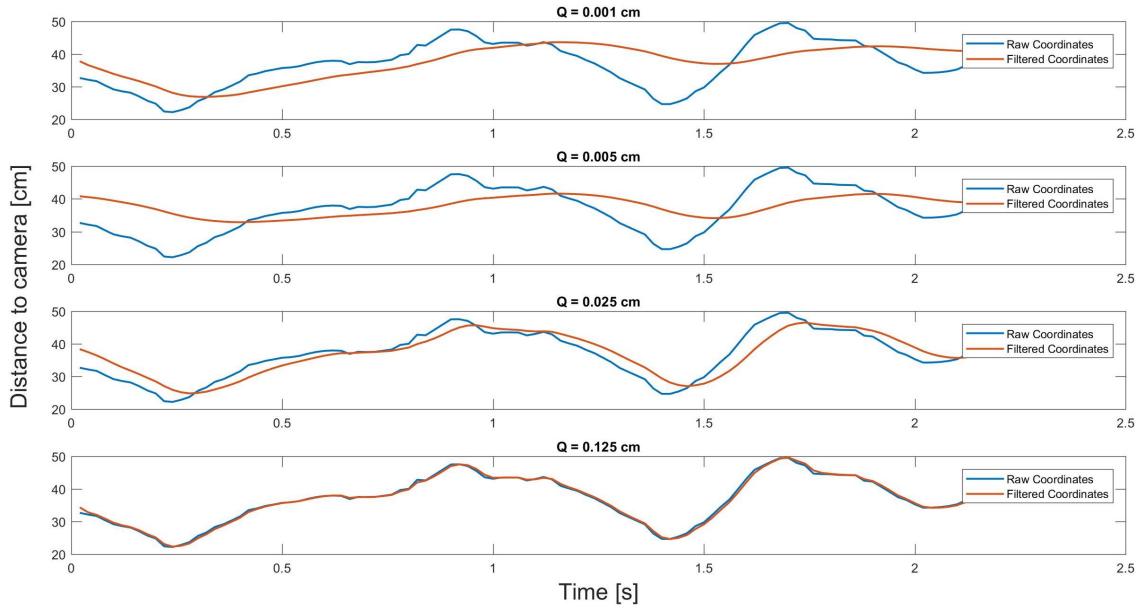


Figure 18: Influence of the  $Q$  value on the filtering process (Given values are the mean over the matrix)

### 4.3 Gesture Recognition

Since we also use gesture recognition for our thesis, we want to analyze how well the gesture recognition of the Intel RealSense works. We focus on 5 gestures of 14 which we thought could be useful. The experiment setup is the following: We perform the specific gesture 20 times in a row and simply count how many times the gesture is recognized correctly. The condition was that the hand has to be already initialized and recognized by the camera. we start with the spreadfinger as initialization of the hand and change to all other gestures in our experiment to count the success rate. For testing the success rate of the spreadfinger gesture we use another gesture as starting point.

We also have to mention that the camera initializes the hand best when we appear from outside in the view of the camera by doing the spreadfinger gesture. Otherwise the camera has difficulties to recognize a hand.

Illustration	Gesture	Recognition[%]	Description
	Spreadfinger	100	Hand open, facing the camera.
	V-Sign	85	Hand closed with index finger and middle finger pointing up.
	Thumb-down	75	Hand closed with thumb pointing down.
	Thumb-up	60	Hand closed with thumb pointing up.
	Fist	30	All fingers folded into a fist. The fist can be in different orientations as long as the palm is in the general direction of the camera.

Table 1: Recognizability of some gestures (Intel 2017)

We can see that the easiest gesture, the spreadfinger, is always detected by the camera. As more complicated the gesture gets (e.g. overlapping of fingers), the worse is the recognition. For example the v-sign is also nearly reliable as well as the thumb-down gesture. Both gestures can be recognized 3 times of 4 or even more. For that reason we use only this three gestures in our implementation. The fist, which is the hardest gesture to identify is just recognized in 30 % of the cases.

## 4.4 Hand Tracking

It is important for our application to know how reliable the camera can recognize a human hand when there is really one. To find this out we focused us for two interactions:

Firstly, grabbing which is a common gesture when playing with a robot (the gesture is similar to spreadinger as in table 1 trying to grab with open hand). Secondly, closing the hand to a fist for usual interactions like pointing with one finger or even trying to punch in which the hand forms a closed posture.

We also differentiated by the speed of the hand movement in 3 parts:

- Hand is holding still
- Hand moves with low speed
- Hand movement with high speed

To simulate the environment as real as possible the hand movements are straight to the camera.

Our simulation shows that for an interaction with an open hand, like grabbing, is even at higher speed possible to detect the hand. Almost all attempts were successful at high speed hand movement.

For closed hand it looks differently. The camera can detect the closed hand without any difficulties when the hand stays still. But when the hand moves with constant but low speed in direction to the camera there is a detection of the hand of almost 70 % and for faster movements the reliability shrinks to 30 % which is not satisfying. This shows again the importance of the depth data extracting from the camera as described in section 3.1 and 3.2.1 when interacting hand could not recognized but with the depth map still a security check is available but without prediction.

## 4.5 Discussion

Even though the depth map provides an overall improvement of the reliability of the whole system, it does not include prediction. Actually the safety zone is now too small for other objects than hands. When they enter the safety zone, it is already too late. This could be solved by using prediction for the pixels of the depth map. Problematic could be the computational time, since it was already an issue to calculate the safety status for the depth map.

This could lead to a redundancy of the tracked hands. But there are still arguments against it. The depth information in general is not that reliable. Dark object are prone to produce more mistakes because the infrared light can get absorbed by the dark surface. Also shiny objects tend to be estimated nearer than they actually are. This is an important criterion which shows that we cannot rely solely on the depth map since a single false negative can cause big damage.

Therefore, for a strongly reliable software, either a better depth map has to be provided or other laminar approaches have to be added.

As mentioned before, we used 22 individual Kalman Filters for each joint, which are working simultaneously. When testing our application we realized that this has an effect on our predicted hand. When there is a fast hand movement, the predicted hand gets distorted, the bones get stretched and it does not look like a normal human hand anymore. The reason is that the Kalman Filters are not connected. To solve this we could use two approaches:

1. Add some constraints for the length of each bone. These constraints could be derived from the actual lengths of the filtered hand.
2. Instead of using several independent filters, we could use one big filter for all joints. This means that the state vector would have length  $22 \times 6 = 132$  elements and all matrices would also increase by this factor.

The reason that we did not implement that is for one that a small filter is much more convenient and at first we thought that this distortion is even desired for our application. But at the end we realized that the constraint approach would have provided a more truthful prediction.

## 5 Conclusion

To sum it all up, we managed to implement a application which not only fulfills the expected goal, but also could be used as a ground base for further development in this area of research. The software can detect two human hands and show them in real-time in form of an animation. These raw coordinates get filtered by the Kalman Filter which enables us to give a smoother visualization of the hands. In the same time we can achieve a prediction of the hand as well while using the prediction model of the Kalman Filter. With these predicted coordinates we can leave the size of the safety zone small while still detecting the dangerous movements in direction of the robot. We filter out the dangerous actions and reduce the false positives (a hand coming near the robot but not is directed to it). To make the system more general and reliably, we added another module which uses the depth data to check the 3D coordinates of every pixel against the safety zone if there is an interruption from an object other than a hand. This part does not work with prediction.

We assessed the filtering process of the Kalman Filter and found out that the smoothing works satisfactorily. The gesture recognition which we also use is satisfactory for easy hand poses but worsens with the complexity like overlapping fingers. For the hand tracking it is more or less the same, open hand poses are excellent even at high speed, but when you make a closed pose similar to a fist, the success rate drops to a marginal performance when doing swift motions.

## 6 Outlook

During the process of making this application, our focus was especially on the possible further uses for adapting it to other environments. Several possibilities to extend our software are thinkable:

As mentioned in the Discussion, one extension could be made to the depth map. The idea is to apply the prediction to the whole depth information and thus ensure an overall safety for any kind of object. Another possibility would be to include face tracking and use the Kalman Filter as well, since the Intel RealSense camera can also track faces.

Although we focused on a tabletop robot with a frontal interaction, our implementation could be adapted for different robots as well when there is a security problem to be solved. To get the most of the camera and its functionalities we could focus us more to the gesture recognition for some direct interactions with the robot for entertaining purpose like playing rock-paper-scissors. An integrated functionality of the RealSense is to detect the current mood of the person (which is deduced from some landmarks on the face). Interactions like reacting to the mood of the person by saying something could be possible.

Another more complex possibility is not only stopping the current movement of the robot arm, but also dodge the movements by moving away the robot arm, since the coordinates of the robot arm and the absolute position of the tracked human hand are known. This would ensure a more lively and real interaction. Finally the problem of a hand manipulation an object could be seized. The RealSense itself has big problems detecting the hand which is partially occluded. However, for a tabletop interaction it is interesting to still get the hand coordinated and also know what object the hand is holding. The placement of such an object on the table could be tracked as well together with the pose, so that the robot can pick it up.

---

## References

### Bibliography

- A. Geiger, M. Meindl (2015). “Sequentielle Ausgleichung”. In: *Course at ETH Zurich: Geoprocessing und Parameterschätzung*.
- Ali Erol, George Bebis et al. (2007). “Vision-based hand pose estimation: A Review”. In: *Science Direct*.
- Babb, Tim (2016). *How a Kalman filter works, in pictures*. <http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/>.
- Brown, R.G. and P.Y.C. Hwang (1997). *Introduction to random signals and applied Kalman filtering: with MATLAB exercises and solutions*. Wiley.
- Colleen C. (Intel) (2015). *Intel RealSense R200 camera*. <https://software.intel.com/en-us/articles/realsense-r200-camera>.
- Czerniak, Greg (2017). *Easy Introduction to Kalman*. [http://greg.czerniak.info/guides/kalm\\_an1/](http://greg.czerniak.info/guides/kalm_an1/).
- D’Andrea, Raffaello (2016). “Recursive Estimation”. In: *Course at ETH Zurich*.
- David Salmond, Neil Gordon (2005). “An introduction to particle filters”. In:
- Fei Yang, Junzhou Huang et al. (2012). “Robust face tracking with a consume depth camera”. In: *Rutgers University, University of Texas at Arlington*.
- Greg Welch, Gary Bishop (2006). “An Introduction to the Kalman Filter”. In: *University of North Carolina at Chapel Hill*.
- Greg Welch Chis Riley, Thomas Bodenheimer (2001). “Kalman Filter Applications”. In: *University of California*.
- Guillaume, Sébastien (2016). “Geodetic Networks and Parameter Estimation”. In: *Course at ETH Zurich*.
- Henning Hamer, Konrad Schindler et al. (2009). “Tracking a Hand Manipulated an Object”. In: *Computer Vision, 2009 IEEE 12th International Conference*.

- 
- Herbert Bay, Andreas Ess et al. (2006). “Speeded-Up Robust Features (SURF)”. In: *Science Direct*.
- Ho, Tim Kam (1995). “Random Decision Forests”. In: *AT and T Bell Laboratories*.
- Intel (2017). *Intel RealSens SDK 2016 R2 Documentation*. Intel RealSense SDK 2016 R2 Documentation.
- Intel SR300 Datasheet (2016). *Intel RealSense SR300 camera*. <https://software.intel.com/sites/default/files/managed/0c/ec/realsense-sr300-product-datasheet-rev-1-0.pdf>.
- J.Wegner (2016). “Mustererkennung”. In: *Course at ETH Zurich: Bildverarbeitung*.
- Kalman, Rudolph Emil (1960). “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D, pp. 35–45.
- Kalos, Malvin H. and Paula A. Whitlock (2009). *Monte Carlo Methods*. Wiley-VCH Verlag GmbH and Co. KGaA.
- Kohler, Markus (1997). “Using the Kalman Filter to track Human Interactive Motion – Modelling and Initialization of the Kalman Filter for Translational Motion”. In: *University Dortmund, Informatics VII*.
- K.Schindler (2016). “Projective Geometry”. In: *Course at ETH Zurich: Photogrammetrie*.
- Li Deng, Dong Yu (2014). “Deep Learning - Methods and Applications”. In:
- MathWorks (2017). *Optical Flow*. <https://ch.mathworks.com/de/discovery/optical-flow.html>.
- Matthew Turk, Alex Pentland (1991). “Eigenfaces for Recognition”. In: *Journal of Cognitive Neuroscience* 3.1.
- Matthieu Bray, Esther Koller-Meier et al. (2004). “3D hand tracking by rapid stochastic gradient descent using a skinning model”. In: *1st European Conference on Visual Media Production (CVMP)*.
- Moral, Pierre del (1996). “Nonlinear Filtering: Interacting Particle Resolution”. In: *Markov Processes and Related Fields* 2.4, pp. 555–580.

- 
- Network, Microsoft Developer (2017). *Kinect Sensor Components and Specifications*. [https://msdn.microsoft.com/de-de/library/jj131033\(d=printer\).aspx](https://msdn.microsoft.com/de-de/library/jj131033(d=printer).aspx).
- Nhuy L. (Intel) (2016). *Comparison of Intel RealSense Camera SR300 and F200*. <https://software.intel.com/en-us/articles/a-comparison-of-intel-realsense-front-facing-camera-sr300-and-f200>.
- P. Zanuttigh, G. Martin et al. (2016). *Time-of-Flight and Structured Light Depth Cameras*. Vol. XII. Springer.
- Simon J. Julier, Jeffrey K. Uhlmann (1997). “A New Extension of the Kalman Filter to Nonlinear Systems”. In: *The University of Oxford*.
- Szeliski, Richard (2010). *Computer Vision: Algorithms and Applications*. Springer.
- Weise, Thomas (2009). *Global Optimization Algorithms - Theory and Application*. it-weise.de.
- Wikipedia (2017). *Range Imaging*. [www.wikipedia.org/wiki/Range\\_imaging](http://www.wikipedia.org/wiki/Range_imaging).
- Zhou Ren, Junsong Yuan et al. (2013). “Robust Part-Based Hand Gesture Recognition Using Kinect Sensor”. In: *Transaction on Multimedia*.

---

## List of Figures

1	Graphic for the two possible solutions . . . . .	1
2	Eigenfaces Initialization . . . . .	10
3	Eigenfaces Recognition . . . . .	10
4	Animation with face tracking . . . . .	13
5	Hand Joints . . . . .	13
6	SR300 camera model . . . . .	17
7	Embedded 3D imaging system . . . . .	18
8	Kinect . . . . .	19
9	Overview of the filters . . . . .	21
10	Overview Monte-Carlo . . . . .	28
11	Overview of the process . . . . .	30
12	Visualization of the hand . . . . .	34
13	Visualization of the safety status . . . . .	37
14	Layout of the classes . . . . .	46
15	Detailed layout of the classes . . . . .	48
16	Filtering by the Kalman Filter . . . . .	51
17	Influence of the R value on the filtering . . . . .	52
18	Influence of the Q value on the filtering . . . . .	53

## List of Tables

1	Recognizability of some gestures (Intel 2017) . . . . .	54
---	---	----

---

## Appendix

- Matlab code for plotting the recorded coordinates
- Matlab code for plotting the recorded coordinates while changing the Q or R values

## Contents

---

- Calculate the means
  - Extract the index finger tip
  - Plot the raw and the filtered data in 3D
  - Plot the raw and the filtered data in 2D space
  - Calculate and plot absolute position
  - Calculate deviation
  - Compare Prediction against Filtered
- 

```
addpath('Normal')

%%Read the data into a 3D array
Raw = dlmread(['Raw0.txt'], ';');
Raw = reshape(Raw', 22, 6, size(Raw, 1)/6);
Raw = permute(Raw, [2, 1, 3]);
%Raw(1, :, :) = Raw(1, :, :) - 40;

Filtered = dlmread(['Filtered0.txt'], ';');
Filtered = reshape(Filtered', 22, 6, size(Filtered, 1)/6);
Filtered = permute(Filtered, [2, 1, 3]);

Predicted = dlmread(['Predicted0.txt'], ';');
Predicted = reshape(Predicted', 22, 6, size(Predicted, 1)/6);
Predicted = permute(Predicted, [2, 1, 3]);
```

---

## Calculate the means

---

```
MeanRaw = squeeze(mean(Raw, 2));
MeanFiltered = squeeze(mean(Filtered, 2));
MeanPredicted = squeeze(mean(Predicted, 2));
```

---

## Extract the index finger tip

---

```
IndexTipRaw = squeeze(Raw(:, 10, :));
IndexTipFiltered = squeeze(Filtered(:, 10, :));
IndexTipPredicted = squeeze(Predicted(:, 10, :));
```

---

## Plot the raw and the filtered data in 3D

---

```
figure() plot3(IndexTipRaw(1,:), IndexTipRaw(2,:), IndexTipRaw(3,:)) hold on
plot3(IndexTipFiltered(1,:), IndexTipFiltered(2,:), IndexTipFiltered(3,:)) hold off
```

---

## Plot the raw and the filtered data in 2D space

---

```
figure()
axis on
```

---

```
subplot(2,2,1);
plot(IndexTipRaw(1,:),IndexTipRaw(2,:))

plot(IndexTipFiltered(1,:),IndexTipFiltered(2,:));
title('xy plane')

subplot(2,2,2);
plot(IndexTipRaw(1,:),IndexTipRaw(3,:))
hold on
plot(IndexTipFiltered(1,:),IndexTipFiltered(3,:));
title('xz plane')

subplot(2,2,3);
plot(IndexTipRaw(2,:),IndexTipRaw(3,:))
hold on
plot(IndexTipFiltered(2,:),IndexTipFiltered(3,:));
title('yz plane')
leg = legend('Raw Measurements','Filtered Measurements')

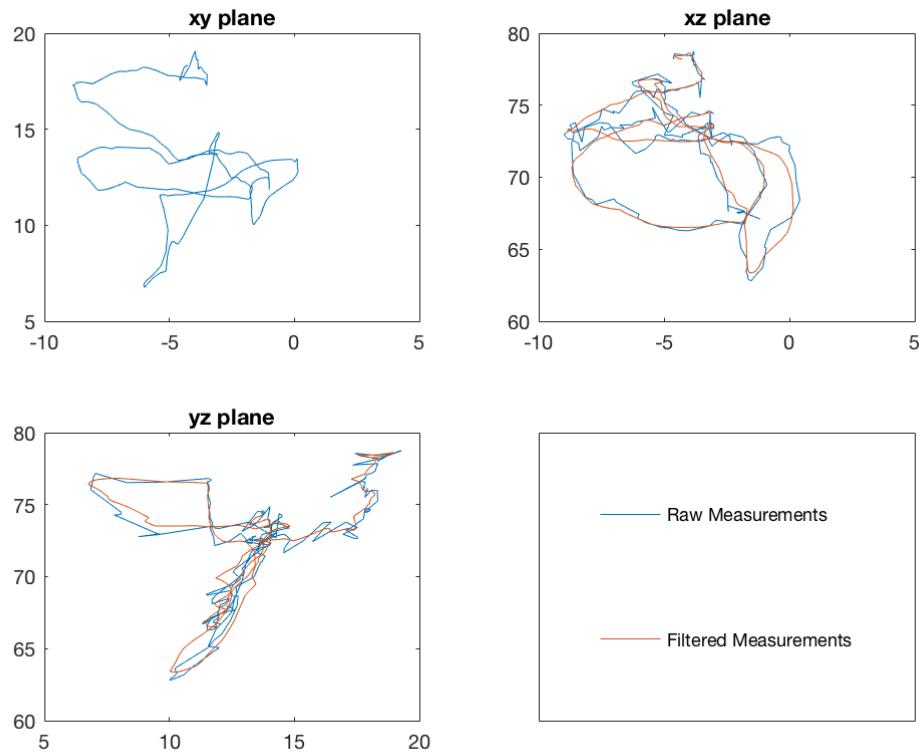
sub = subplot(2,2,4);
axis off
set(leg, 'position',get(sub,'position'))
hold off
```

```
leg =
```

```
Legend (Raw Measurements, Filtered Measurements) with properties:
```

```
    String: {'Raw Measurements'  'Filtered Measurements'}
    Location: 'northeast'
    Orientation: 'vertical'
    FontSize: 9
    Position: [0.2080 0.3643 0.2384 0.0631]
    Units: 'normalized'
```

```
Use GET to show all properties
```



## Calculate and plot absolute position

```

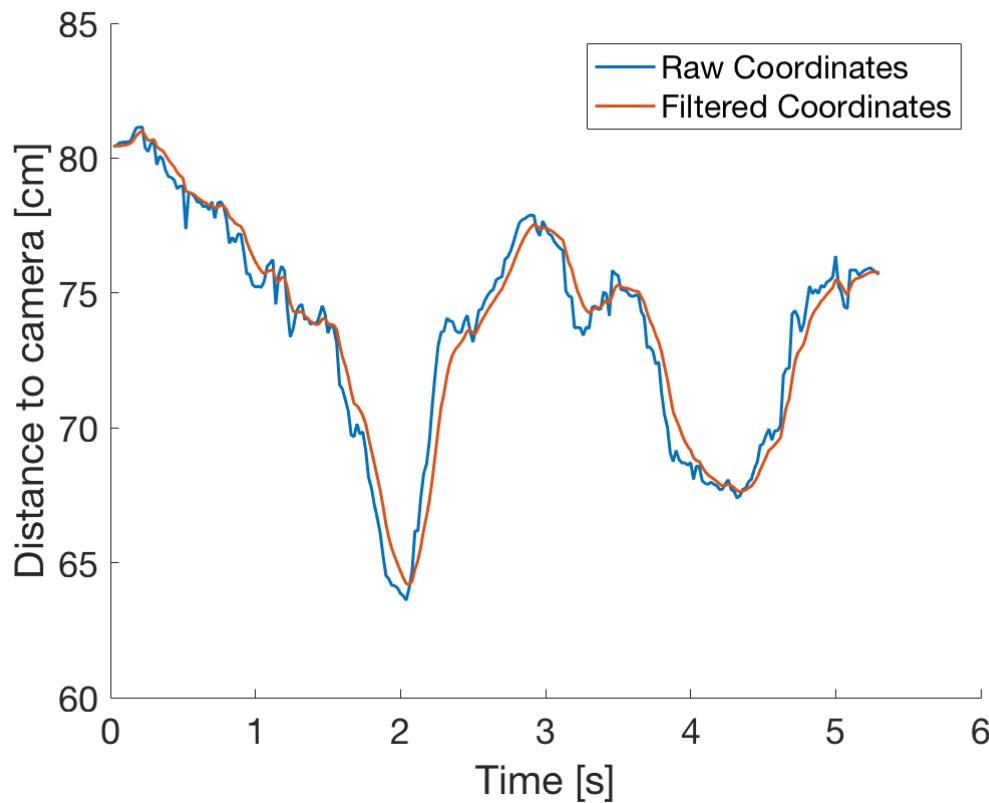
NormRaw = (sum(IndexTipRaw(1:3,:).^2)).^0.5;
NormFiltered = (sum(IndexTipFiltered(1:3,:).^2)).^0.5;
NormPredicted = (sum(IndexTipPredicted(1:3,:).^2)).^0.5;

figure()
set(gca,'fontsize',18)
hold on

plot((1:length(NormRaw))/50,NormRaw,'linewidth',1.5)
% title('Tracking and filtering of the index finger (tip)')
xlabel('Time [s]')
ylabel('Distance to camera [cm]')

plot((1:length(NormFiltered))/50,NormFiltered,'linewidth',1.5)
legend('Raw Coordinates', 'Filtered Coordinates')
% plot(1:length(NormPredicted),NormPredicted)
hold off

```



### Calculate deviation

```
Mean = squeeze(mean(Raw,3));
std1 = zeros(6,22);
for i = 1:size(Raw,3)
    std1 = std1 + (Mean - squeeze(Raw(:,:,i))).^2;
end
std1 = std1./(size(Raw,3)-1);
std1 = std1.^0.5;
std1 = mean(std1,2)

std2 = mean(std(filtered,0,3),2)
```

```
std1 =
```

```
2.2401
2.2232
3.8711
7.0559
7.8777
9.8851
```

```
std2 =
```

```
2.2456
2.2270
```

```
3.8333  
3.7160  
2.7209  
5.6115
```

## Compare Prediction against Filtered

---

```
figure() %plot(1:length(NormRaw),NormRaw) hold on plot(1:length(NormRaw),NormFiltered)  
plot(1:length(NormRaw),NormPredicted) hold off
```

---

*Published with MATLAB® R2017a*

## Contents

---

- Calculate the means
  - Extract the index finger tip
  - Plot the raw and the filtered data in 3D
  - Plot the raw and the filtered data in 2D space
  - Calculate and plot absolute position
  - Calculate deviation
- 

```
addpath('VariantR')
figure()
set(gca,'fontsize',18)
hold on
title('Tracking and filtering of the index finger (tip)')
for count = 1:4
```

---

```
%%Read the data into a 3D array
Raw = dlmread(['Raw',num2str(count), '.txt'], ';');
Raw = reshape(Raw',22,6,size(Raw,1)/6);
Raw = permute(Raw,[2,1,3]);
%Raw(1,:,:,:) = Raw(1,:,:,:) - 40;

Filtered = dlmread(['Filtered', num2str(count), '.txt'], ';');
Filtered = reshape(Filtered',22,6,size(Filtered,1)/6);
Filtered = permute(Filtered,[2,1,3]);

Predicted = dlmread(['Predicted', num2str(count), '.txt'], ';');
Predicted = reshape(Predicted',22,6,size(Predicted,1)/6);
Predicted = permute(Predicted,[2,1,3]);
```

---

## Calculate the means

---

```
MeanRaw = squeeze(mean(Raw,2));
MeanFiltered = squeeze(mean(Filtered,2));
MeanPredicted = squeeze(mean(Predicted,2));
```

---

## Extract the index finger tip

---

```
IndexTipRaw = squeeze(Raw(:,10,:));
IndexTipFiltered = squeeze(Filtered(:,10,:));
IndexTipPredicted = squeeze(Predicted(:,10,:));
```

---

## Plot the raw and the filtered data in 3D

---

```
figure() plot3(IndexTipRaw(1,:),IndexTipRaw(2,:),IndexTipRaw(3,:)) hold on
plot3(IndexTipFiltered(1,:),IndexTipFiltered(2,:),IndexTipFiltered(3,:)) hold off
```

## Plot the raw and the filtered data in 2D space

```
figure() axis on

subplot(2,2,1); plot(IndexTipRaw(1,:),IndexTipRaw(2,:)) hold on plot(IndexTipFiltered(1,:),IndexTipFiltered(2,:)); title('xy
plane')

subplot(2,2,2); plot(IndexTipRaw(1,:),IndexTipRaw(3,:)) hold on plot(IndexTipFiltered(1,:),IndexTipFiltered(3,:)); title('xz
plane')

subplot(2,2,3); plot(IndexTipRaw(2,:),IndexTipRaw(3,:)) hold on plot(IndexTipFiltered(2,:),IndexTipFiltered(3,:)); title('yz
plane') leg = legend('Raw Measurements','Filtered Measurements')

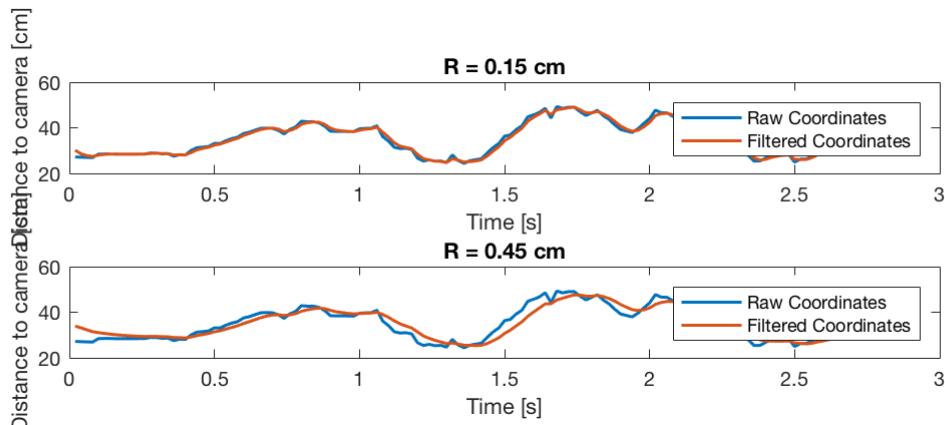
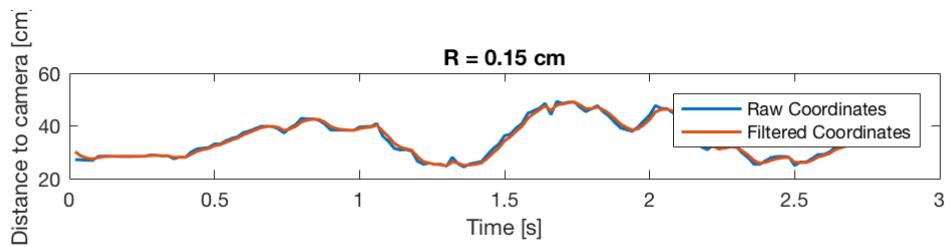
sub = subplot(2,2,4); axis off set(leg, 'position',get(sub,'position')) hold off
```

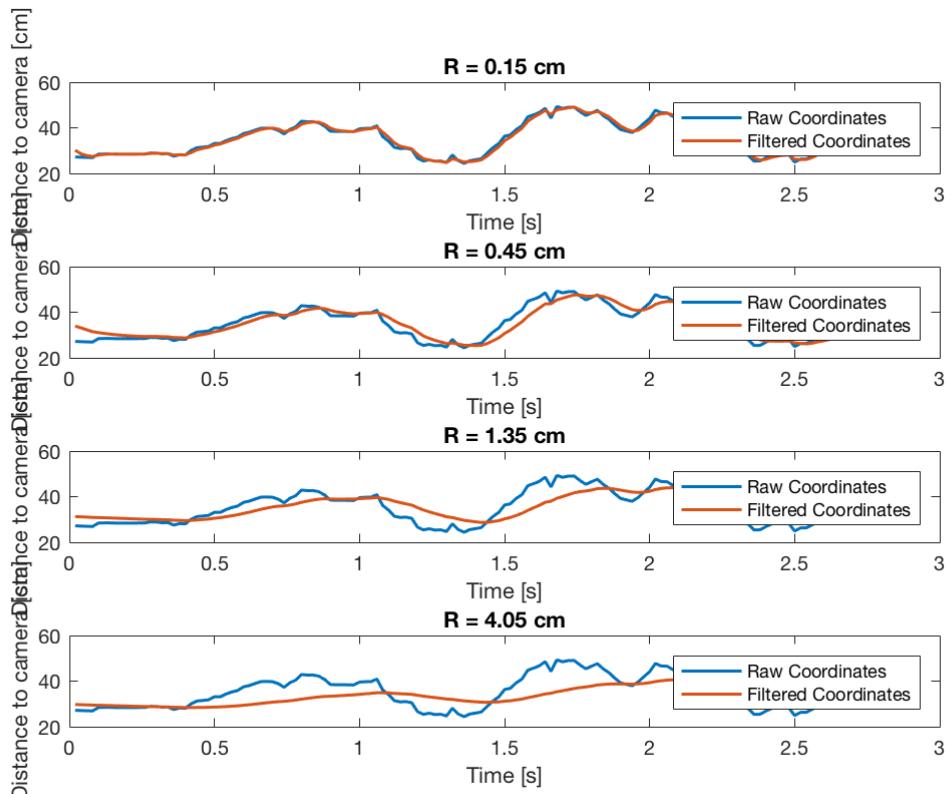
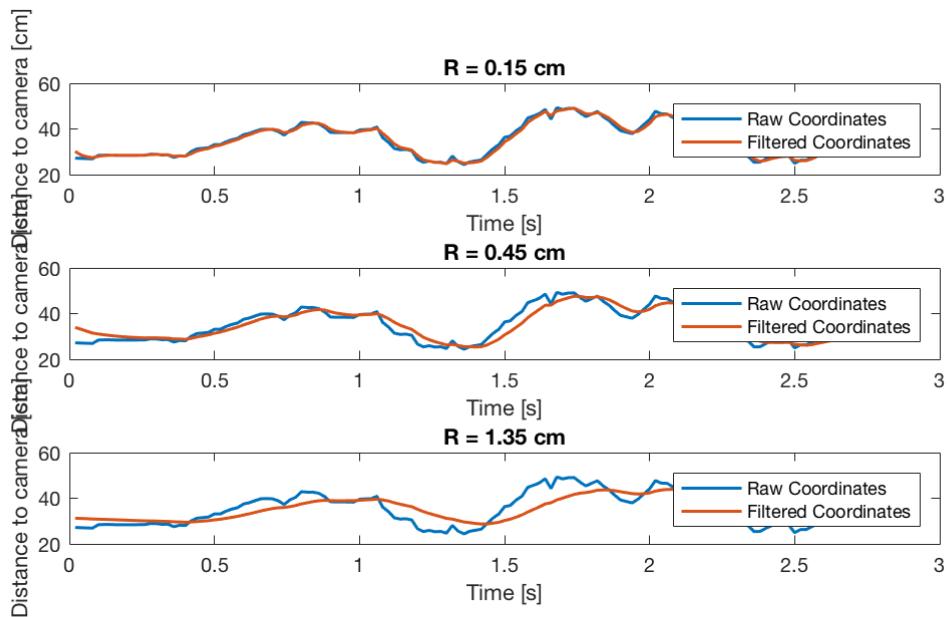
## Calculate and plot absolute position

```
NormRaw = (sum(IndexTipRaw(1:3,:).^2)).^0.5;
NormFiltered = (sum(IndexTipFiltered(1:3,:).^2)).^0.5;
NormPredicted = (sum(IndexTipPredicted(1:3,:).^2)).^0.5;

subplot(4,1,count)
plot((1:length(NormRaw))/50,NormRaw,'linewidth',1.5)
hold on
%title(['Q = ',num2str(0.0002*5^count), ' cm'])
title(['R = ',num2str(0.05*3^count), ' cm'])
xlabel('Time [s]')
ylabel('Distance to camera [cm]')

plot((1:length(NormFiltered))/50,NormFiltered,'linewidth',1.5)
legend('Raw Coordinates', 'Filtered Coordinates')
% plot(1:length(NormPredicted),NormPredicted)
```





## Calculate deviation