TU WIEN Informatics

Tracking Filter Framework

Implementierung und Evaluierung eines Tracking Filter Frameworks für Virtual Reality

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Manuel Gavornik, BSc
Matrikelnummer 00548105

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Mitwirkung: Dipl.-Ing. Dr.techn. Christian Schönauer

Wien, 21. September 2023

Manuel Gavornik	Hannes Kaufmann

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Tracking Filter Framework

Implementation and Evaluation of a Virtual Reality Tracking Filter Framework

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics and Visual Computing

by

Manuel Gavornik, BSc
Registration Number 00548105

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Assistance: Dipl.-Ing. Dr.techn. Christian Schönauer

Vienna, 21st September, 2023

Manuel Gavornik	Hannes Kaufmann

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Manuel Gavornik, BSc

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

Wien, 21. September 2023

Manuel Gavornik

Kurzfassung

Virtuelle Realität stellt eine Computersimulation dar, bei der ein Tracking-System die Haltung des Benutzers erfasst und sensorisches Feedback für einen oder mehrere Sinne ersetzt, um den Benutzer in eine virtuelle Umgebung zu versetzen. Das System muss die Aktionen und Bewegungen des Benutzers genau und schnell erfassen und ihre Darstellungen an die Sinne weitergeben, um den Benutzer in diese Umgebung eintauchen zu lassen. Mess- und Schätzungsfehler sind ein häufiges Problem für solche Systeme, können aber durch den Einsatz von Filteralgorithmen abgeschwächt werden. Diese Arbeit dokumentiert den Entwurf und die Implementierung eines Tracking Filter Framework (TFF) und bewertet seine Fähigkeit Tracking-Fehler zu reduzieren und die Benutzererfahrung zu verbessern. Das TFF wendet Filteralgorithmen auf Tracking-Daten an und liefert das Ergebnis als Ausgabe. Das Lighthouse Tracking System (LHTS) mit Valve's Index wird als Tracking-Datenquelle verwendet, es unterstützt sechs Freiheitsgrade, um die Haltung des Benutzers zu erfassen und nutzt optisches und inertiales Tracking. Die experimentelle Bibliothek libsurvive gewährt Zugriff auf die Inertial-Tracking-Daten. Die Double Exponential Smoothed Prediction (DESP), ein Doppelexponentialfilter, und der Error-state Kalman Filter (ESKF), ein Kalman-Filter mit Fehlerstatus, der optische und inertiale Daten fusionieren kann, werden vorgestellt. Die Benutzerakzeptanz des Systems wird durch die Durchführung einer Benutzerstudie innerhalb einer virtuellen Umgebung mit einem Within-Subject-Design bewertet. Die Ergebnisse zeigen, dass die von libsurvive zur Verfügung gestellten Tracking-Daten mit deaktivierten Optimierungen sich als zu verrauscht und instabil für die vorgestellten Filter erwiesen haben. Die vorgestellten Filter können die auftretenden Tracking-Fehler nicht in dem Maße kompensieren, wie es für eine virtuelle Realitätsanwendung notwendig ist. Die DESP verursacht beim Versuch, verrauschte Tracking-Daten zu filtern, eine erhebliche zeitliche Verzögerung, die für eine virtuelle Realitätsanwendung nicht akzeptabel ist. Der ESKF bietet eine signifikante Verbesserung bei simulierten Tracking-Daten. Bei libsurvive-Trackingdaten ist er jedoch aufgrund der verrauschten und instabilen Daten unzureichend, wie die Ergebnisse der Benutzerstudie zeigen.

Abstract

Virtual reality is a computer simulation wherein a system tracks the user's pose and replaces sensory feedback for one or more senses to place the user into a Virtual environment (VE). In order to immerse users into this environment, the system needs to be accurate and fast in capturing their actions and feed their representations back to the senses. Measurement and pose estimation errors are a common problem for such systems but can be mitigated through the use of filter algorithms. This work documents the design and implementation of a Tracking Filter Framework (TFF), and evaluates its ability to reduce tracking errors and enhance the user experience. The TFF applies filter algorithms to tracking data and provides the result as an output. The Lighthouse Tracking System (LHTS) with Valve's Index is used as tracking data source, it supports six Degrees Of Freedom (DOF) to track the user's pose and uses optical and inertial tracking. The experimental library libsurvive is used to access the inertial tracking data. The Double Exponential Smoothed Prediction (DESP), a double exponential filter, and the Error-state Kalman Filter (ESKF), an error-state Kalman filter capable of fusing optical and inertial data, are introduced. The user acceptance of the framework is evaluated by conducting a user study within a VE, using a within-subject design. The results show that the provided tracking data by libsurvive with disabled optimizations turned out to be too noisy and unstable for the introduced filters. The filters cannot compensate for the occurring tracking errors to the degree that would have been necessary for a Virtual Reality Application (VRA). The DESP causes a significant delay when trying to filter noisy tracking data, which is not acceptable for a VRA. The ESKF provides a significant improvement with simulated tracking data. However, it falls short with libsurvive tracking data because of its noisy and unstable nature, as the results of the user study show.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Virtual reality systems track the position and orientation of real objects in physical space in relation to a reference system. This information is then used to display these objects within a Virtual environment (VE), which allows the user to interact with virtual objects. There are different tracking technologies, and every tracking technology has its advantages and disadvantages. The precision, accuracy, and latency of a tracking system play an important role in fully immersing the user into the experience. In order to provide an enjoyable and immersive experience for the user, an accurate tracking of its pose - namely tracking the user's head, feet and hands - is required. Popular virtual reality solutions for consumers, like Valve's Lighthouse system, use optical and inertial tracking and fuse positioning data to determine the position and orientation of tracked objects. The absolute position and orientation of an object in 3-dimensional space is determined through optical tracking at a specific time interval. The inertial measurements in between these intervals are used to predict the position and orientation by integrating the linear acceleration and angular velocity [28]. Measurement and pose estimation errors impact the user experience. Pose estimation errors can cause unpredictable glitches, residual noise causes jitter, prediction errors can cause drifts, and a latency between movements and their representation causes dizziness or nausea for the user. The goal of this research is to implement and evaluate a filter framework to reduce measurement noise of sensors and improve the pose estimation, in order to provide a better user experience.

1.2 Aim of the Work

The goal of this work is to implement and evaluate a Tracking Filter Framework (TFF), that allows to apply different filters to tracking data, in order to smooth and predict position and orientation. The focus of this work lies on optical and inertial tracking

systems. The framework is implemented as a standalone C++ library and integrated into the Unreal Engine (UE), to use the results within a VE. The processing of tracking data for head and hands is supported. The chosen filters to implement and evaluate are a double exponential filter and a Kalman filter. The Kalman filter allows combining and fusing optical and inertial tracking data. Both filters use quaternions to represent rotations and have the ability to predict the pose in between absolute positioning measurements. Filter parameters used to control an algorithm are configurable. Parameter values are selected by analyzing the filter output and the impact they have on the subjective experience. The performance of the TFF will be evaluated both quantitatively and qualitatively. The results of the filter algorithms are plotted and analyzed qualitatively. Furthermore, the results are tested from the user's perspective within the VE by conducting a user study. Quantitative data is collected by measuring the user's performance and gathering subjective feedback through a questionnaire. The qualitative data is then analyzed to assess the performance of the filter algorithms.

This work addresses the following research questions:

- What types of tracking errors occur for an optical/inertial system?
- Are the selected filter algorithms capable to correct the tracking data?
- Are the selected filters able to improve the user experience?
- Can the filter performance be improved by adjusting its parameters and what are possible unwanted side effects?
- Do tracking errors affect the user's ability to perform tasks?

1.3 Outline

This section outlines the structure of this thesis. Chapter 2 introduces virtual reality, and discusses related tracking technologies and errors. Furthermore, the stochastic fundamentals and related filter algorithms are introduced. Chapter 3 provides an overview of the investigated concepts and design decisions to implement and evaluate a TFF for virtual reality systems. It also describes the selected filter algorithms in detail. Chapter 4 describes the system design and documents the implementation and evaluation approach. Chapter 5 discusses the quantitative and qualitative results, of the performance evaluation and the user study. The last chapter 6 summarizes this work and its findings, as well as addresses issues and shortcomings that are subject for future work.

CHAPTER 2

State of the Art

This chapter introduces the basics of virtual reality and related tracking technologies. Furthermore the fundamentals, which are the basis of the filter algorithms, and the algorithms themselves are introduced.

2.1 Virtual Reality

Virtual reality represents an interactive computer simulation, where the system measures the user's state and interactions and replaces or augments sensory feedback information for one or more senses. This gives the user the feeling of being immersed in the VE or simulation. [33] The VE describes objects, rules and relationships between objects within the simulation [44].

The simulation enables to observe and interact with objects of the VE. The content of this VE can be perceived by the user through vision, hearing and touch. Objects of the VE have properties such as color, texture and shape. These properties can be observed by the user through visual, aural and haptic modalities. [33]

Presence can be described as the feeling of being immersed in the VE and is strengthened by the user through observation and interaction with objects inside the VE. Presence is composed of physical and mental presence. Physical presence is a feeling of being actually in another environment and is created artificially by presenting synthetic stimuli feedback to one or more senses in response to the user's actions. Mental presence is a form of "trance" and describes the degree of engagement with the virtual environment. A high level of mental presence lets the user forget about the real world. [33]

The virtual reality system uses a feedback loop to allow the user to interact with the system through motion. It tracks the user's pose and actions with equipment, information about these actions are incorporated by the system and the changes within the VE are

shown to the user from his perspective. This two way exchange of information enables the user to interact with the virtual reality. [33]

Objects of the VE have positions and orientations within the virtual space. These properties can be influenced by processes within the scene as well as interactions of the user. The position and orientation of tracked objects of the real world - like head, hands and feet of the user - are determined relative to a reference coordinate frame and then translated to the virtual environment. Movement and rotation of objects in the real world and the virtual world are limited by the boundaries defined by the equipment and specifications.

2.2 The Human Factor

A few human factors have to be considered in order to simulate the real world and provide this simulation to the user's senses, as introduced in [27].

The requirement is that the user must be fully immersed into the VE but the solution to achieve this must also be feasible. Figure 2.1 illustrates the contribution of the five human senses according to [18], demonstrating that the vision sense contributes 70% of the information. The quality of the simulation for this sense plays therefore an important

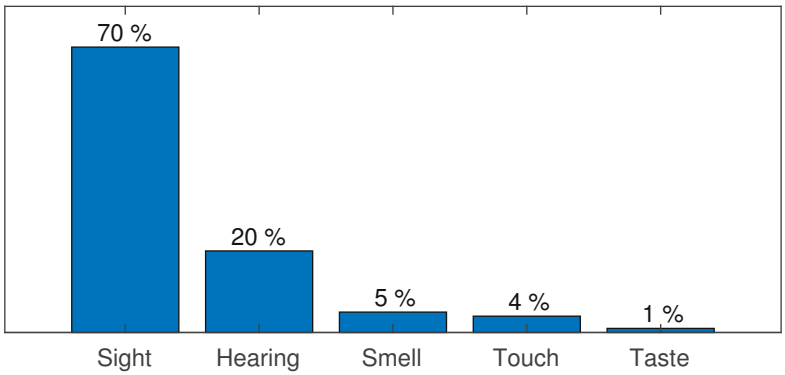


Figure 2.1: Contribution of the five human senses. [18]

role in immersing the user into the experience, and it is the only sense this work focuses on. Two other critical aspects for a system, as mentioned in [41, 42], are the system synchronization and its design. The main cause of the phenomenon called simulator sickness is the synchronization of the stimuli with the user's actions. The design should also address the psychological aspects and is responsible for the depth of immersion in the VE. [27]

2.2.1 Visual Perception

As previously mentioned, visual information is the most critical aspect of the human perception to create an immersion into a VE. As stated in [19], the generated feedback should ideally equal or surpass the limits of the human visual system. However, some compromises have to be considered, as current technologies are not yet capable of achieving this level of performance.

Provided below is a brief summary of critical visual perception characteristics [27]:

Field of view: A human eye has a vertical and horizontal Field of View (FOV) of approximately 180°. The vertical view is limited by the cheeks and eyebrows to about 150°, while the horizontal view is constrained by the nose to around 150°, as noted in [18]. The total FOV for both eyes combined is then approximately 180° with a binocular overlap of 120°.

Visual acuity: The term *visual acuity* describes the sharpness of one's vision. It is measured as the fraction of a pixel, equivalent to one minute of arc or 1/16 of a degree, as explained in [16], and it varies with the arc distance from the line of sight. The fovea, a part of the retina, is capable of resolving the finest details and possesses the highest visual acuity. This area is located approximately two degrees from the line of sight, and sharpness declines rapidly beyond it.

Temporal resolution: The temporal resolution describes the perceived flickering phenomena of screens and is related to the refresh or update rate. This effect can be avoided by using a refresh rate higher than the so called *critical fusion frequency* [16].

Luminance and color: The human eye can differentiate approximately 10 million colors, as noted in [16]. Special color mapping techniques, discussed in [14], must be used for displays with a lower dynamic color range.

Depth perception: The brain is capable of extracting depth information from the state of the eyes and the stereoscopic images provided by both eyes. The depth perception is affected by physiological and psychological aspects, as discussed in [40], and both of these aspects need to be taken into account.

2.2.2 Simulator Sickness

There are potentially many sources responsible for the phenomenon of simulator sickness. However, according to [27], system latency and the frame rate variation play a major contribution to this phenomenon. The studies [26, 37] have identified the most common symptoms that occur in virtual reality systems. These symptoms can be categorized as follows [27]:

- Oculomotor dysfunctions: Eye strain, blurred vision

- Mental dysfunctions: Concentration problems, dizziness
- Physiological dysfunctions: Headache, discomfort, nausea, stomach awareness, vomiting

Here is a brief summary of the two major contributors to the phenomenon of simulator sickness [27]:

Latency and synchronization: The immersion into the VE depends on the quality of the simulation and its resemblance to the real world. This includes image quality as well as the naturalness of the simulation. The response of the system should be fast and accurate. According to [34, 32], one major component responsible for latency is the rendering of the VE. Therefore, the frame rate has the most significant impact on the sense of presence and the efficiency of tasks conducted within the VE [8, 36, 46, 2]. Latencies below 100 ms are sufficient for flight simulators [9], and frame rates of 15 Hz represent the lower limit for establishing a sense of presence within VEs [2]. However, frame rates higher than 60 Hz are preferred [12] and are required to register faster movements [1]. The physiological causes of simulator sickness could be related to the mismatch between visual motion cues and the information received by the vestibular system [19], as humans without a functioning vestibular system are not subject to simulator sickness [13].

Frame rate variations: Users will become disorientated, if the updates of information delivered to the senses do not arrive at the expected time or are delayed. The development of constant frame rate algorithms is required, as discussed in [15], in order to counter the negative impact of non-constant frame rates on the sense of presence and their contribution to simulator sickness.

2.3 Tracking

Tracking allows a seamless interaction and communication between the user and the virtual world [33]. Tracking methods are used to capture the user's movement and actions to create a sense of presence [39].

These methods can be either active or passive. Active tracking methods are triggered by the user with some device, while passive methods track the user's movement and actions. Both methods transmit the tracking information directly to the virtual reality system. [33]

In general tracking systems can be distinguished by delivering absolute data (total position and orientation values) and relative data (change of position and orientation values since the last state). The minimum device requirement for a virtual reality system is a head mounted display to track the position and orientation of the user and to show the rendered images of the VE to the user. Furthermore hands, chest or legs can be

tracked to enable interactions and display the correct state of the user's posture. Tracking 3-dimensional objects requires six Degrees Of Freedom (DOF) to track the position coordinates x,y and z as well as the orientation angles yaw, pitch and roll. [27]

It is required for each tracker to support this data or a subset of it [20]. The object's DOF is defined by the number of independent axes of translation and rotation around it can move [33].

2.3.1 Tracking Requirements and Properties

In order to create an immersive experience for the user inside a virtual world a few goals need to be achieved [47]:

- The user needs to feel presence in the virtual world.
- Objects that are still need to appear as stationary within the simulation, even if the user is moving his head. This is called perceptual stability.
- The user should not experience motion sickness.
- Tracking artifacts should not affect the performance of the task.
- Ideally tracking artifacts should not be recognized by the user.

The quality of a 3-dimensional tracking system is defined by the following properties:

Sampling rate: Specifies how many measurement per second (Hz) will be performed. Higher sampling rates lead to smoother tracking data but require more processing work. [27]

The sampling rate of a tracking system should ideally be running at 1000 Hz or less than 1 millisecond [47].

Latency: The time it takes that the system starts to react to the user's physical action. The latency is usually measured in milliseconds. Lower latencies increase the systems performance and immersiveness. [27]

Accuracy: Describes the measurement errors in absolute values of the system for position and orientation data. The smaller these errors are, the better the systems performance is. Ideally the accuracy should be better then 1 mm for positions and 0.1 degree for orientations. [47]

Resolution: Represents the smallest possible change in position and orientation the system can detect, measured in absolute values. The smaller these values are, the better the systems performance is. [27]

Range: Describes range of the system within the tracked position and orientation data is as accurate as specified [27].

Robust: The system should be resilient against external influences such as temperature, moisture, magnetic field, etc. [33].

Occlusion: The occlusion of sensors is an issue for tracking systems, but a robust system should be able to handle temporarily or partially occlusions [33].

Other properties: Other properties of the system like size, weight, usability, etc. are also important to describe its usefulness and applicability for different applications [27].

2.3.2 Tracking Errors

There are different types of tracking errors which can affect the sense of presence in varying degrees. Tracking errors that affect human perception of the virtual world, can occur while an object is still or moving and result in motion sickness or impact the performance of the task at hand. [47]

Static errors occur if the object is still and can be distinguished as [47]:

Spatial distortion: Reproducible errors at different poses, including sensor scale errors, misalignments, nonlinearity calibration residuals and environmental distortions.

Spatial jitter: Tracker noise that gives the impression that the object is shaking when it is actually still.

Stability or drift: Slow but steady changes in the tracker output that are causing a drift in position or orientation.

Dynamic error occur if the object is moving, and can be distinguished as [47]:

Latency: The mean of the time delay it takes to transmit the corresponding data of a motion.

Latency jitter: A variation in the latency between updates, causing stepping, twitching or spatial jitter along the direction the object is moving.

Dynamic error: Any type of error that can't be accounted for by latency or static inaccuracy. This includes prediction errors from algorithms and sensor errors that are not caused by motion.

Jitter and drift can be considered as representing the high-frequency and low-frequency components of a continuous noise spectrum [47].

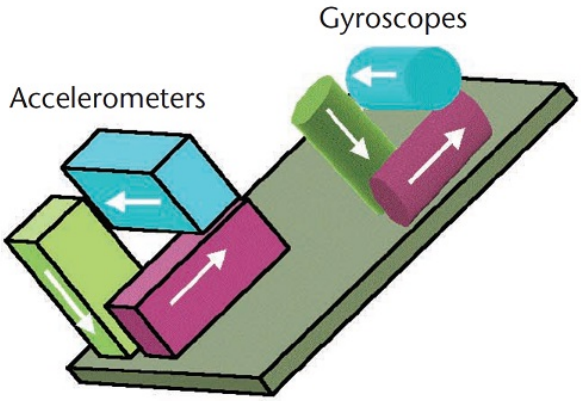


Figure 2.2: Strapdown INS [47]

2.3.3 Tracking Technologies

There are many different tracking algorithms available but relatively few sensing technologies. Available technologies measure electromagnetic fields, electromagnetic waves, acoustic waves or physical forces. Motion tracking system often derive pose estimates from electrical measurements of sensors. There are various types of sensors that can be used in motion tracking systems, including mechanical, inertial, acoustic, magnetic, optical, and radio frequency sensors. Each tracking technology has is advantages and limitations, these limitations arise from the physical medium, the measurement, the signal-processing and the application. [47]

Intertial Principle

The inertial principle combines gyroscopes and accelerometers [33]. A modern system called strap down Intertial Navigation System (INS) is illustrated in figure 2.2, it combines silicon based accelerometers and gyroscopes [47, 33]. The target's orientation is calculated by integrating the measured angular-rates of three orthogonal gyroscopes, which are strapped down to a frame [33]. Three orthogonal linear accelerometers, also strapped to the frame, are used to calculate the position [47]. First the measured acceleration vector is rotated into the navigation frame, using the current orientation determined by the gyroscopes, then gravity compensated and double integrated to estimate the position (see figure 2.3) [47, 33].

Inertial sensors are self-contained and do not require a line of sight to another sensor, and are not sensitive to interfering electromagnetic fields. They have a very low latency of less than a millisecond, provide a high sampling rate of over a thousand samples per second and have a very low jitter. A disadvantage of inertial trackers is drift, the biases of a accelerometers lead to integration errors, which over time stack up and lead to a

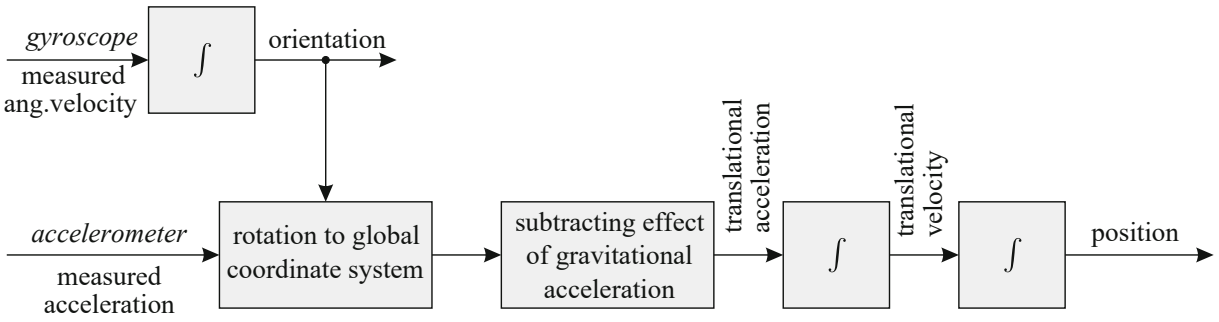


Figure 2.3: Inertial principle algorithm [33]

divergence from the true position. [47]

Optical Principle

Optical tracking systems rely on visual information to track the movement of a user, they use analog or digital sensor devices to measure reflected or emitted light [33, 47]. Active or passive markers in combination with photosensors and light sources can be used to perform these measurements [33].

Analog sensors measure the light intensity or position, while digital sensors provide a discrete image of the scene. Both sensor types can be 1- or 2-dimensional. 2-dimensional sensors provide more information but have lower sampling and processing rates. Lenses and apertures can be used to project images onto the sensor and calculate the angle to the source. The light intensity can be used to estimate the distance to the source. Filters allow to process only certain wavelengths of light, thus infrared light sources combined with filters provide the possibility to operate within the infrared light spectrum, separate from ambient visible light. [47]

A simple analog photosensor changes resistance in relation to the quantity of light reaching it. Photosensors are simple and fast and it is possible to estimate the position with a set of photosensors using relative or ratiometric amplitudes. An analog Position Sensing Detector (PSD) is a semiconductor device, which measures the total light reaching the sensor and produces as set of currents, which can be used to estimate the position of the light source. A Charge Couple Device (CCD) creates a digital image by using a dense array of pixel sensors, that convert light into an electrical charge. The image is created by accumulating the light energy over a short time interval. This process takes time and limits the sampling rate, allowing relatively few measurements per unit of time compared to sensors like the PSD. However a CCD allows to extract pose features like shape, shading or motion from multiple images, but they are computational costly. Variation in lighting, surface properties, independent object motion or occlusion can cause problems for the pose estimation. [47]

The combination of multiple sensors is a common practice to obtain more information

for pose estimation. If the locations of multiple sensors are known, it is possible to estimate the position of a light source with respect to the sensors using triangulation and multibaseline correlation. There are two approaches for an optical system, putting the light sources on the tracked target and the sensors in the environment or the other way around. A system that measures the bearing angles to reference points from the outside to the inside is called outside-in and inside-out when it measures them from the inside to the outside. [47]

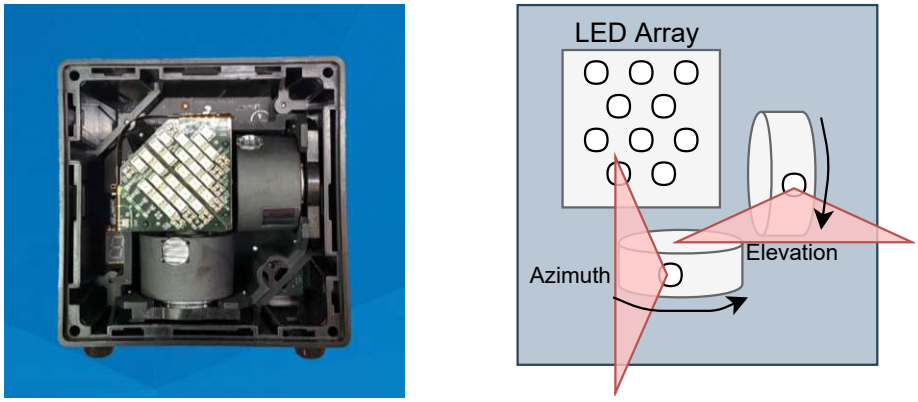
Optical systems need a clear line of sight between the light source and the sensor, which is a disadvantage. Analog photosensors and PSDs have problems with partial occlusions, which can lead to plausible but incorrect results. Image producing sensors are better in recognizing and rejecting partial occlusions but have problems with feature recognition, signal strength and complex error-models that are difficult to predict. The combination of active light sources with analog PSDs allows high spatial precision with high update rates. Systems that use passive image forming devices are slower, but are able to operate in natural environments without a previous setup. [47]

2.3.4 Lighthouse

The Lighthouse Tracking System (LHTS) relies on optical (see chapter 2.3.3) and inertial (see chapter 2.3.3) tracking principals, and can be classified as outside-in tracking system.

The LHTS 1.0 supports up to two base stations, a base station is light sources that transmits infrared light. It consists of two rotating line lenses with mirrors inside that divert a laser generating a fan of light or sweep, and a Light-Emitting Diode (LED) array (see figure 2.4a) [23, 48].

The rotating infrared fans are perpendicular to each other and sweep the environment with an angle of 120° alternately (see figure 2.4b). Tracked devices use photo diodes



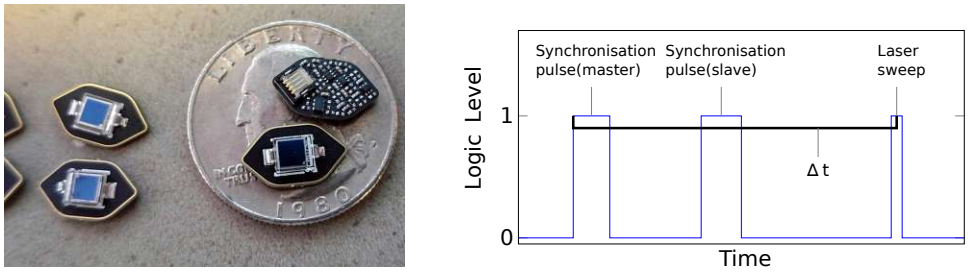
(a) LHTS 1.0 base station[29] (b) LHTS 1.0 base station illustration [23]

Figure 2.4: Lighthouse 1.0

sensitive to the infrared light (see figure 2.5a) to detect the laser sweeps and an Inertial

Measurement Unit (IMU) to track the angular velocity and acceleration. Between laser sweeps the LED array flashes and floods the environment with infrared light. This indicates the start of a laser sweep and is also used to synchronize two base stations with each other. The time difference between the laser flash, marking the start of the sequence, and the time the sweep is hitting each sensor can be measured. [23]

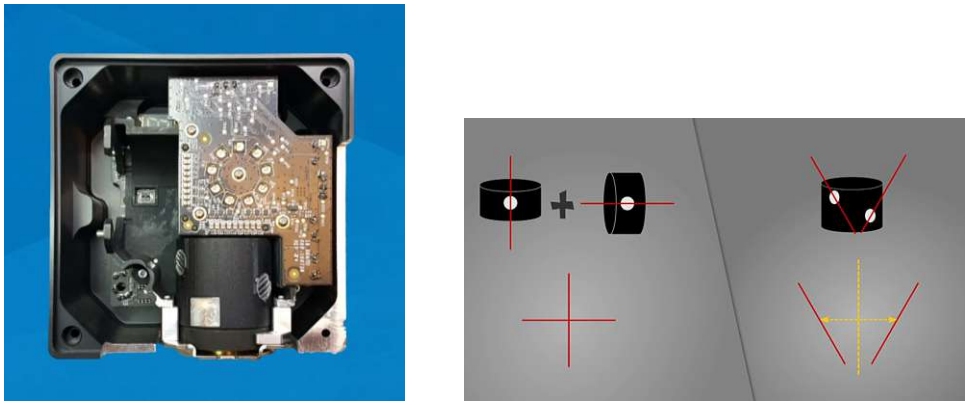
This measurement cycle is illustrated in figure 2.5b, it repeats for each sweep and each base station [47]. Because each rotor is rotating at a constant speed, the time measurement of each sweep can be used to determine the azimuth and elevation angles of a sensor on the receiver hardware in reference to the base station [23, 48].



(a) Photo diode sensor [30] (b) Measurement cycle [29]

Figure 2.5: Lighthouse hardware and measurement cycle

The LHTS 2.0 base stations (see figure 2.6a) use only one rotor with a "V" shaped sweep pattern, instead of two rotors with a perpendicular pattern (see figure 2.6b) [29, 45]. The second generation uses a sync-on-bream signal to eliminate the necessity of the synchronization flash between sweeps [29, 45]. This also enables the support of more than two base stations [45].



(a) LHTS 2.0 base station [29] (b) Sweep pattern [30]

Figure 2.6: Lighthouse 2.0 and sweep patterns

2.3.5 libsurvive

libsurvive is an open source repository on *Github* and provides tools and a library that enables six DOF tracking with LHTSs. It supports the *Lighthouse* generation 1.0 and 2.0 and can run on any device [10].

The library decodes IMU and *lightcap* data signals. The *lightcap* data contains a sensor ID, a light pulse length and the time of the light pulse. IMU data contains the angular velocity and acceleration of a device. The light data together with the sync or sweep pulse is used to calculate the angles to each base station for each sensor. [3]

The library calibrates the devices and base stations, estimates the position and orientation for each tracked device, and provides the estimated pose as well as IMU and *light* data through an Application Programmming Interface (API). The libsurvive library and its API has been used for this work to implement and test the Tracking Filter Framework (TFF) with real tracking data.

2.4 Quaternion

A Quaternion provides a convenient way to represent a rotation, offers better numerical stability than rotation matrices, can be converted to and from rotation matrices, and avoids the issue of a gimbal lock. It follows its definition and fundamental characteristics as described in [43].

2.4.1 Definition

A quaternion can be derived by the *Cayley-Dickson* construction, constructing the complex numbers

$$A = a + bi,\ C = c + di,\ Q = A + Ci \tag{2.1}$$

and defining $k \triangleq ij$ yields a complex number Q with four dimensions in the hypercomplex number space $\mathbb{H}$,

$$Q = a + bi + cj + dk \in \mathbb{H}, \tag{2.2}$$

as described in [43]. Where $\{a, b, c, d\} \in \mathbb{R}$ are real numbers and $\{i, j, k\}$ imaginary unit numbers so that

$$i^2 = j^2 = k^2 = ijk = -1, \tag{2.3}$$

from which the rules

$$ij = -ji = k,\ \ jk = -kj = i,\ \ ki = -ik = j \tag{2.4}$$

can be derived. Quaternions are a four dimensional extension of complex numbers, a quaternion of unit length $q = e^{u_x i + u_y j + u_z k)\theta/2}$ can be used to encode rotations in 3D space. [43]

2.4.2 Alternative Representation

The notation 2.2 is not always convenient. The algebra 2.3 and 2.4 can be used to represent a quaternion as sum of a scalar and vector,

$$Q = q_w + q_x i + q_y j + q_z k \Leftrightarrow Q = q_w + q_v, \tag{2.5}$$

where q_w is denoted as *real* or *scalar* part and $q_v = q_x i + q_y j + q_z k = (q_x, q_y, q_z)$ as *imaginary* or *vector* part. In this work we represent a quaternion Q as four dimensional vector q,

$$q \triangleq \begin{bmatrix} q_w \\ q_v \end{bmatrix} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix}, \tag{2.6}$$

the vector notation allows the usage of matrix algebra for operations involving quaternions. [43]

2.4.3 Properties

Sum

The sum of a quaternion is defined as

$$p \pm q = \begin{bmatrix} p_w \\ p_v \end{bmatrix} \pm \begin{bmatrix} q_w \\ q_v \end{bmatrix} = \begin{bmatrix} p_w \pm q_w \\ p_v \pm q_v \end{bmatrix} \tag{2.7}$$

and is *commutative* $p + q = q + p$ and *associative* $p + (q + r) = (p + q) + r$ [43].

Product

The product operator of two quaternions is denoted as $\otimes$, the product can be constructed using 2.2 and 2.4

$$p \otimes q = \begin{bmatrix} p_w q_w - p_x q_x - p_y q_y - p_z q_z \\ p_w q_x + p_x q_w + p_y q_z - p_z q_y \\ p_w q_y - p_x q_z + p_y q_w + p_z q_z \\ p_w q_z + p_x q_y - p_y q_x + p_z q_w \end{bmatrix}, \tag{2.8}$$

which can also be represented in scalar and vector parts

$$p \otimes q = \begin{bmatrix} p_w q_w - p_v^T q_v \\ p_w q_v + q_w p_v + p_v \times q_v \end{bmatrix}. \tag{2.9}$$

[43]

In general the quaternion product is *not commutative*

$$p \otimes q \neq q \otimes p, \tag{2.10}$$

except when $p_v \times q_v = 0$, one quaternion is real $p = p_w$ or $q = q_w$, or both parts are parallel $p_v || q_v$. However the quaternion product is *associative*

$$(p \otimes q) \otimes r = p \otimes (q \otimes r), \tag{2.11}$$

and *distributive* over the sum

$$p \otimes (q + r) = p \otimes q + p \otimes r \text{ and } (p + q) \otimes r = p \otimes r + q \otimes r. \tag{2.12}$$

[43]

The product of two quaternions is bi-linear and can also be expressed in form of the two matrix products

$$q_1 \otimes q_2 = [q_1]_L \, q_2 \text{ and } q_1 \otimes q_2 = [q_2]_R \, q_1. \tag{2.13}$$

[43]

The left- and right quaternion-product matrices $[q]_L$ and $[q]_R$ can be derived from 2.8 and 2.13 [43]:

$$[q]_L = \begin{bmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & -q_z & q_y \\ q_y & q_z & q_w & -q_x \\ q_z & -q_y & q_x & q_w \end{bmatrix} = q_w I + \begin{bmatrix} 0 & -q_v^T \\ q_v & [q_v]_\times \end{bmatrix}, \tag{2.14}$$

$$[q]_R = \begin{bmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & q_z & -q_y \\ q_y & -q_z & q_w & q_x \\ q_z & q_y & -q_x & q_w \end{bmatrix} = q_w I + \begin{bmatrix} 0 & -q_v^T \\ q_v & -[q_v]_\times \end{bmatrix} \tag{2.15}$$

The *skew operator* $[\bullet]_x$ produces the skew-symmetric $[a]_\times^T = -[a]_\times$ cross-product matrix

$$[a]_\times \triangleq \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}, \tag{2.16}$$

which is equivalent to the cross product

$$[a]_\times b = a \times b, \;\; \forall a, b \in \mathbb{R}^3, \tag{2.17}$$

as described [43].

Identity

The identity quaternion with respect to the quaternion product is defined as

$$q_1 = 1 = \begin{bmatrix} 1 \\ 0_v \end{bmatrix}, \tag{2.18}$$

so that $q_1 \otimes q = q \otimes q_1 = q$ [43].

Conjugate

The conjugate of a quaternion is gives as

$$q^* \triangleq q_w - q_v = \begin{bmatrix} q_w \\ 0_v \end{bmatrix}, \tag{2.19}$$

and has the following properties [43]:

$$q \otimes q^* = q^* \otimes q = q_w^2 + q_x^2 + q_y^2 + q_z^2 = \begin{bmatrix} q_w^2 + q_x^2 + q_y^2 + q_z^2 \\ 0_v \end{bmatrix} \tag{2.20}$$

$$(p \otimes q)^* = q^* \otimes p^*. \tag{2.21}$$

Norm

The norm of a quaternion is defined by

$$||q|| \triangleq \sqrt{q \otimes q^*} = \sqrt{q^* \otimes q} = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} \in \mathbb{R} \tag{2.22}$$

and has the property

$$||p \otimes q|| = ||q \otimes p|| = ||p||\,||q||, \tag{2.23}$$

as described in [43].

Inverse

A quaternion q times its inverse q^{-1} gives the identity quaternion q_1

$$q \otimes q^{-1} = q^{-1} \otimes q = q_1, \tag{2.24}$$

the computation of the inverse quaternion can be done with

$$q^{-1} = \frac{q^*}{||q||^2}, \tag{2.25}$$

as described in [43].

Unit or Normalized

In case of a unit quaternion $||q|| = 1$ the inverse equals the conjugate [43]:

$$q^{-1} = q^* \tag{2.26}$$

The inverse rotation of a quaternion, that represents a rotation operator or orientation specification, equals its conjugate. A unit quaternion can be represented as

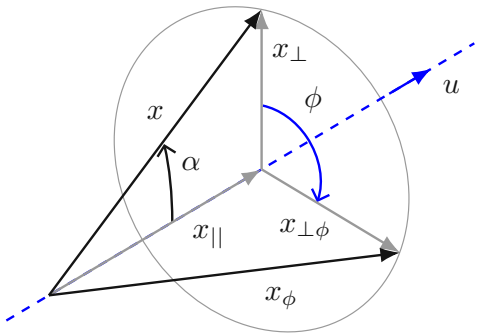
$$q = \begin{bmatrix} \cos\theta \\ u \sin\theta \end{bmatrix} \tag{2.27}$$

where $u = u_x i + u_y j + u_z k$ is a unit vector and θ a scalar. [43]

2.5 Rotation

This chapter is a summary of the introduction into rotations in 3-dimensional space from [43]. It describes the *vector rotation formula* and its relationship to the *rotation action formula*, as well as the the *Special Orthogonal group SO(3)* and its relationship to the rotation matrix and the quaternion. Additionally the exponential map, a tool to deal with continuous change within the rotational space, will be introduced.

2.5.1 The Vector Rotation Formula



$$x = x_{||} + x_\perp$$
$$x_{||} = u\, u^T x$$
$$x_\perp = x - u\, u^T x$$

Figure 2.7: Rotation of vector x, by the angle ϕ, around the axis u. [43]

The figure 2.7 illustrates the right-hand rule rotation of the 3-dimensional vector x, rotated by the angle ϕ, around the axis defined by the unit vector u. The vector x is composed of $x_{||}$ parallel to u, and $x_\perp$ orthogonal to u. It is given by

$$x = x_{||} + x_\perp. \tag{2.28}$$

[43]

The vector $x_{||}$ and $x_{\perp}$ can be calculated with [43]:

$$x_{||} = u(||x||\ cos\ \alpha) = u\ u^T x \tag{2.29}$$
$$x_{\perp} = x - x_{||} = x - u\ u^T x. \tag{2.30}$$

The parallel part $x_{||}$ does not rotate, therefore

$$x_{||\phi} = x_{||}, \tag{2.31}$$

and the orthogonal part $x_{\perp}$ is rotated along the planar plane normal to u. The orthogonal base $\{e_1, e_2\}$ of this plane is given by

$$e_1 = x_{\perp} \tag{2.32}$$
$$e_2 = u \times x_{\perp} = u \times x. \tag{2.33}$$

[43]

If e_1 and e_1 are unit vectors $||e_1|| = ||e_2||$, then $x_{\perp} = e_1 \cdot 1 + e_2 \cdot 0$. A rotation on this plane by ϕ rad can by calculated with

$$x_{\perp\phi} = e_1\ cos\ \phi + e_2\ sin\ \phi = x_{\perp}\ cos\ \phi + (u \times x) sin\ \phi. \tag{2.34}$$

[43]

The rotated vector $x_{\phi} = x_{||\phi} + x_{\perp\phi}$ is then given by the *vector rotation formula* [43]:

$$\boxed{x_{\phi} = x_{||} + x_{\perp}\ cos\ \phi + (u \times x)\ sin\ \phi} \tag{2.35}$$

2.5.2 The Rotation Group SO(3)

A rotation is a linear transformation that preserves distances, angles and relative orientations of a rigid body upon motion. The *Special Orthogonal group SO(3)* is the group of all possible rotations around the origin in 3-dimensional Euclidean space $\mathbb{R}^3$ under the operation of composition that preserve vector length and relative vector orientations. [43]

A rotation operator $r : \mathbb{R}^3 \to \mathbb{R}^3; v \mapsto r(v)$ acting on the vector $v \in \mathbb{R}^3$ that preserves these properties can be defined from metrics of Euclidean space, using the dot and cross product, as follows [43]:

- Vector norm

$$||r(v)|| = \sqrt{\langle r(v), r(v)\rangle} = \sqrt{\langle v, v\rangle} \triangleq ||v||,\ \forall v \in \mathbb{R}^3 \tag{2.36}$$

- Angles between vectors

$$\langle r(v), r(w)\rangle = \langle v, w\rangle = ||v||\ ||w||\ cos\ \alpha,\ \forall v, w \in \mathbb{R}^3 \tag{2.37}$$

- Relative orientation of vectors

$$u \times v = w \Longleftrightarrow r(u) \times r(v) = r(w) \tag{2.38}$$

The first two conditions are equivalent so that the rotation group *SO(3)* can be defined as [43]:

$$SO(3) : \{r : \mathbb{R}^3 \to \mathbb{R}^3 / \ \forall v, w \in \mathbb{R}^3,\ ||r(v)|| = ||v||,\ r(v) \times r(w) = r(v \times w)\} \tag{2.39}$$

Rotations can be represented as rotation matrixes or quaternions, both representations are equally valid. One of the most important differences is that the unit quaternion group constitutes a double cover of *SO(3)*, which is technically not *SO(3)* itself, but is not critical in most applications. [43]

2.5.3 The Rotation Group and its Relationship to the Rotation Matrix

The operator $r()$ is defined by linear scalar and vector products and is therefore also linear. It can be represented by a matrix $R \in \mathbb{R}^{3\times3}$, which rotates vectors $v \in \mathbb{R}^3$ through the matrix product

$$r(v) = Rv. \tag{2.40}$$

[43]

The *orthogonality* condition on R can be derived by injecting the dot product $\langle a, b\rangle = a^T b$ into the vector norm condition 2.36, which results in [43]:

$$\boxed{R^T R = I = RR^T} \tag{2.41}$$

The *Orthogonal group O(3)* represents the set of transformations which preserves vector norms and angles. This group includes rotations, which are rigid, and reflections, which are not rigid. The group implies that the product of two orthogonal matrices is also an orthogonal matrix, and that an inverse matrix for each orthogonal matrix exists. [43]

The orthogonality condition 2.41 implies that each column vector of the rotation matrix is of unit length and orthogonal to each other, and that the inverse of the rotation matrix equals its transposed matrix [43]:

$$R^{-1} = R^T \tag{2.42}$$

Adding the relative orientation condition 2.38 discards reflections and guarantees rigid body motion by adding another constraint on R [43]:

$$\boxed{det(R) = 1} \tag{2.43}$$

Orthogonal matrices with a positive unit determinant are called *special*. This set of *special orthogonal matrices* is called *Special Orthogonal group SO(3)* and is a subset of *O(3)*. The group implies that the product of two rotation matrices is always a rotation matrix. [43]

The Exponential Map exp()

The exponential map is an essential tool for estimation problems in the space of rotations and orientations. It provides a corpus of infinitesimals calculus to work with continuous change within the rotational space and allows to properly define and manipulate derivatives, perturbations and velocities. [43]

Rotations are rigid motions, this allows to define a continuous rotation on a path $r(t)$ in $SO(3)$. A rigid body continuously rotates along this path from its initial orientation $r(0)$, to its current orientation $r(t)$. Looking into the time-derivate of the orthogonality condition 2.41 we get

$$\frac{d}{dt}(R^T R) = R^{T'} R + R^T R' = 0 \Rightarrow R^T R' = -(R^T R')^T, \tag{2.44}$$

where matrix $R^T R'$ is skew-symmetric. The set of skew-symmetric 3×3 matrices is called *Lie algebra* of $SO(3)$ and is denoted as $\mathfrak{so}(3)$. The skew-symmetric 3×3 matrix has been introduced in 2.16, it corresponds to the cross-product matrix (see 2.17) and has 3 DOF. This defines the one-to-one mapping $\omega \in \mathbb{R}^3 \leftrightarrow [\omega]_\times \in \mathfrak{so}(3)$ so we can write

$$R^T R' = [\omega]_\times, \tag{2.45}$$

which gives us

$$\boxed{R' = R[\omega]_\times}, \tag{2.46}$$

denoted as the Ordinary Differential Equation (ODE). [43]

For rotations around the origin $R = I$ applies, which reduces the ODE to $R' = [\omega]_\times$. The Lie algebra $\mathfrak{so}(3)$ describes the space of derivatives of $r(t)$ at the origin, meaning it is the *tangent space* to $SO(3)$ also called *velocity space*. The vector ω represents instantaneous angular velocities. [43]

If the vector ω is constant, the ODE (2.46) can be time-integrated [43]:

$$R(t) = R(0)\, e^{[\omega]_\times t} = R(0)\, e^{[\omega t]_\times} \tag{2.47}$$

The exponential term $e^{[\omega t]_\times}$ is defined by its Taylor series and is also a rotation matrix. Defining the vector $\Phi \triangleq \omega \Delta t$, which describes the full rotation over the time period Δt, we receive

$$\boxed{R = e^{[\Phi]_\times}}. \tag{2.48}$$

[43]

This is the exponential map, which is an application from $\mathfrak{so}(3)$ to $SO(3)$ [43]:

$$\exp : \mathfrak{so}(3) \to SO(3)\ ;\ [\Phi]_\times \mapsto \exp([\Phi]_\times) = e^{[\Phi]_\times} \tag{2.49}$$

The Capitalized Exponential Map Exp()

To avoid possible ambiguities we further define the mapping $\mathbb{R}^3 \mapsto SO(3)$ denoted as capitalized Exp [43]:

$$\text{Exp} : \mathbb{R}^3 \mapsto SO(3) \; ; \; \Phi \mapsto \text{Exp}(\Phi) = e^{[\Phi]_\times} \tag{2.50}$$

Its relationship to the exponential map is defined as [43]:

$$\text{Exp}(\Phi) \triangleq \exp([\Phi]_\times) \tag{2.51}$$

Rotation Matrix and the Rotation Vector

The rotation or angle-axis vector $\Phi = \omega\Delta t = \phi u$ encodes the angle ϕ and axis of rotation vector u. The rotation matrix is be defined by mapping the rotation vector Φ through the exponential map $\exp([\Phi]_\times)$ (2.48), using the the cross-product matrix $[\Phi]_\times = \phi[u]_\times$ (2.17). The Taylor expansion of $R = e^{\phi[u]_\times}$ (see [43]) leads to the rotation matrix through the rotation vector, the *Rodrigues rotation formula*

$$\boxed{R = I + sin\phi[u]_\times + (1 - cos\phi)[u]_\times^2}, \tag{2.52}$$

which is denoted as $R\{\Phi\} \triangleq \text{Exp}(\Phi)$. [43]

The Rotation Action

The rotation of a vector x by the angle ϕ around the unit axis u can be achieved with the linear product

$$\boxed{x_\phi = Rx}, \tag{2.53}$$

where $R = \text{Exp}(\phi u)$. [43]

Note that the validity of the rotation action 2.53 can be proven by deriving the vector rotation formula 2.35 using the *Rodrigues rotation formula* 2.52 (see [43]).

2.5.4 The Rotation Group and its Relationship to the Quaternion

In this chapter the relationship between quaternions and rotation matrices as representations of the rotation group $SO(3)$ are highlighted. The quaternion rotation action formula is given by

$$r(v) = q \otimes v \otimes q^*, \tag{2.54}$$

injecting this formula into the orthogonality condition 2.36 and using 2.23 to develop

$$||q \otimes v \otimes q^*|| = ||q||^2||v|| = ||v|| \Rightarrow ||q||^2 = 1, \tag{2.55}$$

we receive the unit norm condition on the quaternion

$$\boxed{q^* \otimes q = 1 = q \otimes q^*}, \tag{2.56}$$

which is similar to the orthogonality condition 2.41 of the rotation matrix (see [43] for details). The relative orientation condition 2.38 is also satisfied. [43]

The set of unit quaternions under operation of multiplication form a group, which is topologically a 3-sphere. The surface of this 3-sphere is a 3-dimensional unit sphere of $\mathbb{R}^4$ and is denoted as S^3. [43]

The Exponential Map exp()

We consider a unit quaternion $q \in S^3$, for which the orthogonally condition $q^* \otimes q = 1$ applies and look into its time derivative

$$\frac{d(q^* \otimes q)}{dt} = q^{*'} \otimes q + q^* \otimes q' = 0 \Rightarrow q^* \otimes q' = -(q^{*'} \otimes q) = -(q^* \otimes q')^* \tag{2.57}$$

where the term $q^* \otimes q'$ is the pure quaternion denoted as Ω, meaning it is equal to minus its conjugate and its real part is zero. This gives us

$$q^* \otimes q' = \Omega = \begin{bmatrix} 0 \\ \Omega \end{bmatrix} \in \mathbb{H}_p, \tag{2.58}$$

which yields the differential equation

$$q' = q \otimes \Omega. \tag{2.59}$$

[43]

Around the origin $q = 1$, so that $q' = \Omega \in \mathbb{H}$. The pure space of quaternions $\mathbb{H}_p$ is isomorphic to the Lie algebra $\mathfrak{so}(3)$, which describes the space of infinitesimal rotations in 3-dimensional space. It is also isomorphic to the *tangent space* of the 3-dimensional unit sphere S^3, which allows to use quaternions to represent infinitesimal rotations around any point on S^3. In the case of quaternions, it actually is the space of half-velocities, as will be explained later. Given a constant Ω, the differential equation 2.59 can be time-integrated as:

$$q(t) = q(0) \otimes e^{\Omega t} \tag{2.60}$$

[43]

Since $q(0)$ and $q(t)$ are unit quaternions, the exponential term $e^{\Omega t}$ is also a unit quaternion. Defining the quaternion $V \triangleq \Omega \Delta t$ we receive [43]:

$$q = e^V \tag{2.61}$$

This gives us the exponential map for quaternions, an application from the space of pure quaternions $\mathbb{H}_p$ to the space of infinitesimal rotations S^3 represented by unit quaternions [43]:

$$\exp : \mathbb{H}_p \leftarrow S^3; V \mapsto \exp(V) = e^V \tag{2.62}$$

The Capitalized Exponential Map

The pure quaternion $V = \theta u = \phi\, u/2$ in the exponential map 2.62 encodes the rotation axis u and only half of the rotation angle $\theta = \phi/2$. The rotation is accomplished by the double product $x_\phi = q \otimes x \otimes q^*$, meaning the vector x is rotated twice the angle encoded in q, or q only encodes only half the intended rotation. The direct relationship between the angle-axis rotation $\Phi = \phi u \in R^3$ and the quaternion V, can be expressed with the capitalized exponential map:

$$\mathrm{Exp} : R^3 \leftarrow S^3; \Phi \mapsto \mathrm{Exp}(\Phi) = e^{\Phi/2} \tag{2.63}$$

[43]

The relation to the exponential map is defined as [43]:

$$\mathrm{Exp}(\Phi) \triangleq \exp(\Phi/2) \tag{2.64}$$

We define the vector of angular velocities $\omega = 2\Omega \in \mathbb{R}^3$ and so that differential equation 2.59 becomes

$$q' = \frac{1}{2} q \otimes \omega \tag{2.65}$$

and its time-integral 2.60

$$q = e^{\omega t/2}. \tag{2.66}$$

[43]

Quaternion and Rotation Vector

The rotation vector $\Phi = \phi u$ represents a rotation of an angle ϕ in rad around the axis u. The exponential map can be developed using the *Euler formula*, which gives us:

$$\boxed{q \triangleq \mathrm{Exp}(\phi u) = e^{\phi u/2} = cos\frac{\phi}{2} + u\, sin\frac{\phi}{2} = \begin{bmatrix} cos(\phi/2) \\ u\, sin(\phi/2) \end{bmatrix}} \tag{2.67}$$

[43]

This equation is called the *rotation vector to quaternion formula*, which is denoted as $q = q\{\Phi\} = \mathrm{Exp}(\Phi)$ [43].

The Rotation Action

The rotation of a vector through a quaternion is done with the double quaternion product

$$x' = q \otimes x \otimes q^*, \tag{2.68}$$

where the quaternion $q = \text{Exp}(u\phi) \in S^3$. The vector x is written in quaternion form

$$x = xi + yi + zk = \begin{bmatrix} 0 \\ x \end{bmatrix} \in \mathbb{H}_p. \tag{2.69}$$

[43]

Note that the rotation action 2.68 can be proven by deriving the vector rotation formula 2.35, using the quaternion product 2.9 and the previously defined rotation vector to quaternion equation 2.67. [43]

The Double Cover of the Manifold of SO(3)

The angle θ between a unit quaternion q and the identity quaternion $q_1 = [1, 0, 0, 0]$ representing the origin of orientation is

$$cos\,\theta = q_1^T q = q(1) = q_w. \tag{2.70}$$

[43]

The rotation of an object in 3-dimensional space, about the angle ϕ, by the quaternion q is given by [43]:

$$q = \begin{bmatrix} q_w \\ q_v \end{bmatrix} = \begin{bmatrix} cos\,\phi/2 \\ u\ sin\,\phi/2 \end{bmatrix}. \tag{2.71}$$

This shows that $q_w = cos\,\theta = cos\,\phi/2$, meaning that the angle between a quaternion and the identity in 4-dimensional space is half the angle rotated by the quaternion in 3-dimensional space, so that [43]:

$$\theta = \phi/2. \tag{2.72}$$

The Figure 2.8 illustrates the rotation in 3-dimensional and 4-dimensional space. A rotation of the angle $\theta = 2\pi$ in 4-dimensional space results in a rotation of the angle $\phi = 2\theta = 4\pi$ in 3-dimensional space. [43]

2.5.5 Rotation Matrix and its Relationship to Quaternion

Given the rotation vector $\Phi = u\phi$, the exponential map produces the operator $q = \text{Exp}(u\phi)$ for a unit quaternion and $R = \text{Exp}(u\phi)$ for the rotation matrix. These operators rotate the vector x by the angle ϕ around the axis u. This means that

$$\forall \Phi, x \in \mathbb{R}^3, q = \text{Exp}(\Phi), R = \text{Exp}(\Phi) \Rightarrow q \otimes x \otimes q^* = R\,x. \tag{2.73}$$

[43]

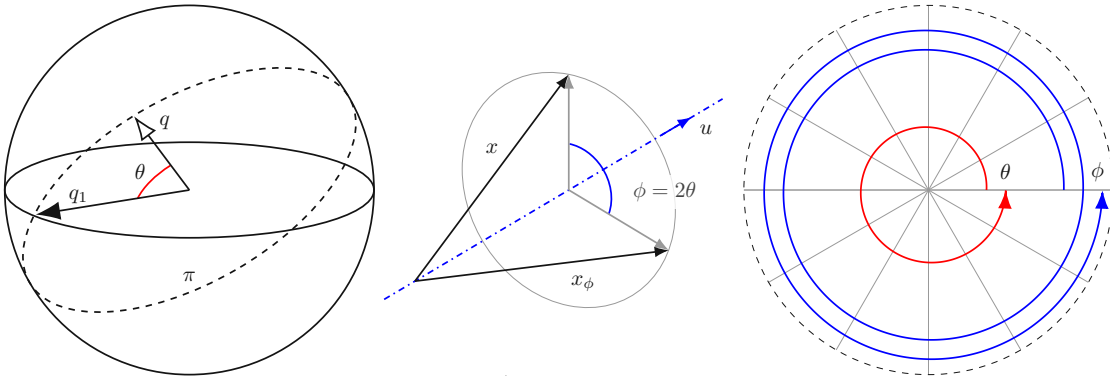


Figure 2.8: Double cover of the rotation manifold [43]
The left illustration shows the angle θ between the quaternion q and the identity q_1 in the unit 3-sphere. The illustration in the middle shows the resulting 3-dimensional rotation of $x_\phi = q \otimes x \otimes q^*$, which has double the angle of θ. The right illustration shows the 4- and 3-dimensional rotation planes. Red represents the one turn of the quaternion over the 3-sphere and blue represents two turns of the rotated vector in 3-dimensional space. [43]

Both sides of this identity are linear in x, therefore the *quaternion to rotation matrix* formula can be derived as

$$\boxed{R = \begin{bmatrix} q_w^2 + q_x^2 - q_y^2 - q_z^2 & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) \\ 2(q_x q_y + q_w q_z) & q_w^2 - q_x^2 + q_y^2 - q_z^2 & 2(q_y q_z - q_w q_x) \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{bmatrix}}, \tag{2.74}$$

which is denoted as $R = R\{q\}$. Using the matrix form of the quaternion product 2.13 - 2.15 and some easy developments we get

$$\boxed{R = (q_w^2 - q_v^T q_v) I + 2 q_v q_v^T + 2 q_w [q_v]_\times}. \tag{2.75}$$

[43]

The following properties apply to the relationship between the rotation matrix R and the quaternion q [43]:

$$R\{[1,0,0,0]^T\} = I \tag{2.76}$$
$$R\{-q\} = R\{q\} \tag{2.77}$$
$$R\{q^*\} = R\{q\} \tag{2.78}$$
$$R\{q_1 \otimes q_2\} = R\{q_1\} R\{q_2\} \tag{2.79}$$

We note that [43]:

2.76: The identity quaternion encodes the null rotation.

2.77: A quaternion encodes the the same rotation as its inverse.

2.78: The conjugate of a quaternion encodes the inverse rotation.

2.79: The order of consecutive rotations using quaternion products is the same as for matrix multiplications.

2.6 Time Derivatives and Integration

For this work we use a local local reference frame, it follows an introduction into local perturbations, their time derivatives and integration.

2.6.1 Local Perturbations

A perturbed orientation $\tilde{q}$ can be expressed as a composition of the unperturbed orientation q and a small local perturbation $\Delta q_{\mathcal{L}}$. So that we have,

$$\tilde{q} = q \otimes \Delta q_{\mathcal{L}} \qquad \tilde{R} = R\,\Delta R_{\mathcal{L}}, \tag{2.80}$$

for the quaternion and matrix composition. Note that the local perturbation appears on the right side of the composition, because of the *Hamilton convention*. The local perturbations $\Delta q_{\mathcal{L}}$ and $\Delta R_{\mathcal{L}}$ can be obtained through their equivalent vector form $\Delta\Phi = u\Delta\phi_{\mathcal{L}}$, using the exponential map. Which gives us

$$\tilde{q}_{\mathcal{L}} = q_{\mathcal{L}} \otimes \mathrm{Exp}(\Delta\Phi_{\mathcal{L}}) \qquad \tilde{R}_{\mathcal{L}} = \tilde{R}_{\mathcal{L}} \cdot \mathrm{Exp}(\Delta\Phi_{\mathcal{L}}) \tag{2.81}$$

which leads to the following expression of the local perturbation

$$\Delta\Phi_{\mathcal{L}} = Log(q^*_{\mathcal{L}} \otimes \tilde{q}_{\mathcal{L}}) = Log(R^T_{\mathcal{L}} \cdot \tilde{R}_{\mathcal{L}}). \tag{2.82}$$

[43]

It is possible to approximate a small perturbation angle $\Delta\Phi_{\mathcal{L}}$ by the Taylor expansions of 2.67 and 2.48 up to the linear terms with [43]:

$$\Delta q_{\mathcal{L}} \approx \begin{bmatrix} 1 \\ \frac{1}{2}\Delta\Phi_{\mathcal{L}} \end{bmatrix} \qquad \Delta R_{\mathcal{L}} \approx I + [\Delta\Phi_{\mathcal{L}}]_\times. \tag{2.83}$$

This means that perturbations can be specified in the local vector space $\Delta\Phi_{\mathcal{L}}$ tangent to the manifold of 3-dimensional rotations, denoted as $SO(3)$, at the actual orientation. [43]

2.6.2 Time Derivatives

The specification of local perturbations in a vector space allows to develop their time-derivatives. We define the original state as $q = q(t)$, the perturbed state as $\tilde{q} = q(t+\Delta t)$ and apply the definition of the time-derivative

$$\frac{dq(t)}{dt} \triangleq \lim_{\Delta t \to 0} \frac{q(t+\Delta t) - q(t)}{\Delta t}, \tag{2.84}$$

which gives us the local angular rates vector $\omega_{\mathcal{L}}$ of the local angular perturbation $\Delta\Phi_{\mathcal{L}}$ defined by q

$$\omega_{\mathcal{L}} \triangleq \frac{d\Phi_{\mathcal{L}}}{dt} \triangleq \lim_{\Delta t \to 0} \frac{\Delta\Phi_{\mathcal{L}}}{\Delta t}. \tag{2.85}$$

[43]

Developing the time-derivative results in [43]:

$$q' \triangleq \frac{1}{2} q \otimes \begin{bmatrix} 0 \\ \omega_{\mathcal{L}} \end{bmatrix} \tag{2.86}$$

Defining

$$\Omega(\omega) \triangleq [\omega]_R = \begin{bmatrix} 0 & -\omega^T \\ \omega & -[\omega]_\times \end{bmatrix} = \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix}, \tag{2.87}$$

and applying 2.13 and 2.86 we get

$$\boxed{q' = \frac{1}{2}\Omega(\omega_{\mathcal{L}})\, q = \frac{1}{2} q \otimes \omega_{\mathcal{L}} \qquad\qquad R' = R[\omega_{\mathcal{L}}]_\times}. \tag{2.88}$$

[43]

The expressions above are identical to the developments 2.65 and 2.46 of the rotation group $SO(3)$. However in this case the angular rates vector $\omega_{\mathcal{L}}$ refers to the local frame defined by the orientation q or R. [43]

2.6.3 Time Integration

The local rotation rate can be estimated by integrating the differential equation 2.88. The measurements $\omega(t_n)$ are provided by local sensors at discrete times $t_n = n\Delta t$. The differential equation 2.88 is

$$q'(t) = \frac{1}{2} q(t) \otimes \omega(t). \tag{2.89}$$

[43]

The zeroth order integration can be accomplished using the Taylor expansion of $q(t_n+\Delta t)$ around the time $t = t_n$. Defining

$$q \triangleq q(t), \qquad q_n \triangleq q(t_n), \qquad \omega \triangleq \omega(t), \qquad \omega_n \triangleq \omega(t_n), \tag{2.90}$$

the Taylor series is given as

$$q_{n+1} = q_n + q'_n\Delta t + \frac{1}{2!}q''_n\Delta t^2 + \frac{1}{3!}q'''_n\Delta t^3 + \frac{1}{4!}q'''_n\Delta t^4 + \cdots . \tag{2.91}$$

[43]

Zeroth Order Forward Integration

If the angular rate ω_n is constant over the time period $[t_n, t_{n+1}]$, then $\omega' = 0$ and 2.91 reduces to

$$q_{n+1} = q_n \otimes \left(1 + \frac{1}{2}\omega_n\Delta t + \frac{1}{2!}\left(\frac{1}{2}\omega_n\Delta t\right)^2 + \frac{1}{3!}\left(\frac{1}{2}\omega_n\Delta t\right)^3 + \cdots\right), \tag{2.92}$$

where we can identify the Taylor series of the exponential term $e^{\omega_n\Delta t/2}$ (see [43] for details). We can see from from the rotation vector to quaternion formula 2.67 that this exponential term corresponds to the quaternion, which represents the incremental rotation $\Delta\theta = \omega_n\Delta t$

$$e^{\omega\Delta t/2} = \mathrm{Exp}(\omega\Delta t) = q\{\omega\Delta t\} = \begin{bmatrix} \cos(\|\omega\|\Delta t/2) \\ \frac{\omega}{\|\omega\|}\sin(\|\omega\|\Delta t/2) \end{bmatrix}. \tag{2.93}$$

[43]

The forward integration is then given as [43]:

$$\boxed{q_{n+1} = q_n \otimes q\{\omega_n\Delta t\}} \tag{2.94}$$

2.7 Exponential Smoothing

Exponential smoothing was first introduced by Brown in 1950 [6], and extended by Holt in 1957 to deal with trends [21]. Since then a variety of different exponential smoothing variations have been proposed [11].

It follows a brief overview about the basic principals of exponential smoothing as introduced in [22]. The exponential smoothing algorithms used in this work are not based on statistical models. The presented algorithms therefore generate point forecasts instead of forecast distributions like the Kalman filter. The naive method treats the most recent observation as the only important one,

$$\hat{y}_{T+h|T} = y_T, \; h = 1, 2, 3 \ldots, \tag{2.95}$$

where T is the total number of observations and h and the number of steps ahead in time. All forecasts are equal to the last observed value of the series. This method assumes that previous observations provide no information and are therefore ignored. The average method on the other hand forecasts by averaging past observations,

$$\hat{y}_{T+h|T} = \frac{1}{T}\sum_{t=1}^{T} y_t,\ h = 1, 2, 3 \ldots, \tag{2.96}$$

this method assumes that all previous observation are of equal importance, no matter how far back in time they have been made. Exponential smoothing is in between those two extremes. It uses weighted averages of past observations to smooth current or forecast future values. The applied weights decay exponentially over time, meaning more recent observations have a higher associated weight. Exponential smoothing is considered to be a suitable and effective algorithm to smooth and forecast linear time series. [22]

It follows a brief introduction to simple exponential smoothing, and double exponential smoothing with one and two parameters. Furthermore a specific double exponential variation called Double Exponential Smoothed Prediction (DESP) will be introduced.

2.7.1 Brown's Simple Exponential Smoothing

The simplest form of exponential smoothing is called simple exponential smoothing. This method is suitable to forecast data without a clear trend or seasonal pattern. Simple exponential smoothing uses weighted averages to smooth current or forecast future values, the weights applied to observations decrease exponentially over time. A weighted average is given by

$$\hat{y}_{T+h|T} = \alpha y_T + \alpha(1-\alpha)y_{T-1} + \alpha(1-\alpha)^2 y_{T-2} + \ldots\ , 0 \leq \alpha \leq 1, \tag{2.97}$$

where α represents the smoothing parameter that controls the rate at which the weights decrease, meaning the further an observation lies in the past the smaller is the associated weight. The forecast

$$\hat{y}_{T+1|T} = \alpha y_T + (1-\alpha)\hat{y}_{T|T-1}, \tag{2.98}$$

is one step ahead in time, it is a weighted average of all past observations in the series $y_1 \ldots y_T$, which is equal to the weighted average between the most recent observation y_T and the previous forecast $\hat{y}_{T|T-1}$. The fitted values can be written as

$$\hat{y}_{t+1|t} = \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}, \tag{2.99}$$

for observations at the times $t = 1 \dots T$. Given the first fitted value, denoted as ℓ_0, we get

$$\begin{aligned} \hat{y}_{2|1} &= \alpha y_1 + (1-\alpha)\ell_0 \\ \hat{y}_{3|2} &= \alpha y_2 + (1-\alpha)\hat{y}_{2|1} \\ \hat{y}_{4|3} &= \alpha y_3 + (1-\alpha)\hat{y}_{3|2} \\ &\dots \\ \hat{y}_{T|T-1} &= \alpha y_{T-1} + (1-\alpha)\hat{y}_{T-1|T-2} \\ \hat{y}_{T+1|T} &= \alpha y_T + (1-\alpha)\hat{y}_{T|T-1}. \end{aligned} \tag{2.100}$$

[22]

Substituting each of the above equations into the following, we get the forecast equation [22]:

$$\hat{y}_{T+1|T} = \sum_{j=0}^{T-1} \alpha(1-\alpha)^j y_{T-j} + (1-\alpha)^T \ell_0 \ , \ 0 \leq \alpha \leq 1 \tag{2.101}$$

The component form provides an alternative representation. It considers only the level component ℓ_t of the time series at the time t and consists of a forecast and a smoothing equation

$$\begin{aligned} \textbf{Forecast equation} \quad & \hat{y}_{t+h|t} = \ell_t \\ \textbf{Smoothing equation} \quad & \ell_t = \alpha y_t + (1-\alpha)\ell_{t-1}, \ 0 \leq \alpha \leq 1. \end{aligned} \tag{2.102}$$

[22]

Setting $h = 1$ gives estimated values, so called fitting values, one step ahead in time and setting $t = T$ gives true forecast beyond the observed training data. Looking at the forecast equation, the forecast value at time $t + 1$ equals the estimated level at time t. The estimated level at each period t can be obtained by the smoothing equation (level equation). [22]

2.7.2 Brown's Linear Exponential Smoothing

This method, which is also called one-parameter exponential smoothing, is an extension of the simple exponential smoothing method and is able to deal with trends. The component form of this method consists of a forecast $\hat{y}_{t+h|t}$, a level ℓ_t and a trend b_t equation, as well as equations for the *smoothing statistics* Sy'_t and Sy''_t:

$$\begin{aligned} \textbf{Forecast equation} \quad & \hat{y}_{t+h|t} = \ell_t + hb_t \\ \textbf{Level equation} \quad & \ell_t = 2Sy'_t - Sy''_t \\ \textbf{Trend equation} \quad & b_t = \frac{\alpha}{1-\alpha}(Sy'_t - Sy''_t) \\ \textbf{First smoothing statistic} \quad & Sy'_t = \alpha y_t + (1-\alpha)Sy'_{t-1}, \ 0 \leq \alpha \leq 1 \\ \textbf{Second smoothing statistic} \quad & Sy''_t = \alpha Sy'_t + (1-\alpha)Sy''_{t-1}, \ 0 \leq \alpha \leq 1 \end{aligned} \tag{2.103}$$

[5]

The level and trend component use the double smoothing statistics, denoted as Sy'_t and Sy''_t, and the same smoothing parameter α is used to control both, the level and the trend. Note that Sy'_t smooths the observation of the time series $y_1 \dots y_t$ and Sy''_t smooths the values produces by the equation Sy'_t. [5]

2.7.3 Holt's Linear Exponential Smoothing

The component form of this method consists of a forecast equation, as well as a level and a trend smoothing equation [22]:

$$\begin{aligned} \textbf{Forecast equation} \quad & \hat{y}_{t+h|t} = \ell_t + hb_t \\ \textbf{Level equation} \quad & \ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1}),\ 0 \leq \alpha \leq 1 \\ \textbf{Trend equation} \quad & b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1},\ 0 \leq \beta \leq 1 \end{aligned} \tag{2.104}$$

The estimate of the level of the time series at time t is denoted as ℓ_t and the estimate of the trend or slope of the time series at time t is denoted as b_t. The level smoothing parameters is denoted as α and the trend smoothing parameter is denoted as β. The level equation ℓ_t is a weighted average of the observation y_t and a one step ahead forecast $\ell_{t-1} + b_{t-1}$ for the time t. The trend equation b_t is a weighted average of the estimated trend $\ell_t - \ell_{t-1}$ and the previous estimated trend b_{t-1}. [22]

2.7.4 Double Exponential Smoothing Based Prediction

Double Exponential Smoothed Prediction (DESP) is a double exponential smoothing method that extends Brown's linear exponential smoothing. It models a time series with a linear regression and its component form is given as:

$$\begin{aligned} \textbf{Forecast equation} \quad & \hat{y}_{t+h|t} = \ell_t + b_{t+h} \\ \textbf{Level equation} \quad & \ell_t = 2Sy'_t - Sy''_t - tb_t \\ \textbf{Trend equation} \quad & b_t = \frac{\alpha}{1-\alpha}(Sy'_t - Sy''_t) \\ \textbf{First smoothing statistic} \quad & Sy'_t = \alpha y_t + (1-\alpha)Sy'_{t-1},\ 0 \leq \alpha \leq 1 \\ \textbf{Second smoothing statistic} \quad & Sy''_t = \alpha Sy'_t + (1-\alpha)Sy''_{t-1},\ 0 \leq \alpha \leq 1 \end{aligned} \tag{2.105}$$

[31]

Applying some algebraic manipulations (see [31]) the forecast equation is as follows,

$$\hat{y}_{t+h} = (2 + \frac{\alpha h}{1-\alpha})Sy'_t - (1 + \frac{\alpha h}{1-\alpha})Sy''_t. \tag{2.106}$$

This method interpolates between the low $\hat{y}_{t+\lfloor h \rfloor}$ and high $\hat{y}_{t+\lceil h \rceil}$ forecast values to predict with real numbers as time steps h instead of integers. An interpolated forecast is given by the equation

$$\hat{y}_{t+h} = (\hat{y}_{t+\lceil h \rceil} - \hat{y}_{t+\lfloor h \rfloor})(h - \lfloor h \rfloor) + \hat{y}_{t+\lfloor h \rfloor}. \tag{2.107}$$

[31]

For rotations represented as quaternions, the spherical linear interpolation is used. The interpolated forecast equation for a rotations is then given as [31]:

$$\begin{aligned} \hat{q}_{t+h} &= \frac{\hat{q}_{t+\lfloor h \rfloor} sin((1-\rho)\Omega) + \hat{q}_{t+\lceil h \rceil} sin(\rho\Omega)}{sin\Omega} \\ \rho &= h - \lfloor h \rfloor \\ \Omega &= arccos(\hat{q}_{t+\lfloor h \rfloor} \odot \hat{q}_{t+\lceil h \rceil}) \end{aligned} \tag{2.108}$$

Where $\odot$ stands for a quaternion dot product. The predictions $\hat{q}_{t+\lfloor h \rfloor}$ and $\hat{q}_{t+\lceil h \rceil}$ have to be normalized before applying the spherical linear interpolation to ensure their rotational representation lies on the unit sphere of S^3. The initial smoothing statistics for both position and orientation at time zero are equal to their initial observation. [31]

2.8 Stochastic Fundamentals

This chapter introduces stochastic fundamentals the Kalman filter builds upon.

2.8.1 Probability

This section summarizes the basics of probability theory from [4]. The probability of the occurrence of a discrete event A is defined as

$$p(A) = \frac{Number\ of\ outcomes\ of\ event\ A}{Total\ number\ of\ possible\ outcomes} \tag{2.109}$$

The probability that either event A or B will occur is given by

$$p(A \cup B) = p(A) + p(B) - p(A \cap B), \tag{2.110}$$

if the two events are mutually exclusive then

$$p(A \cap B) = 0, \tag{2.111}$$

so that 2.110 reduces to

$$p(A \cup B) = p(A) + p(B). \tag{2.112}$$

Two events are called *independent* if one does not affect the other. If the events A and B are independent from each other, the probability that both will occur is equal to the product of their probabilities:

$$p(A \cap B) = p(A)p(B) \tag{2.113}$$

The *conditional probability* that A occurs after B has occurred, is given by:

$$p(A|B) = \frac{p(A \cap B)}{p(B)} \tag{2.114}$$

2.8.2 Random Variables and Density Function

This section introduces random variables and the density function as described in [4]. A *random variable* is a function that maps each point in the sample space to a real number. In a tracking scenario a random variable $X(t)$ maps time to a position and gives for any time t the expected position. For a *continuous variable* the probability $p(A)$ of a single discrete event A equals zero, $p(A) = 0$. Therefore we use the *cumulative distribution function* to evaluate the probability of events within some interval:

$$F_X(x) = P(-\infty, x] \tag{2.115}$$

[4]

The function $F_X(x)$ stands for the cumulative probability of the continuous random variable X, for all uncountable events up to and including x, and has the following properties [4]:

- $F_X(x) \to 0$ as $x \to -\infty$
- $F_X(x) \to 1$ as $x \to +\infty$
- $F_X(x)$ is a non-decreasing function of x.

The *probability density function* is given by the derivative of the cumulative distribution function 2.115 [4]:

$$f_X(x) = \frac{d}{dx} F_X(x) \tag{2.116}$$

The probability density function has, additionally to the properties of the cumulative distribution function, the following properties [4]:

- $f_X(x)$ is a non-negative function
- $\int_{-\infty}^{\infty} f_X(x)dx = 1$

The probability of all uncountable events within some interval $[a, b]$ for a random variable X, as illustrated in figure 2.9, is given by the integration of its density function [4]:

$$P(a \leq X \leq b) = \int_a^b f_X(x)dx \tag{2.117}$$

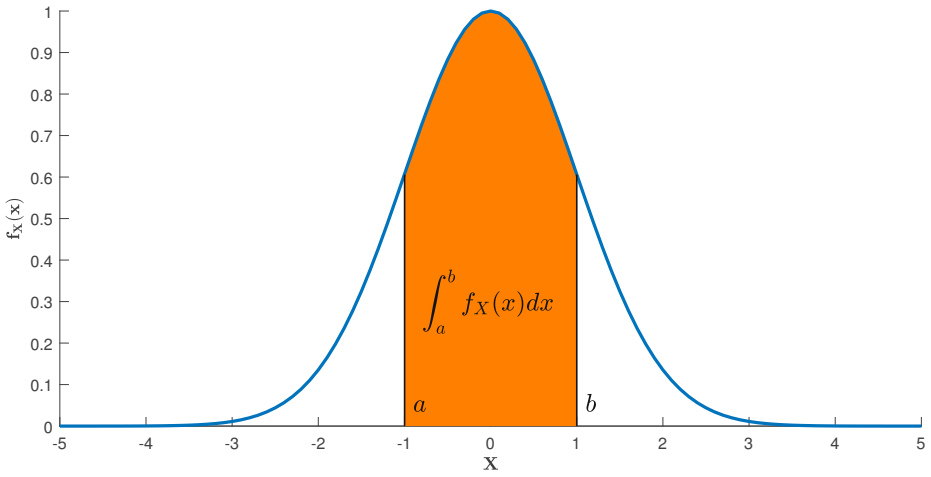


Figure 2.9: Probability within some interval $[a, b]$

2.8.3 Expected Value and Variance

This section introduces the expected value and variance as described in [4]. The average of a sequence of numbers also called *sample mean* of a discrete random variable X with N samples is given by [4]:

$$\overline{X} = \frac{X_1 + X_2 + \cdots + X_N}{N} \tag{2.118}$$

In tracking scenarios *continuous signals* have an uncountable sample space with an infinite number of samples. A *discrete* random variable can be approximated by averaging probability weighted events for each of the n possible outcomes of $x_1 \ldots x_n$ [4]:

$$\overline{X} \approx \frac{p_1 N x_1 + \cdots + p_n N x_N}{N} \tag{2.119}$$

By drawing samples indefinitely, we get the definition of the *expected value* for discrete random variables with n possible outcomes $x_1 \ldots x_n$ and their corresponding probabilities $p_1 \ldots p_n$ [4]:

$$\text{Expected value of X} = E(X) = \sum_{i=1}^{n} p_i x_i \tag{2.120}$$

For a *continuous* random variable the expected value is defined as [4]:

$$\text{Expected value of X} = E(X) = \int_{-\infty}^{\infty} x f_X(x) dx \tag{2.121}$$

Applying a function of a random variable X to the equations 2.120 and 2.121 will give us

$$E(g(X)) = \sum_{i=1}^{n} p_i g(x_i), \tag{2.122}$$

and,

$$E(g(X)) = \int_{-\infty}^{\infty} g(x) f_X(x) dx. \tag{2.123}$$

[4]

The expected value of a random variable is called *first statistical moment*. Applying $g(X) = X^k$ to the equation 2.123, we obtain the k^{th} statistical moment of a continuous random variable X,

$$E(X^k) = \int_{-\infty}^{\infty} x^k f_X(x) dx. \tag{2.124}$$

[4]

The *second moment* of a continuous random variable is given by [4]:

$$E(X^2) = \int_{-\infty}^{\infty} x^2 f_X(x) dx \tag{2.125}$$

Applying $g(X) = X - E(X)$ to the equation 2.125, we get the *variance* of the signal about the mean [4]:

$$\text{Variance of X} = E[(X - E(X))^2] = E(X^2) - E(X)^2 \tag{2.126}$$

The variance indicates how much a signal deviates from its mean and is a useful characteristic to determine how much jitter or noise is present within a signal. Another useful characteristic is the *standard deviation*, which is always positive and has the same unit as the original signal

$$\text{Standard deviation of X} = \sigma_X = \sqrt{\text{Variance of X}}. \tag{2.127}$$

[4]

2.8.4 Joint Probability of Continuous Random Variables

Joint random variables are called *multivariate* and deal simultaneously with more then one random variable. The case of two joint random variables is called *bivariate*. The cumulative distribution function for the bivariate case is

$$F_{XY}(x_0, y_0) = P(\{X \leq x_0\}) \text{ and } \{Y \leq y_0\}) = \int_{-\infty}^{y_0} \int_{-\infty}^{x_0} f_{XY}(x, y) dx dy, \tag{2.128}$$

where x_0 lies within X and y_0 within Y [7].

The probability that both X and Y lie within a region R is given by [7]:

$$P(\{X, Y\} \in R) = \int \int_R f_{XY}(x, y) dx dy \tag{2.129}$$

The joint probability density function is a complete description of the probabilistic relationship between the random variables X and Y. Note that two continuous random

variables X and Y are *statistically independent* if the product of their individual probabilities $f_X(x)$ and $f_Y(x)$ is equal to their joint probability $f_{XY}(x, y)$:

$$f_{XY}(x, y) = f_X(x) f_Y(y) \tag{2.130}$$

[7]

2.8.5 Bayes' Theorem

The *Bayes' theorem* describes the probability of an event, based on prior knowledge related to this event and can be derived from the conditional probability 2.114. The Bayes' theorem is given by [38]:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.131}$$

The *posterior* probability $P(A|B)$ describes the probability of the event B given that the event A has occurred. $P(B|A)$ represents the *likelihood*, which is the probability of event B given that A has ocurred. $P(A)$ and $P(B)$ are the observed probabilities for event A and B, called *prior* and *marginal* probability. [38].

The Bayes' theorem for continuous random variables is given by [4]:

$$f_{X|Y}(x) = \frac{f_{Y|X}(y) f_X(x)}{f_Y(y)} \tag{2.132}$$

2.8.6 Recursive Bayesian Estimation

The Bayesian estimation is applied recursive so that the prior and the current measurement are fused to compute the new posterior estimate. If another measurement is taken we can use that and our previous posterior estimate as prior estimate to compute a new posterior estimate, which is based now on two observations. [35]

At each time step k an estimate for the state, considering all observations up to that time ($Z_{1:k} = z_1 \dots z_k$), will be obtained with [35]:

$$\underbrace{f_{X|Z_{1:k}}(x)}_{posterior} = \frac{\overbrace{f_{Z|X}(z_k)}^{\text{measurement model}} \overbrace{f_{X|Z_{1:k-1}}(x)}^{\text{prior}}}{\underbrace{f_{Z|Z_{1:k-1}}(z_k)}_{\text{normalization constant}}} \tag{2.133}$$

The posterior probability function $f_{X|Z_{1:k}}(x)$ is a result of incorporating all measurements up to and including time k. The measurement model $f_{Z|X}(z_k)$ represents the likelihood of the k^{th} measurement and $f_{X|Z_{1:k-1}}(x)$ the prior estimate. The recursive Bayesian estimator allows to add new information by simply multiplying the prior and the current likelihood, whatever form of probability density functions they have. The Kalman filter is using this mechanism with a Gaussian prior and likelihood to create estimates. [35]

2.8.7 Correlation and Covariance

The expected value of the product of two random variables X and Y is given by [7]:

$$E(XY) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} xy f_{XY}(x,y) dxdy \tag{2.134}$$

In the case that X and Y are independent (see 2.130), f_{XY} from the equation 2.134 can be factored and reduced to [7]:

$$E(XY) = \int_{-\infty}^{\infty} x f_X(x) dx \int_{-\infty}^{\infty} y f(y) dy = E(X)E(Y) \tag{2.135}$$

The two random variables X and Y are *uncorrelated*, if the equation 2.135 applies. Furthermore the random variables X and Y are *orthogonal*, if $E(XY) = 0$. The *covariance* of X and Y is defined as:

$$\text{Covariance of X and Y} = Cov(X,Y) = E[(X - E(X))(Y - E(Y))] \tag{2.136}$$

[7]

2.8.8 Spectral Signal Characteristics

Additionally to the previously introduced spatial characteristics of a random variable we can look at its temporal, hence spectral characteristics. The magnitude of the variance tells us how much noise or jitter is present within the signal, but it doesn't say anything about the rate of jitter over time. A useful spectral characteristic is the *autocorrelation*, its the correlation of a random signal with itself over time. [4]

The autocorrelation of a random signal $X(t)$ for the arbitrary sample times t_1 and t_2 is given by

$$R_X(t_1,t_2) = E[X(t_1)X(t_2)], \tag{2.137}$$

it is a measure for how much a process is correlated at two different times with itself [7].

If a process is *stationary* - meaning its density is invariant over time - the equation 2.137 can reduced to a function of the time difference $\tau = t_2 - t_1$ [7]:

$$R_X(t_1,t_2) = E[X(t)X(t+\tau)] \tag{2.138}$$

If the underlying process is changing rapidly over time, the autocorrelation function will decrease rapidly with an increase of τ. The auto correlation is a function of time, meaning it has also a spectral interpretation in the frequency domain. For a stationary process the temporal-spectral relationship is given by the *Wiener-Khinchine relation*:

$$S_X(j\omega) = \mathcal{F}[R_X(\tau)] = \int_{-\infty}^{\infty} R_X(\tau) e^{-j\omega\tau} d\tau \tag{2.139}$$

[4]

Where $\mathcal{F}[\cdot]$ stands for the Fourier transformation, and ω indicates the number of 2π cycles per second [7]. The *power spectral density* function of the random signal is denoted as S_X. This equation describes the relationship between time and frequency spectrum of the underlying signal [4].

White noise can described as autocorrelation function of a *dirac delta* function $\delta(\tau)$, with a value of zero everywhere and a constant magnitude A when $\tau = 0$. The dirac delta function is given by:

$$R_X(\tau) = \begin{cases} \text{if } \tau = 0 \text{ then } A \\ \text{else } 0 \end{cases} \tag{2.140}$$

[4]

In that case the autocorrelation function is a spike and results in a constant frequency spectrum through the Fourier transformation. This means white noise has in theory power at all frequencies in the spectrum and is completely uncorrelated with itself at any time except the present. In reality random signals can be modelled as filtered white noise which is both band-limited in the frequency domain and more correlated in the time domain. [4]

2.8.9 Normal or Gaussian Distribution

Many naturally occurring random processes appear to be normally distributed or very close [4]. Furthermore the *central limit theorem* states that "under moderate conditions the sum of random variables with any distribution tends towards a normal distribution" [4]. The probability density function for a normally distributed continuous random process $X \sim \mathcal{N}(\mu, \sigma^2)$ with a mean μ and a variance σ^2 is given by [4]:

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}, -\infty < x < \infty \tag{2.141}$$

The bell shaped Gaussian distribution function is shown as graph in figure 2.10.

2.8.10 Multivariate and Covariance Matrix

Multivariate random variables can be represented as a random vector X with k random variables $X_1, X_2 \dots X_k$ [7]:

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_k \end{bmatrix} \tag{2.142}$$

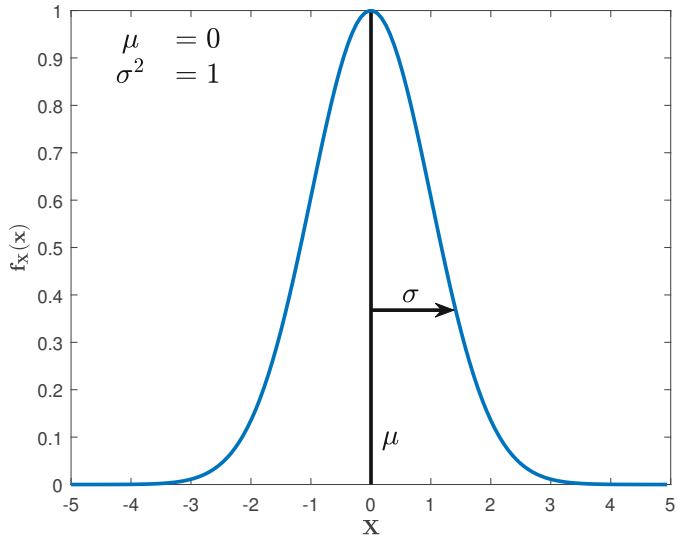


Figure 2.10: Gaussian density function

The random variables of X may be correlated and have non-zero means [7]. The mean values $\mu_1, \mu_2, \dots \mu_k$ of these random variables can be written as a vector μ [7]:

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{bmatrix} \tag{2.143}$$

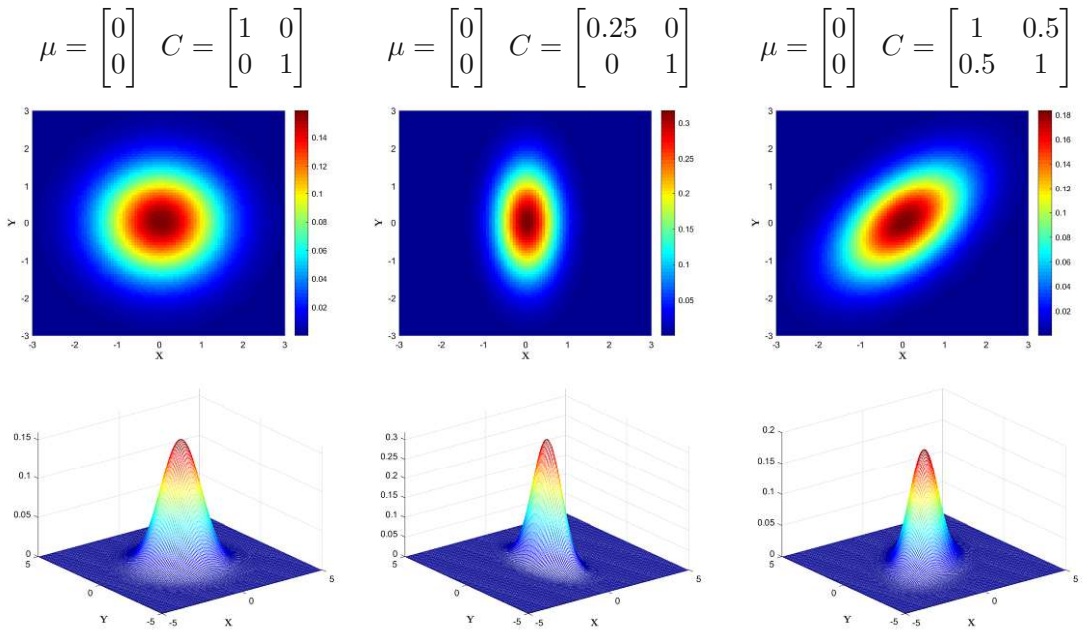
The variances and correlation of the k variates of the random vector X can be represented with the *covariance matrix* [7]:

$$C = \begin{bmatrix} E[(X_1 - \mu_1)^2] & E[(X_1 - \mu_1)(X_2 - \mu_2)] & \dots & E[(X_1 - \mu_1)(X_k - \mu_k)] \\ E[(X_2 - \mu_2)(X_1 - \mu_1)] & E[(X_2 - \mu_2)^2] & \dots & [E(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(X_k - \mu_k)(X_1 - \mu_1)] & [(X_k - \mu_k)(X_2 - \mu_2)] & \dots & E[(X_k - \mu_k)^2] \end{bmatrix} \tag{2.144}$$

The terms along the major diagonal of the covariance matrix C represent the variances and the off-diagonal terms the covariances [7]. The random variables $X_1, X_2, \dots X_k$ are called *jointly normal* or *jointly gaussian* if their joint probability can be described with the density function [7]:

$$f_X(x_1, \dots x_k) = \frac{1}{\sqrt{(2\pi)^k |C|}} e^{-\frac{1}{2}(x-\mu)^T C^{-1}(x-\mu)} \tag{2.145}$$

Note that in order for f_X to be properly defined $C^{-}1$ must exist, meaning C must be non-singular [7].



(a) Independent symmetric (b) Independent asymmetric (c) Dependent asymmetric

Figure 2.11: Bivariate gaussian normal density function

Figure 2.11 illustrates different bivariate gaussian density functions. The center of the gaussian density function is defined by the mean values μ_X and μ_Y of X and Y. The form of the gaussian bell curve is shaped by the covariance matrix C. If the random variables are uncorrelated, meaning $E[(X-\mu_X)(Y-\mu_y)] = 0$, the bell curve is distorted vertically or/and horizontally, otherwise it is also distorted diagonally.

2.9 State-Space-Model

A state-space-model uses state variables to estimate and control the state of a system, and keeps track of otherwise untraceable system dynamics [4]. It can be described by an n-th order difference equation [4]:

$$y_{i+1} = a_{0,i}y_i + \cdots + a_{n-1,i}y_{i-n+1} + u_i, i \geq 0 \tag{2.146}$$

Where u_i represents a white noise random process with a zero-mean and an autocorrelation of [4]:

$$E(u_i, u_j) = R_u = Q_i\delta_{ij} \tag{2.147}$$

The initial random variables $\{y_0, y_{-1}, \ldots, y_{-n+1}\}$ have a zero-mean and a known $n \times n$ covariance matrix [4]:

$$P_0 = E(y_{-j}, y_{-k}), j, k \in \{0, n-1\} \tag{2.148}$$

It is also assumed that [4]:

$$E(u_i, y_i) = 0 \text{ for } -n+1 \leq j \leq 0 \text{ and } i \geq 0 \implies E(u_i, y_i) = 0, i \geq j \geq 0 \tag{2.149}$$

This means that the noise is statistically independent from the estimated process [4]. The difference equation 2.146 can also be written as [4]:

$$\grave{x}_{i+1} \equiv \begin{bmatrix} y_{i+1} \\ y_i \\ y_{i-1} \\ \vdots \\ y_{i-n+2} \end{bmatrix} = \underbrace{\begin{bmatrix} a_0 & a_1 & \dots & a_{n-2} & a_{n-1} \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}}_{F} \underbrace{\begin{bmatrix} y_i \\ y_{i-1} \\ y_{i-2} \\ \vdots \\ y_{i-n+1} \end{bmatrix}}_{\grave{x}_i} + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{G} u_i \tag{2.150}$$

Which gives us the *state-space-model* [4]:

$$\grave{x}_{i+1} = F\grave{x}_i + Gu_i \tag{2.151}$$

$$\grave{y}_i = H\grave{x}_i \tag{2.152}$$

The new state $\grave{x}_{i+1}$ is a linear combination of the previous state $\grave{x}_i$ and the *process noise* u_i. The process measurement or observation denoted as $\grave{y}_i$ and is derived from the internal state $\grave{x}_i$. The equation 2.151 is referred to as *process model* and the equation 2.152 as *measurement process*. These two equation are the basis of linear estimation methods such as the Kalman filter. The *process model* describes the transformation of the process state over time and the *measurement model* the relationship between the process state and the measurements. [4]

2.9.1 Measurement and Process Model

Sensors measurements are typically noisy, additionally electrical noise is added to the signal through electrical circuits. Each type of sensor has its own limitations related to the physical medium they depend on, pushing the envelope on those limitations causes a degradation of the signal. The time varying noise applied to the signal affects the quantity and the quality of information provided by a sensor. A solution for this problem is to interpret a result obtained from a sensor as one part of an overall sequence of estimates and use analytical measurement models to incorporate some notion of measurement noise or uncertainty. Another problem is that the actual state transform model is unknown, but it is possible to establish models based on recent state transformations and make predictions over relatively short intervals. The models and their predications are not always accurate, therefore similar to sensor information, ongoing estimates are qualified as they are combined with measurements and previous estimates. Like measurement models, process models incorporate some notion of random motion or uncertainty. Which leads us to the *observer design problem*, a general problem of linear systems theory. The basic problem is that a linear systems internal state has to be estimated by its

output. This is a also known as *black box* problem, where one can access the systems output signals but cannot observe what's inside. This problem can be tackled with the previously introduced state-space model. [4]

The process model and its corresponding process noise, denoted as random variable w_k, can be represented as difference equation similar to equation 2.151 [4]:

$$x_k = Fx_{k-1} + Bu_k + w_{k-1} \tag{2.153}$$

The measurement model and its corresponding measurement noise, denoted as random variable v_k, can be represented as linear expression similar to equation 2.152 [4]:

$$z_k = Hx_k + v_k \tag{2.154}$$

The measurements, represented as the dependent variable z_k, do do not have to be elements of the state, but can be any linear combination of the states elements [4].

2.10 Linear Kalman Filter

2.10.1 Process Description

The Kalman filter relies on the state-space model (see section 2.9) to estimate the state $x \in \mathbb{R}^n$ of a discrete process, which changes over time and is controlled by the stochastic linear difference equation

$$x_k = Fx_{k-1} + Bu_k + w_k, \tag{2.155}$$

and the measurement $z \in \mathbb{R}$, which can be described by the linear expression

$$z_k = Hx_k + v_k. \tag{2.156}$$

[4]

The equation 2.155 is also called *time update* equation and predicts a new estimate. The $n \times n$ matrix F represents the state transition matrix and relates the previous state x_{k-1} to the current state x_k, while the $n \times l$ matrix B relates the optional control input $u \in \mathbb{R}^l$ to the current state x_k. The random variable w_k represents the process noise of the time update. The equation 2.156 is also referred to as *measurement update* equation. The $m \times n$ matrix H relates the state x_k to the measurement z_k. The random variable v_k represents the measurement noise. [4]

Note that the random variables w and v are white noise, are independent from each other, and have normal probability distributions [4]:

$$f_W(w) \sim \mathcal{N}(0, Q) \tag{2.157}$$

$$f_V(v) \sim \mathcal{N}(0, R) \tag{2.158}$$

The process covariance matrix is denoted as Q and the measurement covariance matrix as R. In reality the matrices F, B, H, Q and R might change with each time step, but for simplicity we assume they are constant. [4]

2.10.2 Derivation

Measurement Update

The measurement model can be represented as linear expression [35]:

$$z_k = Hx_k + v_k \tag{2.159}$$

The measurement z_k is corrupted by a Gaussian noise $f_V(v) \sim \mathcal{N}(0, R)$, which has a mean of zero, a covariance R and number of dimensions n, using 2.145 we have [35]:

$$f_V(v) = \frac{1}{\sqrt{(2\pi)^n |R|}} e^{-\frac{1}{2} v^T R^{-1} v} \tag{2.160}$$

Assuming that the measurement likelihood and the prior density functions are Gaussian distributed, the measurement likelihood is given by [35]:

$$f_{Z|X}(z) = \frac{1}{\sqrt{(2\pi)^n |R|}} e^{-\frac{1}{2}(z-Hx)^T R^{-1}(z-Hx)} \tag{2.161}$$

We define the *prior* state estimate $\hat{x}_{k|k-1} \in \mathbb{R}^n$ and the $n \times n$ covariance matrix $P_{k|k-1}$ at the time step k, which are representing prior knowledge of the process. Additionally we define the *posterior* state estimate $\hat{x}_{k|k} \in \mathbb{R}^n$, the $n \times n$ covariance matrix $P_{k|k}$ and the measurement z_k at time step k. [4]

The prior probability is then given by[35]:

$$f_{X|Z_{1:k-1}}(x) = \frac{1}{\sqrt{(2\pi)^n |P_{k|k-1}|}} e^{-\frac{1}{2}(x-\hat{x}_{k|k-1})^T P_{k|k-1}^{-1}(x-\hat{x}_{k|k-1})}. \tag{2.162}$$

The posterior probability $f_{X|Z_{1:k}}(x)$ can be calculated for every time step k by applying the *Bayes theorem* recursively (see 2.8.6) [35]. Applying the measurement likelihood 2.161 and the prior 2.162 to the posterior equation 2.133 we get [35]:

$$f_{X|Z_{1:k}}(x) = \frac{\frac{1}{\sqrt{(2\pi)^n |R|}} e^{-\frac{1}{2}(z-Hx)^T R^{-1}(z-Hx)} \frac{1}{\sqrt{(2\pi)^n |P_{k|k-1}|}} e^{-\frac{1}{2}(x-\hat{x}_{k|k-1})^T P_{k|k-1}^{-1}(x-\hat{x}_{k|k-1})}}{C(z)} \tag{2.163}$$

Since a Gaussian multiplied with a Gaussian equals a Gaussian, the resulting posterior $f_{X|Z_{1:k}}(x)$ will also be a Gaussian. The scale factor and the exponential function of the equation 2.163 can be ignored which gives us [35]:

$$(z - Hx)^T R^{-1}(z - Hx) + (x - \hat{x}_{k|k-1})^T P_{k|k-1}^{-1}(x - \hat{x}_{k|k-1}) \tag{2.164}$$

The equation above can be expressed in a quadratic way [35]:

$$(x - x_{\hat{k}|k})^T P_{k|k}^{-1}(x - x_{\hat{k}|k}) \tag{2.165}$$

It is possible to derive the Minimum mean square error (MMSE) estimate $\hat{x}_{k|k}$ and the covariance $P_{k|k}$ with some lengthy algebraic modifications, by expanding and rearranging the equation 2.164 and 2.165 (see [35]).

Given a measurement z_k, its uncertainty covariance matrix R, the prior state estimate $\hat{x}_{k|k-1}$ and the covariance $P_{k|k-1}$, the new estimate and covariance are given as [35]:

$$\begin{aligned} S_k &= HP_{k|k-1}H^T + R \\ K_k &= P_{k|k-1}H^T S_k^{-1} \\ \hat{x}_{k|k} &= \hat{x}_{k|k-1} + K(z_k - H\hat{x}_{k|k-1}) \\ P_{k|k} &= P_{k|k-1} - K_k S_k K_k^T \end{aligned} \tag{2.166}$$

The term $z_k - H\hat{x}_{k|k-1}$ is denoted as *Innovation*, the covariance matrix S as *Innovation Covariance* and the Kalman gain as K. [35]

Time Update

The time update process can be modelled using the stochastic linear difference equation [4]:

$$x_k = Fx_{k-1} + Bu_k + w_k \tag{2.167}$$

The uncertainty in the time update process model is incorporated by adding the Gaussian noise $f_W(w) \sim \mathcal{N}(0, Q)$ [35]. The new MMSE estimate $\hat{x}_{k|k-1}$ at time step k, incorporates measurements up to and including time step $k-1$, and is given by [35]:

$$\begin{aligned} \hat{x}_{k|k-1} &= E[x_k|Z_{1:k-1}] && (2.168)\\ &= E[Fx_{k-1} + Bu_k + w_k|Z_{1:k-1}] && (2.169)\\ &= FE[x_{k-1}|Z_{1:k-1}] + Bu_k + E[w_k|Z_{1:k-1}] && (2.170)\\ &= F\hat{x}_{k-1|k-1} + Bu_k + 0 && (2.171) \end{aligned}$$

This means our time update process model gives us the new best estimate $\hat{x}_{k|k-1}$ at time step k [35]. The covariance matrix of the prediction step is given by [35]:

$$P_{k|k-1} = E[(x_k - \hat{x}_{k|k-1})(x_k - \hat{x}_{k|k-1})^T|Z_{1:k}] \tag{2.172}$$

Applying the equations 2.167 and 2.171 to 2.172 we receive [35]:

$$P_{k|k-1} = E[(F(x_{k-1} - \hat{x}_{k-1|k-1}) + w_k)(F(x_{k-1} - \hat{x}_{k-1|k-1}) + w_k)^T|Z_{1:k-1}] \tag{2.173}$$

Assuming that the previous best estimate $\hat{x}_{k-1|k-1}$ and the noise vector w_k are uncorrelated, and expanding 2.173, causes all cross terms between $\hat{x}_{k-1|k-1}$ and w_k to disappear [35]. Which gives us the prior covariance matrix [35]:

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q \tag{2.174}$$

2.10.3 Equations

The Kalman filter is a *recursive* algorithm where its output becomes the input of the next iteration and consists of a time update (prediction) and a measurement update [4, 35]. The initial values for $\hat{x}_{0|0}$ and $P_{0|0}$ are derived based on knowledge of the data [35]. The Kalman filter is a *predictor-corrector* algorithm where the time update functions as *predictor* and the measurement update as *corrector* [4].

Time update equations (predictor) [4]:

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + Bu_k \tag{2.175}$$
$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q \tag{2.176}$$

The time update projects the current state and error covariance estimates forward in time to obtain the *prior* estimates for the next time step [4].

Measurement update equations (corrector) [4]:

$$S_k = HP_{k|k-1}H^T + R \tag{2.177}$$
$$K_k = P_{k|k-1}H^T S_k^{-1} \tag{2.178}$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K(z_k - H\hat{x}_{k|k-1}) \tag{2.179}$$
$$P_{k|k} = P_{k|k-1} - K_k S_k K_k^T \tag{2.180}$$

The measurement update corrects the prediction by fusing the *prior* with a new measurement to obtain the improved *posterior* estimate [7, 35].

The difference between the measurement z_k and the prediction $H\hat{x}_{k|k-1}$ is called *innovation*. They are both identical, if the prediction is in complete agreement with the measurement. The measurement update does not have to happen every iteration. Instead if no new measurement is available the last best estimate at time k is simply the prediction $\hat{x}_{k|k-1}$. At each time update Q is added to covariance matrix $P_{k|k-1}$, which gets therefore inflated. This means the time update increases the uncertainty, because our model is inaccurate. The measurement update on the other hand deflates the covariance matrix $P_{k|k}$, because $K_k S_k K_k^T$ gets subtracted from the prior covariance $P_{k|k-1}$. Each measurement update adds information to the state estimate $\hat{x}_{k|k}$, which increases its certainty. [35]

2.11 Extended Kalman Filter

2.11.1 Process Description

The linear Kalman filter from section 2.10 is only applicable to linear processes, but the majority of real world applications require non-linear models [35].

The extended Kalman filter is able to solve this problem and deal with non-linear scenarios like rotations. In order to deal with non-linear processes the estimation can be linearized with a Taylor series approximation around the current estimate using the partial derivatives of the prediction and measurement function. This makes it possible to compute estimates for non-linear relationships [4].

We want to estimate the state vector $x \in \mathbb{R}^n$ of a non-linear process governed by the non-linear stochastic difference equation

$$x_k = f(x_{k-1}, u_k, w_k), \tag{2.181}$$

and the measurement $z \in \mathbb{R}^n$,

$$z_k = h(x_k, v_k). \tag{2.182}$$

[4, 35]

The time update process noise is denoted as w_k and the measurement update process noise as v_k. The non-linear function f relates the previous state x_{k-1} at time step $k-1$ to the current state x_k at time step k. The input parameters of the time update function are the control input u_k and the zero mean process noise w_k. The non-linear function h relates the state x_k to the measurement z_k. In the practice we do not know the values of w_k and v_k, but the state and measurement can be approximated without them. [4]

2.11.2 Derivation

Time Update

The process and measurement model are given by [35]:

$$x_k = f(x_{k-1}, u_k) + w_k \tag{2.183}$$
$$z_k = h(x_k, u_k) + v_k \tag{2.184}$$

The extended Kalman filter linearizes the non-linear prediction and measurement model around the current estimate, by using a Taylor series expansion [35]. Let's assume that we have $\hat{x}_{k-1|k-1}$ so that x_k is given by [35]:

$$x_k = f(\hat{x}_{k-1|k-1}, u_k) + F_x(x_{k-1|k-1} - \hat{x}_{k-1|k-1}) + \dots \tag{2.185}$$

The term F_x represents the Jacobian matrix of the function f with respect to the state x [35]. The jacobian matrix contains all the first order derivatives of the vector function

$f(x)$ and is specified as [35]:

$$F_x = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2.186}$$

Knowing that

$$\hat{x}_{k|k-1} = E[x_k|Z_{1:k-1}] \tag{2.187}$$

and

$$\hat{x}_{k-1|k-1} = E[x_{k-1}|Z_{1:k-1}], \tag{2.188}$$

we can substitute the equation 2.185 into 2.187, and apply 2.188 to get the estimate [35]:

$$\begin{aligned} \hat{x}_{k|k-1} &= E[f(\hat{x}_{k-1|k-1}, u_k) + F_x(x_{k-1} - \hat{x}_{k-1|k-1}) + \cdots + w_k|Z_{1:k-1}] \\ &= E[f(\hat{x}_{k-1|k-1}, u_k)|Z_{1:k-1}] + F_x(\hat{x}_{k-1|k-1} - \hat{x}_{k-1|k-1}) \\ &= f(\hat{x}_{k-1|k-1}, u_k) \end{aligned} \tag{2.189}$$

This means that our new prediction at time step k is the result of our non-linear prediction model f given the last best estimate $\hat{x}_{k-1|k-1}$ and the control signal u_k as input parameters. Next we derive the covariance by looking at the behavior

$$\tilde{x}_{k|k-1} = x_k - \hat{x}_{k|k-1}, \tag{2.190}$$

because we know that

$$P_{k|k-1} = E[\tilde{x}_{k|k-1}\tilde{x}_{k|k-1}^T|Z_{1:k-1}]. \tag{2.191}$$

[35]

The Jacobians of f are evaluated at $\hat{x}_{k-1|k-1}$, so that we can write [35]:

$$\begin{aligned} \tilde{x}_{k|k-1} &= x_k - \hat{x}_{k|k-1} && (2.192) \\ &\approx f(\hat{x}_{k-1|k-1}, u_k) + F_x(x_{k-1} - \hat{x}_{k-1|k-1}) + w_k - f(\hat{x}_{k-1|k-1}, u_k) && (2.193) \\ &= F_x(x_{k-1|k-1} - \hat{x}_{k-1|k-1}) + w_k && (2.194) \\ &= F_x(\tilde{x}_{k-1|k-1}) + w_k && (2.195) \end{aligned}$$

so that [35]:

$$\begin{aligned} P_{k|k-1} &= E[\tilde{x}_{k|k-1}\tilde{x}_{k|k-1}^T|Z_{1:k-1}] && (2.196) \\ &\approx E[(F_x\tilde{x}_{k-1|k-1} + w_k)(F_x\tilde{x}_{k-1|k-1} + w_k)^T|Z_{1:k-1}] && (2.197) \\ &= E[F_x\tilde{x}_{k-1|k-1}\tilde{x}_{k-1|k-1}^T F_x^T|Z_{1:k-1}] + E[w_k w_k^T|Z_{1:k-1}] && (2.198) \\ &= F_x P_{k-1|k-1} F_x^T + Q && (2.199) \end{aligned}$$

The prediction model is then given as

$$x_k = f(x_{k-1}, u_k, w_k), \tag{2.200}$$

where the noise w_k is not simply additive. The posterior equations can be deriving by applying a multivariate Taylor series expansion. The derivation procedure is complex but the end result is intuitive. The state estimate remains unchanged and the posterior covariance matrix becomes :

$$P_{k|k-1} = F_x P_{k-1|k-1} F_x^T + W_w Q W_w^T \tag{2.201}$$

[35]

The jacobian matrix W_w is defined as [4]:

$$W_w = \frac{\partial f}{\partial w} \tag{2.202}$$

Measurement Update

We assume our non-linear measurement model is given by [35]:

$$z_k = h(x_k) + v_k \tag{2.203}$$

The predicted measurement $z_{k|k-1}$ is given by projecting the last best estimate $\hat{x}_{k|k-1}$ through the measurement model [35]:

$$z_{k|k-1} = E(z_k | Z_{1:k-1}) \tag{2.204}$$

$$z_{k|k-1} = h(\hat{x}_{k|k-1}) \tag{2.205}$$

The innovation, which is the difference of the actual measurement z_k and its prediction $z_{k|k-1}$, can be derived using a Taylor series [35]:

$$z_k \approx h(\hat{x}_{k|k-1}) + H_x(\hat{x}_{k|k-1} - x_k) + \cdots + v_k \tag{2.206}$$

$$\tilde{z}_{k|k-1} = z_k - z_{k|k-1} \tag{2.207}$$

$$= h(\hat{x}_{k|k-1}) + H_x(\hat{x}_{k|k-1} - x_k) + \ldots v_k - h(\hat{x}_{k|k-1}) \tag{2.208}$$

$$= H_x(\hat{x}_{k|k-1} - x_k) + v_k \tag{2.209}$$

$$= H_x(\tilde{x}_{k|k-1}) + v_k \tag{2.210}$$

Where the term H_x is the Jacobian matrix [4]:

$$H_x = \frac{\partial h}{\partial x} \tag{2.211}$$

The innovation covariance matrix S is then given by [35]:

$$S = E[\tilde{z}_{k|k-1} \tilde{z}_{k|k-1}^T | Z_{1:k-1}] \tag{2.212}$$

$$= E[(H_x(\tilde{x}_{k|k-1}) + v_k)(H_x(\tilde{x}_{k|k-1}) + v_k)^T | Z_{1:k-1}] \tag{2.213}$$

$$= H_x E[\tilde{x}_{k|k-1} \tilde{x}_{k|k-1}^T | Z_{1:k-1}] + E[v_k v_k^T | Z_{1:k-1}] \tag{2.214}$$

$$= H_x P_{k|k-1} H_x^T + R \tag{2.215}$$

The term $E[\tilde{x}_{k|k-1}v_k^T|Z_{1:k}]$ is zero because the measurement noise and the prediction error are expected to be uncorrelated [35]. Let's assume we have a gain K to relate the prediction and the innovation with the posterior [35]:

$$\hat{x}_{k|k} = \underbrace{\hat{x}_{k|k-1}}_{prediction} + \underbrace{K}_{gain}\underbrace{(\underbrace{z_k}_{measurement} - \underbrace{h(\hat{x}_{k|k-1})}_{\text{predicted measurement}})}_{innovation} \tag{2.216}$$

We can now look at the expansion of $\tilde{x}_{k|k}$ and take its square to get the covariance matrix $P_{k|k}$ [35]:

$$P_{k|k} = E[\tilde{x}_{k|k}\tilde{x}_{k|k}^T|Z_{1:k}] \tag{2.217}$$
$$\tilde{x}_{k|k} = \hat{x}_{k|k} - x_k \tag{2.218}$$
$$= \hat{x}_{k|k} - x_k \tag{2.219}$$
$$= \hat{x}_{k|k-1} + K(z_k - h(\hat{x}_{k|k-1})) - x_k \tag{2.220}$$
$$= \hat{x}_{k|k-1} + K\tilde{z}_{k|k-1} - x_k \tag{2.221}$$

Substituting our innovation estimate $\tilde{z}_{k|k-1}$ with 2.210

$$= \hat{x}_{k|k-1} + K(H_x\tilde{x}_{k|k-1} + v_k) - x_k \tag{2.222}$$
$$= \tilde{x}_{k-1} + KH_x\tilde{x}_{k|k-1} + Kv_k \tag{2.223}$$
$$= (I - KH_x)\tilde{x}_{k|k-1} + Kv_k \tag{2.224}$$

The covariance matrix $P_{k|k}$ is now given by [35]:

$$P_{k|k} = E[\tilde{x}_{k|k-1}\tilde{x}_{k|k-1}^T|Z_{1:k}] \tag{2.225}$$
$$= E[((I - KH_x)\tilde{x}_{k|k-1} + Kv_k)((I - KH_x)\tilde{x}_{k|k-1} + Kv_k)^T|Z_{1:k}] \tag{2.226}$$
$$= (I - KH_x)E[\tilde{x}_{k|k-1}\tilde{x}_{k|k-1}^T|Z_{1:k}](I - KH_x)^T + KE[v_kv_k^T|Z_{1:k}]K^T \tag{2.227}$$
$$= (I - KH_x)P_{k|k-1}(I - KH_x)^T + KRK^T \tag{2.228}$$

The expected value of $\tilde{x}$ is zero, the cross-terms of the expansion are therefore also zero [35]. Using a calculus and some algebraic modifications we receive the Kalman gain K [35]:

$$S = H_xP_{k|k-1}H_x^T + V_vRV_v^T \tag{2.229}$$
$$K = P_{k|k-1}H_x^TS^{-1} \tag{2.230}$$

Substituting the equation

$$P_{k|k-1}H_x^T = KS \tag{2.231}$$

into the equation 2.228 gives us the posterior covariance matrix [35]:

$$P_{k|k} = P_{k|k-1} - KSK^T \tag{2.232}$$

The jacobian matrix V_v is defined as [4]:

$$V_v = \frac{\partial f}{\partial v} \tag{2.233}$$

2.11.3 Equations

The extended Kalman filter allows to estimate non-linear relationships [4]. Similar to the linear Kalman filter it consists of a time and measurement update.

Time update equations (predictor) [4]:

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k) \tag{2.234}$$
$$P_{k|k-1} = F_x P_{k-1|k-1} F_x^T + W_w Q W_w^T \tag{2.235}$$

The time update of the extended Kalman filter projects the state and covariance estimates from the previous time step $k-1$ to the current time step k. The function f represents the state vector approximation. F_x and W_v are the process Jacobian matrices and Q the process noise at time step $k-1$. [4]

Measurement update equations (corrector) [4]:

$$S = H_x P_{k|k-1} H_x^T + V_v R V_v^T \tag{2.236}$$
$$K = P_{k|k-1} H_x S^{-1} \tag{2.237}$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K(z_k - h(\hat{x}_{k|k-1})) \tag{2.238}$$
$$P_{k|k} = P_{k|k-1} - KSK^T \tag{2.239}$$

The measurement update corrects the state and covariance estimates with the measurement z_k. The function h represents the measurement approximation, H_x and V_v the measurement Jacobian matrices and R the measurement noise at time step k. [4]

2.12 Error-State Kalman Filter

2.12.1 Introduction

The Error-state Kalman Filter (ESKF) utilizes dead-reckoning to determine the positioning by integrating accelerometer and gyrometer readings of the IMU. The readings are fused with absolute positioning data, such as vision tracking data, to avoid a drift with time caused by accumulated integration errors of IMU readings. [43]

The ESKF has the following properties:

- It relies on a minimal error-state to avoid issues related to over-parameterization and the risk of singularity of the involved covariance matrices [43].

- "The error-state is always operating close to the origin" [43], thus preventing possible parameter singularities, gimbal lock issues and it guarantees the validity of the linearization [43].
- "The error-state is always small, meaning that all second-order products are negligible. This makes the computation of Jacobians very easy and fast." [43].
- The procedure to apply Kalman filter corrections, in order to observe the errors, is slow because all the large-signal dynamics have to be integrated in the nominal-state, but the corrections are applied at a lower rate than the predictions [43].

2.12.2 Process Description

The ESKF distinguishes between true-, nominal- and error-state values. The true-state x_t is a composition of the nominal-state x and error-state δx. The nominal-state is considered as large-signal (non-linear integrable) and the error-state as small-signal (linear integrable). High frequency IMU data u_m is integrated into the nominal-state x, while the noise terms w and other possible model imperfections are not considered, which leads to the accumulation of errors. These accumulated errors are collected in the error-state δx and estimated with the ESKF, to incorporate all the noise w and perturbations i. The error-state estimates a discrete time-varying process, consists of small-signal magnitudes and its process is governed by the linear system dynamic

$$\delta x \leftarrow f(x, \delta x, u_m, i) = F_x(x, u_m)\ \delta x + F_i\ i, \tag{2.240}$$

where F_x and F_i are the Jacobians of f in respect to the error and perturbations vectors. Note that $x \leftarrow f(x, \bullet)$ stands for a discrete time update of $x_k = f(x_{k-1}, \bullet_{k-1})$. The ESKF uses values of the nominal-state to compute the dynamic, control and measurement matrices and predicts the Gaussian estimate of the error-state. At this stage the ESKF only predicts, because no absolute measurement is available to correct the estimate. The observation of the true-state by the measurement

$$z = h(x, v), \tag{2.241}$$

allows to correct the filter state upon the availability of absolute positioning information. This is happening at a lower rate then integration phase. The function h of the system state is generally nonlinear and v is a Gaussian noise $v \sim \mathcal{N}\{0, V\}$ with a covariance V. The measurement renders the error observable and provides a posterior Gaussian estimate of the error-state. Next, the error-state's mean is injected into the nominal-state and reset to zero. Additionally the corresponding covariance matrices are updated to reflect the reset. This process repeats itself with each newly available IMU reading or observation. [43]

2.12.3 Variables of the ESKF

All the involved ESKF variables are summarized in table 3.2. We use angular rates that are locally defined with respect to the nominal quaternion. This means that the

Magnitude	True	Nominal	Error	Composition	Measured	Noise
Full state [1]	x_t	x	δx	$x_t = x \oplus \delta x$		
Position	p_t	p	δp	$p_t = p + \delta p$		
Velocity	v_t	v	δv	$v_t = v + \delta v$		
Quaternion	q_t	q	δq	$q_t = q \otimes \delta q$		
Rotation Matrix	R_t	R	δR	$R_t = R\,\delta R$		
Angles vector			$\delta\theta$	$\delta q = e^{\delta\theta/2}$ $\delta R = e^{[\delta\theta]_\times}$		
Accelerometer bias	a_{bt}	a_b	δa_b	$a_{bt} = a_b + \delta a_b$		a_ω
Gyrometer bias	ω_{bt}	ω_b	$\delta\omega_b$	$\omega_{bt} = \omega_b + \delta\omega_b$		ω_w
Gravity vector	g_t	g	δg	$g_t = g + \delta g$		
Acceleration	a_t				a_m	a_n
Angular rate	ω_t				ω_m	ω_n

[1] The symbol $\oplus$ stands for a generic composition.

Table 2.1: Variables of the ESKF [43]

gyrometer measurements ω_m can be used directly, because the provided angular rates are referenced in the body-frame and the angular error $\delta\theta$ is therefore also defined locally [43].

2.12.4 System Kinematics in Continuous Time

This section outlines the kinematic equations for continuous time (see [43] for proof).

We define a kinematic system

$$x'_t = f_t(x_t, u_m, w), \tag{2.242}$$

with a state x_t, noisy IMU readings u_m and a perturbing white Gaussian noise w,

$$x_t = \begin{bmatrix} p_t \\ v_t \\ q_t \\ a_{bt} \\ \omega_{bt} \\ g_t \end{bmatrix}, \quad u_m = \begin{bmatrix} a_m - a_n \\ \omega_m - \omega_n \end{bmatrix}, \quad w = \begin{bmatrix} a_w \\ \omega_w \end{bmatrix}. \tag{2.243}$$

[43]

The true-state kinematics for this system are given by [43]:

$$p'_t = v_t \tag{2.244}$$
$$v'_t = R_t(a_m - a_{bt} - a_n) + g_t \tag{2.245}$$
$$q'_t = \frac{1}{2} q_t \otimes q(\omega_m - \omega_{bt} - \omega_n) \tag{2.246}$$
$$a'_{bt} = a_w \tag{2.247}$$
$$\omega'_{bt} = \omega_w \tag{2.248}$$
$$g'_t = 0 \tag{2.249}$$

Note that the system estimates the gravity vector g_t, for simplicity an initial orientation of $q_0 = (1, 0, 0, 0)$ is usually taken and the initial rotation matrix is therefore $R_0 = R\{q_0\} = I$. [43]

Nominal-State Kinematics

The nominal-state kinematics ignore noise and perturbations and are given by [43]:

$$p' = v \tag{2.250}$$
$$v' = R(a_m - a_b) + g \tag{2.251}$$
$$q' = \frac{1}{2} q \otimes (\omega_m - \omega_b) \tag{2.252}$$
$$a'_b = 0 \tag{2.253}$$
$$w'_b = 0 \tag{2.254}$$
$$g' = 0 \tag{2.255}$$

Error-State Kinematics

The linearized error-state dynamics can be determined by solving each composite equation of the table 3.2 for the error-state and simplifying all second order infinitesimals [43]:

$$\delta p' = \delta v \tag{2.256}$$
$$\delta v' = -R[a_m - a_b]_\times \delta\theta - R\delta a_b + \delta g - R a_n \tag{2.257}$$
$$\delta\theta' = -[\omega_m - \omega_b]_\times \delta\theta - \delta\omega_b - \omega_n \tag{2.258}$$
$$\delta a'_b = a_\omega \tag{2.259}$$
$$\delta\omega'_b = a_w \tag{2.260}$$
$$\delta g' = 0 \tag{2.261}$$

2.12.5 System Kinematics in Discrete Time

The differential equations of the system kinematics in continuous time need to be integrated into difference equations to account for discrete time intervals $\Delta t > 0$. The

integration has to be done for the subsystem of the nominal-state and the error-state. The error-state consists of a deterministic part, incorporating the state and control dynamics, and a stochastic part, incorporating noise and perturbations. [43]

The nominal-state vector x, the error-state vector δx, the input vector u_m and the perturbation impulses vector i for these equations are defined as [43]:

$$x = \begin{bmatrix} p \\ v \\ q \\ a_b \\ \omega_b \\ g \end{bmatrix}, \quad \delta x = \begin{bmatrix} \delta p \\ \delta v \\ \delta\theta \\ \delta a_b \\ \delta\omega_b \\ \delta g \end{bmatrix}, \quad u_m = \begin{bmatrix} a_m \\ \omega_m \end{bmatrix}, \quad i = \begin{bmatrix} v_i \\ \theta_i \\ a_i \\ \omega_i \end{bmatrix} \tag{2.262}$$

Nominal-State Kinematics

The difference equations for the nominal-state are given by [43]:

$$p \leftarrow p + v\Delta t + \frac{1}{2}R((a_m - a_b) + g)\Delta t^2 \tag{2.263}$$
$$v \leftarrow v + (R(a_m - a_b) + g)\Delta t \tag{2.264}$$
$$q \leftarrow q \otimes q\{(\omega_m - \omega_b)\Delta t\} \tag{2.265}$$
$$a_b \leftarrow a_b \tag{2.266}$$
$$\omega_b \leftarrow \omega_b \tag{2.267}$$
$$g \leftarrow g. \tag{2.268}$$

The rotation matrix $R \triangleq R\{q\}$ represents the rotation of the current nominal orientation q and $q\{r\}$ represents the quaternion of the rotation r, (see 2.67). [43]

Error-State Kinematics

The integration of the error-state's deterministic and the stochastic part (see [43]) results in:

$$\delta p \leftarrow \delta p + \delta v\Delta t \tag{2.269}$$
$$\delta v \leftarrow \delta v + (-R[a_m - a_b]_\times \delta\theta - R\delta a_b + \delta g)\Delta t + v_i \tag{2.270}$$
$$\delta\theta \leftarrow R^T\{\omega_m\delta t\}\delta\theta - \delta\omega\Delta t + \theta_i \tag{2.271}$$
$$\delta a_b \leftarrow \delta a_b + a_i \tag{2.272}$$
$$\delta\omega_b \leftarrow \omega_b + \omega_i \tag{2.273}$$
$$\delta g \leftarrow \delta g \tag{2.274}$$

The random impulses v_i, θ_i, a_i and ω_i "are applied to velocity, orientation and bias estimates and are modeling white Gaussian processes" [43]. They have a mean of zero

"and their covariance matrices are obtained by integrating the covariances of a_n, ω_n, a_w and ω_w over the step time Δt" [43], which gives us:

$$V_i = \sigma_{a_n}^2 \Delta t^2 I \tag{2.275}$$
$$\Theta_i = \sigma_{\omega_n}^2 \Delta t^2 I \tag{2.276}$$
$$A_i = \sigma_{a_\omega}^2 \Delta t I \tag{2.277}$$
$$\Omega_i = \sigma_{\omega_\omega}^2 \Delta t I \tag{2.278}$$

The covariances $\sigma_{a_n}^2$, $\sigma_{\omega_n}^2$, $\sigma_{a_\omega}^2$ and $\sigma_{\omega_\omega}^2$ can be obtained from the IMU datasheet or by measurements [43].

2.12.6 Procedure Description

As explained in detail in section 2.12.2 the ESKF makes predictions using IMU data and corrects these predications with absolute positioning data [43]. The whole procedure consists of the following steps [43]:

1. Predication of the error-state δx by incorporating all the noise and perturbations.
2. Observation of the error-state by calculating the error via filter correction.
3. Injecting the observed error into the nominal-state.
4. Reset the error-state.

Prediction of the Error-State

Following the definition of the error-state equation 2.240 the ESKF prediction equations are given by[43]:

Prediction of the error-state

$$\delta\hat{x} \leftarrow F_x(x, u_m)\, \delta\hat{x} \tag{2.279}$$
$$P \leftarrow F_x P F_x^T + F_i Q_i F_i^T \tag{2.280}$$

The error-state δx is a Gaussian random variable with $\delta x \sim \mathcal{N}\{\delta\hat{x}, P\}$. F_x and F_i are the Jacobian matrices of f with respect to the error and perturbation vectors. The covariance matrix of the perturbation impulses is denoted as Q_i. [43]

The Euler form of the system's transition matrix F_x is given by [43]:

$$F_x = \frac{\partial f}{\partial \delta x}\Big|_{x,u_m} = \begin{bmatrix} I & I\Delta t & 0 & 0 & 0 & 0 \\ 0 & I & -R[a_m - a_b]_\times \Delta t & -R\Delta t & 0 & I\Delta t \\ 0 & 0 & R^T\{(\omega_m - \omega_b)\Delta t\} & 0 & -I\Delta t & 0 \\ 0 & 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & 0 & I \end{bmatrix} \tag{2.281}$$

All the state-related values are directly extracted from the nominal-state. Since the mean error δx is initialized with zero, the linear equation 2.279 will always return zero. Its calculation can therefore be omitted. [43]

The matrices F_i and Q_i are defined as [43]:

$$F_i = \frac{\partial f}{\partial i}\Big|_{x,u_m} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad Q_i = \begin{bmatrix} V_i & 0 & 0 & 0 \\ 0 & \Theta_i & 0 & 0 \\ 0 & 0 & A_i & 0 \\ 0 & 0 & 0 & \Omega_i \end{bmatrix} \tag{2.282}$$

The term $F_i Q_i F_i^T$ will grow continuously for every prediction step [43].

Observation of the Error-State

Following the definition of the observation equation 2.241 the correction equations are given by [43]:

Observation of the error-state

$$K = PH^T(HPH^t + V)^{-1} \tag{2.283}$$
$$P \leftarrow (I - KH)P \tag{2.284}$$
$$\delta\hat{x} \leftarrow K(z - h(\hat{x}_t)) \tag{2.285}$$
$$\tag{2.286}$$

"The Jacobian matrix H is defined with respect to the error-state δx, and evaluated at the best true-state estimate $\hat{x}_t = x \oplus \delta\hat{x}$." [43] Because the mean of the error-state is zero - since it has not been observed yet - the true-state estimate equals the nominal-state $\hat{x}_t = x$, leading to [43]:

$$H \equiv \frac{\partial h}{\partial \delta x}\Big|_x \tag{2.287}$$

The Jacobian can be computed using the chain rule [43]:

$$H \triangleq \left.\frac{\partial h}{\partial \delta x}\right|_x = \left.\frac{\partial h}{\partial x_t}\right|_x \left.\frac{\partial x_t}{\partial \delta x}\right|_x = H_x X_{\delta x} \tag{2.288}$$

The term H_x is the standard Jacobian of h with respect of the true-state. The second term $X_{\delta x}$ is the Jacobian of the true-state with respect to error-state. This term can be derived using the ESKF composition of states, which results in all identity 3×3 blocks expect the 4×3 Quaternion term. [43]

This leaves us with the form [43]:

$$X_{\delta x} \triangleq \left.\frac{\partial x_t}{\partial \delta x}\right|_x = \begin{bmatrix} I_6 & 0 & 0 \\ 0 & Q_{\delta\theta} & 0 \\ 0 & 0 & I_9 \end{bmatrix} \tag{2.289}$$

Using the chain rule, the equations 2.13 - 2.15 and the limit $\delta q \xrightarrow[\delta\theta \to 0]{} \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$ the quaternion term $Q_{\delta\theta}$ can be derived and is given by [43]:

$$Q_{\delta\theta} = \frac{1}{2}\begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix} \tag{2.290}$$

Injection of the Observed Error

After the observation of the error-state, the nominal-state gets updated using the appropriate compositions $x \leftarrow x \oplus \delta\hat{x}$ (see table 3.2) [43]:

Injection of the error-state

$$p \leftarrow p + \delta\hat{p} \tag{2.291}$$
$$v \leftarrow v + \delta\hat{v} \tag{2.292}$$
$$q \leftarrow q \otimes \{\delta\hat{\theta}\} \tag{2.293}$$
$$a \leftarrow a_b + \delta\hat{a}_b \tag{2.294}$$
$$\omega \leftarrow \omega_b + \delta\hat{\omega}_b \tag{2.295}$$
$$g \leftarrow g + \delta\hat{g} \tag{2.296}$$

Reset of the Error-State

After the injection of the observed error into the nominal-state, the error-state mean $\delta\hat{x}$ gets reset. To complete the ESKF update, the covariance of the error-state needs to be updated accordingly. [43]

We define an error reset function g, so that [43]:

$$\delta x \leftarrow g(\delta x) = \delta x \ominus \delta \hat{x} \tag{2.297}$$

The $\ominus$ stands for the inverse of the composition $\oplus$ [43]. The ESKF error reset operation is than given by [43]:

Reset of the error-state

$$\delta \hat{x} \leftarrow 0 \tag{2.298}$$

$$P \leftarrow GPG^T \tag{2.299}$$

Where the Jocobian matrix G is defined as [43]:

$$G \triangleq \left. \frac{\partial g}{\delta x} \right|_{\delta x} \tag{2.300}$$

Similarly to the Jacobian of the of the true-state with respect to the error-state $X_{\delta x}$, this Jacobian matrix is the identity on all diagonal blocks expect for the orientation error [43]. The derivation of the orientation error block is $\frac{\partial \delta\theta^+}{\partial \delta\theta} = I - \left[\frac{1}{2}\delta\hat{\theta}\right]_\times$, so that [43]:

$$G = \begin{bmatrix} I_6 & 0 & 0 \\ 0 & I - \left[\frac{1}{2}\delta\hat{\theta}\right]_\times & 0 \\ 0 & 0 & I_9 \end{bmatrix} \tag{2.301}$$

For most cases, the error term $\delta\hat{\theta}$ can be neglected, leading to a Jacobian $G = I_{18}$ and to a trivial error reset [43].

2.13 Unscented Kalman Filter

This section provides a brief overview of the Unscented Kalman Filter (UKF), an alternative to the extended Kalman filter. The extended Kalman filter propagates the Probility density function (PDF) by utilizing a linear approximation of the system's function around the operating point at each time instant. The required Jacobian matrices can be difficult to obtain for higher order systems. Furthermore, the linear approximation introduces errors which cause the state to diverge over time. The nonlinear state estimator UKF propagates the PDF in a simple way and can overcome the mentioned drawbacks. The UKF can estimate the mean and covariance up to second order. [25]

2.13.1 Principle

Consider a random vector $x \in \mathbb{R}^n$ and a nonlinear function $g : \mathbb{R}^n \to \mathbb{R}^m$. The goal is to calculate the PDF of y given the PDF g of x, meaning in the case of $g(x)$ being Gaussian to compute the mean $\bar{y}$ and covariance P_y of y. [25]

The extended Kalman filter linearizes a nonlinear function $g(x)$ around the current estimate of x, in order to get the mean and covariance of y. The mean $\bar{y}^{\text{EKF}}$ and covariance P_y^{EKF} of y are given by the mean $\bar{x}$, covariance P_x of the PDF of x, and the Jacobian ∇g of $g(x)$ at $\bar{x}$:

$$\bar{y}^{\text{FKF}} = g(\bar{x}), \quad P_y^{\text{EKF}} = (\nabla g) P_x (\nabla g)^{\text{T}} \tag{2.302}$$

[25]

The discrete process and observation update equations are given by,

$$x_k = f\left(x_{k-1}, v_{k-1}, u_{k-1}\right) \tag{2.303}$$

$$y_k = h\left(x_k, n_k, u_k\right), \tag{2.304}$$

where $x \in \mathbb{R}^{n_x}$ represents the system state, $v \in \mathbb{R}^{n_v}$ stands for the process noise, $n \in \mathbb{R}^{n_n}$ for the observation noise, u for the input and y for the noisy observation of the system's state. [25]

The UKF operates differently by computing the PDF of y given the PDF of x. In the case of a Gaussian, the mean $\bar{y}^{\text{UKF}}$ and covariance P_y^{UKF} of y are determined. The UKF uses a set of sigma points $x^{(i)}, i \in \{1, \ldots, p\}, p = 2n + 1$, where each point has a weight $w^{(i)}$ associated. It propagates each sigma point through the nonlinear function $y^{(i)} = g(x^{(i)})$. The mean is approximated by the weighted average of the transformed points,

$$\bar{y}^{\text{UKF}} = \sum_{i=0}^{p} w^{(i)} y^{(i)}, \Sigma w^{(i)} = 1 \tag{2.305}$$

and the covariance is then computed by the weighted outer product of the transformed points,

$$P_y^{\text{UKF}} = \sum_{i=0}^{P} w^{(i)} (y^{(i)} - \bar{y})(y^{(i)} - \bar{y})^{\text{T}}. \tag{2.306}$$

[25]

2.13.2 Algorithm

The system is represented by the process update equation 2.303 and observation update equation 2.304. Consider an augmented state at the time instant k,

$$x_k^a \triangleq \begin{bmatrix} x_k \\ v_k \\ n_k \end{bmatrix}, \tag{2.307}$$

with the dimensions $N = n_x + n_v + n_n$, and an augmented state covariance matrix,

$$P^a \triangleq \begin{bmatrix} P_x & 0 & 0 \\ 0 & P_v & 0 \\ 0 & 0 & P_n \end{bmatrix}, \tag{2.308}$$

where P_v and P_n denote the process and observation noise covariance matrices. [25]

The algorithm initializes at $k = 0$ with,

$$\widehat{x}_0 = E\left[x_0\right], \quad P_{x_0} = E\left[\left(x_0 - \widehat{x}_0\right)\left(x_0 - \widehat{x}_0\right)^{\mathrm{T}}\right] \tag{2.309}$$

$$\widehat{x}_0^a = E\left[x^a\right] = E\left[\widehat{x}_0 \quad 0 \quad 0\right]^{\mathrm{T}} \tag{2.310}$$

$$P_0^a = E\left[\left(x_0^a - \widehat{x}_0^a\right)\left(x_0^a - \widehat{x}_0^a\right)^{\mathrm{T}}\right] = \begin{bmatrix} P_x & 0 & 0 \\ 0 & P_v & 0 \\ 0 & 0 & P_n \end{bmatrix}, \tag{2.311}$$

and continues for $k = 1, 2, \ldots, \infty$ with [25]:

1. First, $2N + 1$ sigma-points are calculated using the present state covariance,

$$\mathbf{X}_{i,k-1}^a \begin{cases} \triangleq \hat{x}_{k-1}^a, & i = 0, \\ \triangleq \widehat{x}_{k-1}^a + \gamma \mathbf{S}_i, & i = 1, \ldots, N, \\ \triangleq \widehat{x}_{k-1}^a - \gamma \mathbf{S}_i, & i = N+1, \ldots, 2N, \end{cases} \tag{2.312}$$

   where S_i represents the i-th column of the matrix, $S = \sqrt{P_{k-1}^a}$. The a scaling parameter γ is given by the tuning parameters α and κ,

$$\gamma = \sqrt{N + \lambda}, \quad \lambda = \alpha^2(N + \kappa) - N. \tag{2.313}$$

   The sigma point matrix is defined as,

$$\mathbf{X}_{i,k-1}^a = \begin{bmatrix} \mathbf{X}_{i,k-1}^x \\ \mathbf{X}_{i,k-1}^v \\ \mathbf{X}_{i,k-1}^n \end{bmatrix}, \tag{2.314}$$

   where each column i represents a sigma point, and x, v and n refer to a partition of the state, process noise and measurement noise, respectively.

2. Second, for the time-update the sigma points are transformed through the state-update function,

$$\mathbf{X}_{i,k/k-1}^x = f\left(\mathbf{X}_{i,k-1}^x, \mathbf{X}_{i,k-1}^v, u_{k-1}\right), \quad i = 0, 1, \ldots, 2N. \tag{2.315}$$

Calculate the prior state estimate and covariance matrix,

$$\widehat{x}_k^- = \sum_{i=0}^{2N} \left(w_{\mathrm{m}}^{(i)} \mathbf{X}_{i,k/k-1}^x \right), \tag{2.316}$$

$$P_{x_k}^- = \sum_{i=0}^{2N} w_{\mathrm{c}}^{(i)} \left(\mathbf{X}_{i,k/k-1}^x - \widehat{x}_k^- \right) \left(\mathbf{X}_{i,k/k-1}^x - \hat{x}_k^- \right)^{\mathrm{T}}. \tag{2.317}$$

The weights $w_{\mathrm{m}}^{(i)}$ and $w_{\mathrm{c}}^{(i)}$ are given as,

$$w_{\mathrm{m}}^{(0)} = \frac{\lambda}{N+\lambda}, \quad i = 0, \tag{2.318}$$

$$w_{\mathrm{c}}^{(0)} = \frac{\lambda}{N+\lambda} + \left(1 - \alpha^2 + \beta\right), \quad i = 0, \tag{2.319}$$

$$w_{\mathrm{m}}^{(i)} = w_{\mathrm{c}}^{(i)} = \frac{1}{2(N+\lambda)}, \quad i = 1, \ldots, 2N, \tag{2.320}$$

where β represents a non-negative weighting parameter, which affects the weighting of the zeroth sigma-point required for the calculation of the covariance.

3. Finally, the measurement-update transforms the sigma points through the measurement function,

$$\mathbf{Y}_{i,k/k-1} = h\left(\mathbf{X}_{i,k/k-1}^x, \mathbf{X}_{k-1}^n, u_k \right), \quad i = 0, 1, \ldots, 2N \tag{2.321}$$

and the mean and covariance are calculated,

$$\widehat{y_k^-} = \sum_{i=0}^{2N} w_{\mathrm{m}}^{(i)} \mathbf{Y}_{i,k/k-1}, \tag{2.322}$$

$$P_{\bar{y}_k} = \sum_{i=0}^{2N} w_{\mathrm{c}}^{(i)} \left(\mathbf{Y}_{i,k/k-1} - \hat{y}_k^- \right) \left(\mathbf{Y}_{i,k/k-1} - \hat{y}_k^- \right)^{\mathrm{T}}. \tag{2.323}$$

The cross covariance is given by,

$$P_{x_k y_k} = \sum_{i=0}^{2N} w_{\mathrm{c}}^{(i)} \left(\mathbf{X}_{i,k/k-1}^x - \widehat{x}_k \right) \left(\mathbf{Y}_{i,k/k-1} - \widehat{y}_k \right)^{\mathrm{T}}. \tag{2.324}$$

Finally, the Kalman gain is defined as,

$$K_k = P_{x_k y_k} P_{\bar{y}_k}^{-1}, \tag{2.325}$$

and the UKF estimate and covariance are given by,

$$\hat{x}_k = \hat{x}_k^- + K_k \left(y_k - \hat{y}_k^- \right), \tag{2.326}$$

$$P_{x_k} = P_{x_k}^- - K_k P_{\bar{y}_k} K_k^{\mathrm{T}}. \tag{2.327}$$

CHAPTER 3

Design and Methods

This chapter gives on overview of the investigated concepts and design decisions to implement and evaluate a TFF for virtual reality systems. The requirements and the design approach of the TFF, as well as the filter algorithms DESP and ESKF are introduced.

3.1 Requirements

Virtual reality systems should provide an immersive experience for users. In order to provide such an immersive experience, the user's state and interactions need to be accurately measured and represented within the VE, and the related changes need to be traced back to the user's senses. Any error related to measurement, process or feedback impacts this experience.

The following requirements have been considered:

Tracking system: The tracking system should ideally posses the properties introduced in chapter 2.3.1. Properties that should be considered are a high sampling rate to provide smooth tracking, a low latency to prevent nausea, and accuracy as well as robustness to enable an immersive experience without glitches. The system is required to have optical and inertial tracking, to fuse tracking data for improved performance. Additionally tracking of head, hand and feet should be supported.

Implementation: The implementation of the TFF should be done in the form of an independent *C++* library to be performance efficient. It should read tracking data from a source, apply a filter algorithm and provide the result as an output. A filter algorithm needs to be able to smooth positions and rotations, and use quaternions as a representation for rotations. It needs be possible to control a filter algorithm with configurable filter parameters. A Kalman filter algorithm with the ability to fuse sensor data, as well as an exponential filter algorithm has to be implemented.

Evaluation: The performance of the TFF should be evaluated by analyzing its output through graphs and from the user's perspective in the form of a user study. A VE in which the user has to perform various tasks is to be implemented, to test each filter implementation and the frameworks performance. The evaluation should consider quantitative and qualitative data.

3.2 Design

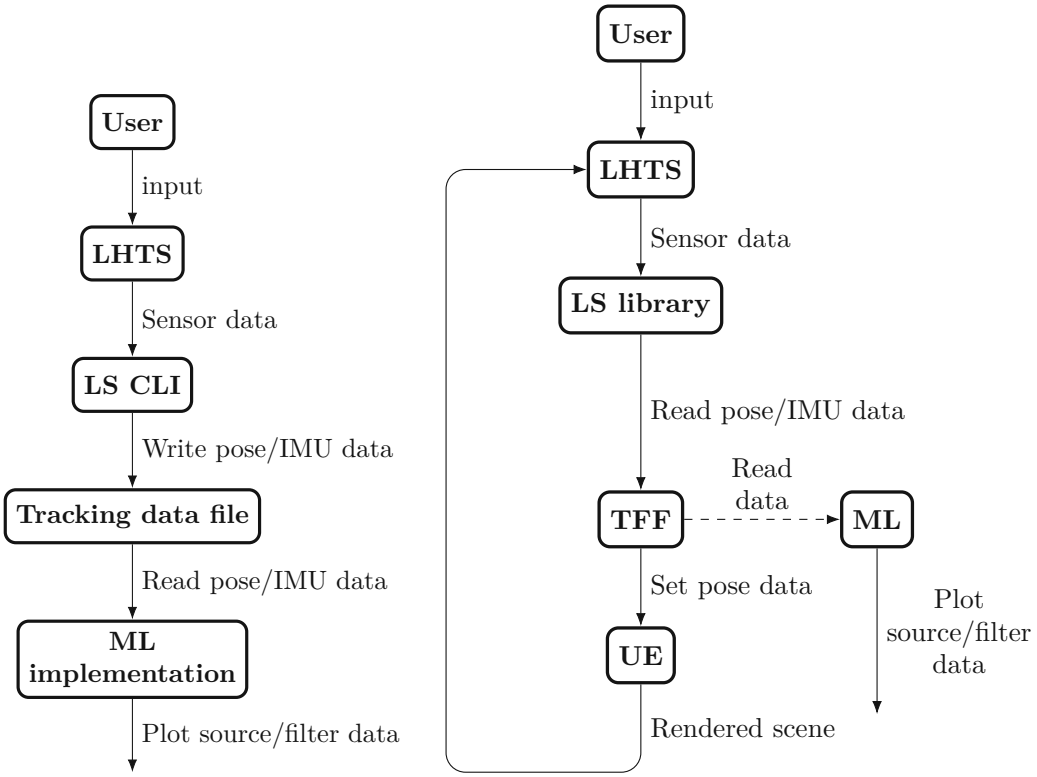
Figure 3.1: Lighthouse 2.0 system [24]

This section outlines the design decisions and gives an overview of this work. The LHTS is chosen as tracking system and the libsurvive library is used to access the tracking data. The filter implementation is done with Matlab and C++, and the integration of the TFF into the UE through a custom plugin that receives the data through websocket. The implementation is evaluated by analyzing its output with Matlab and by conducting a user study at which users carry out tasks for each filter system within a VE.

It follows a description of the individual parts of this work:

Tracking system: The LHTS 2.0 (see chapter 2.3.4) is used as a tracking system for this work. It consists of two tracking base stations, a Head Mounted Display (HMD) to track head movements and display the VE from the user's perspective, and two Knucke Controller (KC) to track hand movements (see figure 3.1). It provides optical (see chapter 2.3.3) and inertial (see chapter 2.3.3) tracking.

libsurvive: The Steam Virtual Reality (SVR) software only provides position and rotation for poses, because of that the open source library libsurvive (see chapter



(a) Matlab filter design (b) Framework filter design

Figure 3.2: Design flow

2.3.5) is used to get the position, rotation, as well the angular velocity and acceleration from the IMU. The libsurvive library uses its own positioning algorithm to estimate position and orientation based on the LHTS lightcap data. It needs to be compiled and configured via parameters to disable it's built-in Kalman filter. It provides two ways to access the tracking data, by parsing the executables standard output or linking the library and using the high or low-level API.

Filter algorithms: The first chosen algorithm is the DESP (see chapter 2.7.4). It smooths the position and orientation of poses delivered by the tracking system, and has the capability to predict in between pose updates. The second algorithm is the ESKF (see chapter 2.12), it smooths and fuses tracking information from two different tracking sources. In theory this should result in a better and more accurate pose prediction. Both algorithms use quaternions as representation for rotations.

TFF: First the filter algorithms are implemented and evaluated with Matlab using tracking data recorded with libsurvive (see figure 3.2a). This makes it possible to gain understanding on the theory and implementation of filter algorithms. It also provides a reference for the TFF's standalone *C++* library implementation.

This library takes libsurvive tracking data as input, applies the filter algorithms and provides the result as an output. Matlab is used as assisting tool to plot and compare libsurvive tracking data with the output of a filter.

UE integration : The filter output is transferred to the UE through websocket, due to compatibility issues of the UE with the C Standard General Utilities Library and the requirement of an independent implementation. A standalone application hosts a websocket server, and links and runs the *C++* filter library. An UE plugin links the websocket client, connects to the websocket server and receives the filter output. This plugin together with a modified version of the open source UE plugin *StreamVR* allows the use of poses for objects within the virtual world from alternative data sources other the SVR (see figure 3.2b).

Evaluation : The evaluation of the TFF covers measurements of important tracking properties, the analysis of the filter outputs with Matlab, and the conduction of a user study. Participants of the user study have to complete tasks within a VE, which is hosted within the Virtual Reality Application (VRA). Qualitative data is collected by asking the users for subject feedback on the filter performance. Quantitative data is collected by recording the time and number of interactions it takes to complete a task.

3.3 Double Exponential Filter

The DESP described in chapter 2.7.4 has been chosen as exponential filter algorithm for this work. The DESP variables are listed in table 3.1. An exponential filter uses weighted averages to smooth observation values (see chapter 2.7.1). The applied weights decrease exponentially over time, meaning the longer an observation lies in the past the less impact it has on the current estimate. The DESP is an extensions of Brown's linear exponential smoothing (see chapter 2.7.2). It uses the parameter α to control the smoothing rate, and the first Sy'_t and second smoothing statistic Sy''_t, representing the weighted average of level and trend from the time series $y_1 \dots y_t$. The parameter h is a factor and stands for the number of Prediction Time Steps (PTS) into the future. The parameter h and booth calculated smoothing statistics are used to predict the forecast $\hat{y}_{t+h}$ of the observed time series. The DESP is able calculate forecasts for position $\hat{p}_{t+h}$ and rotation $\hat{q}_{t+h}$ time series provided by the tracking system.

3.3.1 Parameters

The parameter α controls the rate at which the level and trend for position and rotation is smoothed. The condition $0 \leq \alpha \leq 1$ applies to the smoothing rate, low values result in strong and high values in weak smoothing. The prediction time frame Δt is the time interval between the last measurement t_m and the time of the prediction t_n. The sampling rate f_s stands for the number of observations per second. The parameter h represents the PTS from the last measurement to the time of the prediction.

Variable	Parameter	Measurement	Prediction	First smoothing statistic	Second smoothing statistic
Smoothing rate	α				
PTS	h				
Current time		t_n			
Measurement time		t_m			
Delta time		Δt			
Sampling rate		f_s			
Position		p_t	$\hat{p}_{t+h}$	Sp'_t	Sp''_t
Rotation		q_t	$\hat{q}_{t+h}$	Sq'_t	Sq''_t

Table 3.1: Variables of the DESP

3.3.2 Procedure

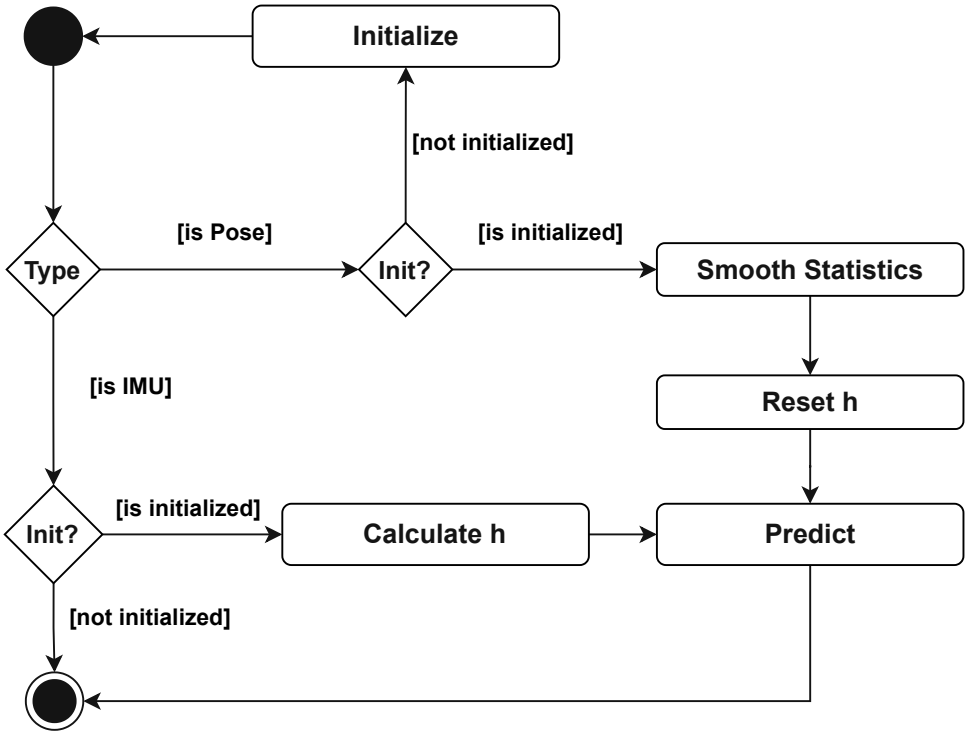


Figure 3.3: DESP procedure workflow

Figure 3.3 illustrates the workflow of the DESP algorithm. The DESP uses the global coordinate system of libsurvive. On the first pose measurement the algorithm initializes the smoothing statistics. At the arrival of new pose measurements the algorithm updates the smoothing statistics and calculates an estimate. On IMU measurements the PTS h

is calculated and used to predict an estimate. This predicted estimate is based on the smoothing statistics, the sampling rate and the time since the last measurement. The procedure is repeated for every new pose and IMU measurement.

It follows a detailed descriptions of the individual steps:

Initialize: Initialize the first and second smoothing statistic on the first pose measurement.

a) Position:

$$Sp'_{t-1} = Sp''_{t-1} = p_t \tag{3.1}$$

b) Rotation:

$$Sq'_{t-1} = Sq''_{t-1} = q_t \tag{3.2}$$

Smooth statistics: Calculate the first and second smoothing statistics for every pose measurement (see chapter 2.7.2).

a) Position:

$$Sp'_t = \alpha p_t + (1-\alpha) Sp'_{t-1} \tag{3.3}$$
$$Sp''_t = \alpha Sp'_t + (1-\alpha) Sp''_{t-1} \tag{3.4}$$

b) Rotation:

$$Sq'_t = \alpha q_t + (1-\alpha) Sq'_{t-1} \tag{3.5}$$
$$Sq''_t = \alpha Sq'_t + (1-\alpha) Sq''_{t-1} \tag{3.6}$$

Reset h: Reset the PTS h and update the last measurement time stamp t_m to the current time t_n on pose measurements:

$$h = 0 \tag{3.7}$$
$$t_m = t_n \tag{3.8}$$

Calculate h: Calculate the time span Δt since the last pose measurement t_m and the PTS h on IMU measurements.

$$\Delta t = t_n - t_m \tag{3.9}$$
$$h = \Delta t f_s \tag{3.10}$$
$$\tag{3.11}$$

Predict: Predict estimates for position and rotation using the smoothing statistics (see chapter 2.7.4).

a) Low and high PTS: Calculate the low $\lfloor h \rfloor$ and high $\lceil h \rceil$ PTS required for the interpolation at the prediction step.

$$\lfloor h \rfloor = floor(h) \tag{3.12}$$
$$\lceil h \rceil = ceil(h) \tag{3.13}$$

b) Position: Calculate the low $\hat{p}_{t+\lfloor h \rfloor}$ and high $\hat{p}_{t+\lceil h \rceil}$ estimates for the position.

$$\hat{p}_{t+\lfloor h \rfloor} = (2 + \frac{\alpha \lfloor h \rfloor}{1-\alpha}) Sp'_t - (1 + \frac{\alpha \lfloor h \rfloor}{1-\alpha}) Sp''_t \tag{3.14}$$
$$\hat{p}_{t+\lceil h \rceil} = (2 + \frac{\alpha \lceil h \rceil}{1-\alpha}) Sp'_t - (1 + \frac{\alpha \lceil h \rceil}{1-\alpha}) Sp''_t \tag{3.15}$$
$$\tag{3.16}$$

The final estimate for the position $\hat{p}_{t+h}$ is given by the linear interpolation of the high and low estimates.

$$\hat{p}_{t+h} = (\hat{p}_{t+\lceil h \rceil} - \hat{p}_{t+\lfloor h \rfloor})(h - \lfloor h \rfloor) + \hat{p}_{t+\lfloor h \rfloor} \tag{3.17}$$

c) Rotation: The low $\hat{q}_{t+\lfloor h \rfloor}$ and high $\hat{q}_{t+\lceil h \rceil}$ estimates for the rotation are given by (note that the estimates have to be normalized):

$$\hat{q}_{t+\lfloor h \rfloor} = \left\| (2 + \frac{\alpha \lfloor h \rfloor}{1-\alpha}) Sq'_t - (1 + \frac{\alpha \lfloor h \rfloor}{1-\alpha}) Sq''_t \right\| \tag{3.18}$$
$$\hat{q}_{t+\lceil h \rceil} = \left\| (2 + \frac{\alpha \lceil h \rceil}{1-\alpha}) Sq'_t - (1 + \frac{\alpha \lceil h \rceil}{1-\alpha}) Sq''_t \right\| \tag{3.19}$$
$$\tag{3.20}$$

The final estimate for the rotation $\hat{q}_{t+h}$ is then given by spherical linear interpolation.

$$\rho = h - \lfloor h \rfloor \tag{3.21}$$
$$\Omega = arccos(\hat{q}_{t+\lfloor h \rfloor} \odot \hat{q}_{t+\lceil h \rceil}) \tag{3.22}$$
$$\hat{q}_{t+h} = \frac{\hat{q}_{t+\lfloor h \rfloor} sin((1-\rho)\Omega) + \hat{q}_{t+\lceil h \rceil} sin(\rho\Omega)}{sin\Omega} \tag{3.23}$$

3.4 Kalman Filter

The ESKF described in chapter 2.12 is used as Kalman filter for this work. For an overview of the filter variables see table 3.2. The Kalman filter is a best practice approach combining several stochastic fundamentals.

The true-state of a system x_t is estimated by the Kalman filter. Each dimension within the true-state represents a continuous random variable with a density function (see chapter 2.8.2). Each of these continuous random variables is a normally distributed

process (see chapter 2.8.9) and can therefore described by their mean and variance (see chapter 2.8.3). The true-state x_t combines multiple continous random variables (see chapter 2.8.4) into a random vector. Joint continuous random variables are also referred to as multivariate (see chapter 2.8.10). A random vector can be represented as a vector of means and a matrix of correlated variances also called covariance matrix (see chapter 2.8.7). The nominal-state x holds the mean and the covariance matrix P the correlated variances of these continuous random variables.

The Kalman filter builds on the state-space-model to keep track of the system dynamics (see chapter 2.9). The state space model allows to model a continuous random process, it assumes that each random process can be described by a difference equation and that the noise present within this process is white, meaning it has a mean of zero and is statistical independent (see chapter 2.8.8). It enables to model the process (predict), the measurement (correct) and noise of continuous random variables.

The Kalman filter applies the *Bayes' theorem* (see chapter 2.8.6) recursively to estimate the posterior, meaning prior knowledge and the measurement is used to predict the posterior estimate. The linear Kalman filter (see chapter 2.10) can only deal with linear processes and the extended Kalman filter (see chapter 2.11) is able to deal with non-linear processes like rotations. The extended Kalman filter linearizes about the current estimate with a Taylor series approximation using partial derivations of the process and measurement function with respect to the state.

The libsurvive framework provides optical pose estimates (position and rotation) and IMU measurements (acceleration and angular velocity). The ESKF (see chapter 2.12) allows to fuse these two different data sources to improve the pose estimation. It uses a nominal-state x for large-signal (non-linear) and an error-state δx (linear) for small-signal system dynamics. The small signal IMU readings are integrated into the nominal-state x during the time-update/prediction phase (see chapter 2.12.5). The large signal estimates position and rotation, allows to observe the error-state δx and correct the nominal-state x during the observation/correction phase.

The nominal-state vector x and the error-state vector δx are defined as:

$$x = \begin{bmatrix} p \\ v \\ q \\ a_b \\ \omega_b \\ g \end{bmatrix}, \quad \delta x = \begin{bmatrix} \delta p \\ \delta v \\ \delta \theta \\ \delta a_b \\ \delta \omega_b \\ \delta g \end{bmatrix} \tag{3.24}$$

3.4.1 Parameter

The observation/correction phase can be controlled by the noise covariance vector $\sigma^2_{p_n}$ and $\sigma^2_{q_n}$, representing the gaussian noise for position and rotation. They determine the

Variable	True	Nominal	Error	Measurement	Noise	Dimensions
Position	p_t	p	δp	p_m	p_n	
Velocity	v_t	v	δv			
Rotation	q_t	q	δq	q_m	q_n	
Angles vector			$\delta\theta$			
Acceleration bias	a_{bt}	a_b	δa_b		a_ω	3×1
Gyrometer bias	ω_{bt}	ω_b	$\delta\omega_b$		ω_ω	
Gravity vector	g_t	g	δg			
Acceleration	a_t			a_m	a_n	
Angular velocity	ω_t			ω_m	ω_n	
Rotation	R_t	R	δR			3×3
State	x_t	x				19×1
Error-state			δx			18×1
Error-State covariance		P				18×18
Nominal noise					V	19×19
Time-update			$f(x, \bullet)$			18×1
Time-update Jacobian			F_x			18×18
Perturbation Jacobian			F_i			18×12
Perturbation noise					Q_i	12×12
True-state Jacobian		H_x				19×19
True-to-error-state q.			$Q_{\delta\theta}$			4×3
True-to-error-state			$X_{\delta x}$			19×18
Measurement Jacobian			H			19×18
Kalman filter gain		K				18×19
Measurement		z				19×1
Measurement		$h(x)$				
Reset Jacobian			G			18×18

Table 3.2: Variables of the ESKF

noise covariance matrix V which impacts the Kalman filter gain K. The gain is used to correct the covariance matrix P and to observe the error-state δx.

The time-update/prediction phase can be controlled by the noise covariance vectors $\sigma^2_{a_n}$, $\sigma^2_{\omega_n}$, $\sigma^2_{a_\omega}$ and $\sigma^2_{\omega_\omega}$, representing the gaussian noise for acceleration, angular velocity, acceleration bias and angular velocity bias. They will be added to the covariance matrix P for each time-update, meaning the covariance matrix will grow continuously with each prediction.

3.4.2 Procedure

Figure 3.4 illustrates the workflow of the ESKF algorithm. The ESKF uses the local coordinate system of libsurvive and translates poses using the translation vector mentioned in chapter 4.2.6. On the first pose measurement the algorithm initializes the state

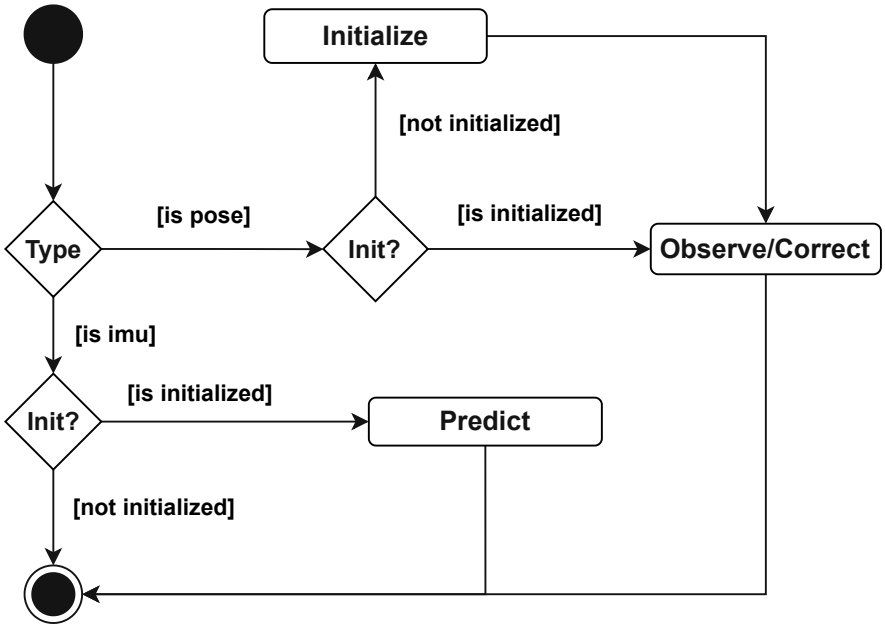


Figure 3.4: ESKF procedure workflow

vectors and covariance matrices. At the arrival of any pose measurement the algorithm observes the error-state, injects it into the nominal-state and performs a reset. On IMU measurements the acceleration and angular velocity are integrated into the nominal-state and the prior covariance matrix is calculated. These steps are repeated for every pose and IMU measurement.

It follows a detailed descriptions of the individual steps:

Initialize: Set the nominal-state x and the covariance matrix P on the first optical tracking update.

$$x = \begin{bmatrix} p_m \\ 0 \\ q_m \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad P = I_{19} \tag{3.25}$$

Predict on IMU updates: Integrate the acceleration a_m and angular velocity ω_m measurement into the nominal-state x and calculate the prior covariance matrix P.

1. Calculate the angular rate ω and the acceleration a by correcting the measurements a_m and ω_m with the biases a_b and ω_b from the nominal-state x

(2.243).

$$a = a_m - a_b, \qquad \omega = \omega_m - \omega_b \tag{3.26}$$

2. Calculate the relative rotation $q\{\omega\Delta t\}$ with the angular velocity ω and the time span Δt since the last IMU measurement.
   a) First calculate the relative angle vector $\Delta\omega$ for the time span Δt, then determine the angle ϕ and the axis vector u:

$$\Delta\omega = \omega\Delta t, \qquad \phi = \|\Delta\omega\|, \qquad u = \frac{\Delta\omega}{\phi} \tag{3.27}$$

   b) Then calculate the relative rotation quaternion with (2.71):

$$\theta = \phi/2, \qquad q\{\omega\Delta t\} = \begin{bmatrix} cos\theta \\ u\ sin(\theta) \end{bmatrix}. \tag{3.28}$$

3. Determine the rotation matrices R and $R\{\omega\Delta t\}$ for the rotation q of the nominal-state x and the relative rotation $q\{\omega\Delta t\}$. Using the rotation matrix equation (2.75)

$$R = (q_w^2 - q_v^T q_v)I + 2q_v q_v^T + 2q_w[q_v]_\times, \tag{3.29}$$

   we can calculate

$$R = R\{q\}, \qquad R\{\omega\Delta t\}. \tag{3.30}$$

4. Integrate the IMU measurements into the nominal-state (2.12.5)

$$x = \begin{bmatrix} p + v\Delta t + \frac{1}{2}(Ra + g)\Delta t^2 \\ v + (Ra + g)\Delta t \\ q \otimes q\{\omega\Delta t\} \\ a_b \\ \omega_b \\ g \end{bmatrix} \tag{3.31}$$

5. Determine the skew matrix $[a]_\times$ for the acceleration a (2.16).

$$[a]_\times = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \tag{3.32}$$

6. Determine the state transition Jacobian matrix F_x of f with respect to the error-state δx using the skew matrix $[a]_\times$, the rotation matrices R, $R\{\omega\Delta t\}$ and the time span Δt (2.281).

$$F_x = \begin{bmatrix} I & I\Delta t & 0 & 0 & 0 & 0 \\ 0 & I & -R[a]_\times\Delta t & -R\Delta t & 0 & I\Delta t \\ 0 & 0 & R^T\{\omega\Delta t\} & 0 & -I\Delta t & 0 \\ 0 & 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & 0 & I \end{bmatrix} \tag{3.33}$$

7. The Jacobian matrix F_i of f with respect to the perturbation vector i is given as (2.262, 2.282):

$$F_i = \begin{bmatrix} 0 & 0 & 0 & 0 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.34}$$

8. The covariance vectors V_i,Θ_i,A_i and Ω_i are obtained by integrating the covariance of their respective noise a_n, ω_n, a_ω and ω_ω over the time span Δt. The covariances can be obtained by the IMU datasheet or measured while the system is still. In our case the acceleration noise a_n and the angular velocity noise ω_n are provided by the libsurvive tracking system for each device (2.275).

$$V_i = \sigma_{a_n}^2 \Delta t^2 I \tag{3.35}$$
$$\Theta_i = \sigma_{\omega_n}^2 \Delta t^2 I \tag{3.36}$$
$$A_i = \sigma_{a_\omega}^2 \Delta t I \tag{3.37}$$
$$\Omega_i = \sigma_{\omega_\omega}^2 \Delta t I \tag{3.38}$$

9. The covariance matrix Q_i of the perturbation vector i is defined as (2.282):

$$Q_i = \begin{bmatrix} V_i & 0 & 0 & 0 \\ 0 & \Theta_i & 0 & 0 \\ 0 & 0 & A_i & 0 \\ 0 & 0 & 0 & \Omega_i \end{bmatrix} \tag{3.39}$$

10. Finally we can update the covariance matrix P with with the Jabobian matrix F_x and add the process noise covariance with the Jacobian matrix F_i and the covariance matrix Q_i (2.280).

$$P = F_x P F_x^T + F_i Q_i F_i^T \tag{3.40}$$

Observe on optical tracking updates: Observe the error-state δx by incorporating the position p_m and rotation q_m measurement. Inject the error-state δx into the nominal-state x and perform a reset.

1. Derive the Jacobian matrix H_x of h with respect to the true-state x_t (2.211).

$$H_x = \frac{\partial h}{\partial x_t} = \begin{bmatrix} I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.41}$$

2. Calculate quaternion term $Q_{\delta\theta}$ of the Jacobian matrix $X_{\delta x}$ with the measured rotation q_m (2.290):

$$Q_{\delta\theta} = \frac{1}{2}\begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix} \tag{3.42}$$

3. Determine the Jacobian matrix $X_{\delta x}$ of the true-state x_t with respect to the error-state δx defined as (2.289):

$$X_{\delta x} = \begin{bmatrix} I_6 & 0 & 0 \\ 0 & Q_{\delta\theta} & 0 \\ 0 & 0 & I_9 \end{bmatrix} \tag{3.43}$$

4. Calculate the Jacobian matrix H of h with respect to the error-state δ (2.288):

$$H = H_x X_{\delta x} \tag{3.44}$$

5. Determine observation noise covariance matrix V with the observation noise p_n and q_n for the position and rotation.

$$V = \begin{bmatrix} \sigma^2_{p_n} & 0 & 0 & 0 & 0 & 0 \\ {\scriptstyle 3\times 3} & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma^2_{q_n} & 0 & 0 & 0 \\ & & {\scriptstyle 4\times 4} & & & \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.45}$$

6. Calculate the Kalman filter gain K with the covariance matrix P, the Jabobian matrix H and the observation noise covariance matrix V (2.283).

$$K = PH^T(HPH^T + V)^{-1} \tag{3.46}$$

7. Correct the covariance matrix P with the Jacobian matrix H and the filter gain K (2.284).

$$P = (I - KH)P \tag{3.47}$$

8. Set the measurement vector z and the apply the measurement function h.

$$z = \begin{bmatrix} p_m \\ 0 \\ q_m \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad h(\hat{x}_t) = h(x) = \begin{bmatrix} p \\ 0 \\ q \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.48}$$

9. Calculate the observed error-state δx with the filter gain K, the measurement vector z and the measurement function h (2.285).

$$\delta\hat{x} = K(z - h(\hat{x}_t)) \tag{3.49}$$

10. Inject the observed error-state δx into the nominal-state x (2.12.6).

$$x = x \oplus \delta\hat{x} = \begin{bmatrix} p + \delta\hat{p} \\ v + \delta\hat{v} \\ q \otimes \{\delta\hat{\theta}\} \\ a_b + \delta\hat{a}_b \\ \omega_b + \delta\hat{\omega}_b \\ g + \delta\hat{g} \end{bmatrix} \tag{3.50}$$

11. Determine the skew matrix $[]_\times$ of the term $1/2\,\delta\hat{\theta}$ and the Jacobian matrix G of the reset function g in respect to the nominal-state x (2.301).

$$G = \begin{bmatrix} I_6 & 0 & 0 \\ 0 & I - \left[\frac{1}{2}\delta\hat{\theta}\right]_\times & 0 \\ 0 & 0 & I_9 \end{bmatrix} \tag{3.51}$$

12. Reset the covariance matrix P with the reset covariance matrix G and set the error-state δx to zero (2.298, 2.299).

$$\delta\hat{x} = 0 \tag{3.52}$$

$$P = GPG^T \tag{3.53}$$

CHAPTER 4

Implementation and Evaluation

This chapter documents the system design, and the implementation and evaluation of the TFF.

4.1 System Design

This section gives an overview of the TFF's implementation, it describes selected hard- and software technologies, experimental setup, the project structure, dependencies, used tools and frameworks.

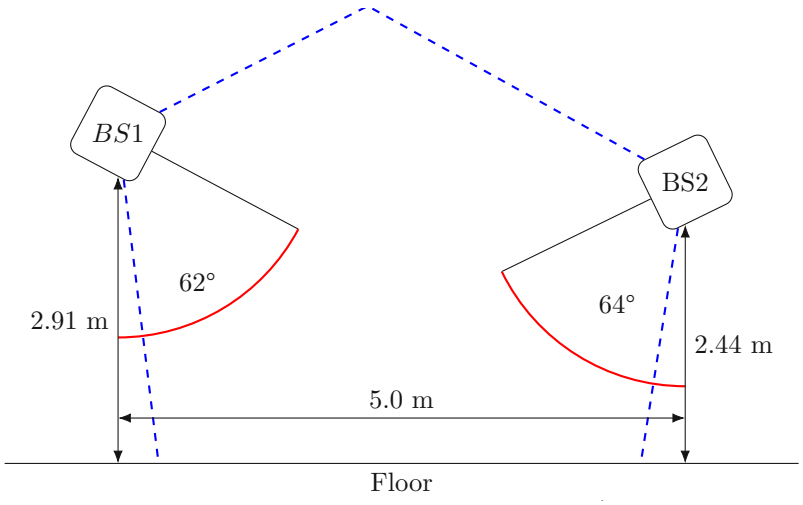


Figure 4.1: LHTS 2.0 base station setup (BS = Base station).

4.1.1 Experimental Setup

The experimental room setup for the Lighthouse tracking base stations is illustrated in figure 4.1. The LHTS setup consists of two base station, the tracked HMD and the two KC devices. Each base station has a 150-degree horizontal field of view and a 110-degree vertical field of view. The two base stations are placed facing each other 5 m apart. The first base station is positioned at a height of 2.91 m and an angle of approximately 62° to the wall. The second one at a height of 2.44 m and an angle of 64°.

The computer specifications used for the implementation and evaluation of the TFF are as follows:

CPU	AMD Ryzen 7 5800X 8-Core Processor
Mainboard	MSI MEG X570 UNIFY (MS-7C35)
Memory	G.Skill Trident Z Neo DIMM Kit 32GB, DDR4-3600, CL16-19-19-39, 1799.6 MHz
Graphics	GIGABYTE Aorus GeForce RTX 3080 Master 10G
Harddisk	GIGABYTE Aorus NVMe Gen4 SSD 1TB
Power supply	be quiet! Straight Power 11 850W ATX 2.4
Operating system	Microsoft Windows 10 (10.0) Professional 64-bit (Build 19043)

SVR is part of the *Steam* software and can be download and installed from here `https://store.steampowered.com/about/`. The hardware and firmware revisions of the SVR LHTS 2.0 used for this work are:

Base station	Valve SR Imp
	Hardware Rev.: Valve Corp. 1004, 0.0, FB02922CF8 V001017-20.A
	Firmware: 3008
Headset	Index Valve
	Hardware Rev.: product 34 rev 21.65.9 lot 2000/0/0 0
	Firmware: 1601324091/1623823641
Controller	Knuckles Valve
	Hardware Rev.: product 17 rev 14.0.9 lot 2020/7/3 0
	Firmware: 1562916277/0 (2019-07-12)
SteamVR	Version 1.20.4 (1634602223)

4.1.2 System Architecture

The TFF consists of two applications, the websocket server (*TFF.Server*) and the VRA (*TFF.VRA*). Both are standalone executables and depend on other components as illustrated in figure 4.2. The application *TFF.VRA* is built with the UE and is implemented in *C++* and the visual scripting language Blueprint. The application uses a modified version of the *SteamVR* plugin, which utilizes the *TFF.Plugin*, both plugins are built with the UE and implemented in *C++*. *TFF.Plugin* uses the websocket client library *TFF.Client*. The websocket client relies on the libraries *TFF.Websocket*, *TFF.Shared* and the external framework *boost*. The application *TFF.Server*, and the libraries *TFF.Client*, *TFF.Process*, *TFF.Websocket* and *TFF.Shared* are all implemented

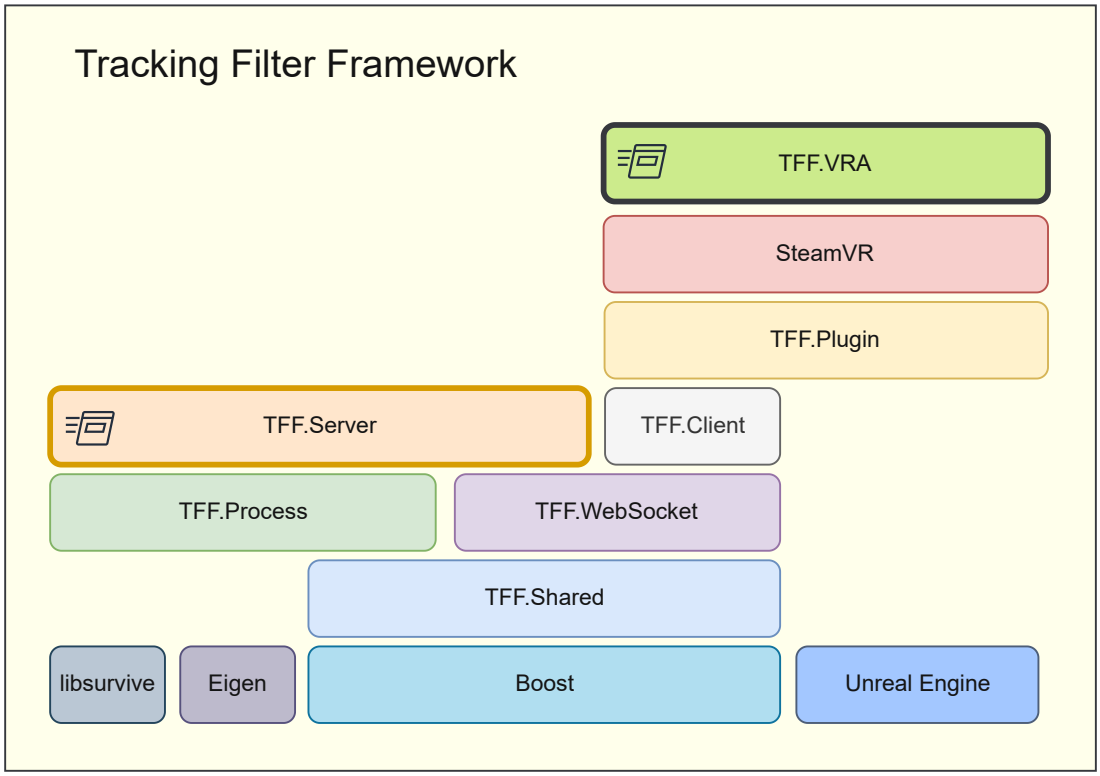


Figure 4.2: TFF applications and dependencies

in *C++* and built using *CMake* on Windows. The *TFF.Server* utilizes the libraries *TFF.Filter*, *TFF.Websocket*, *TFF.Shared*, as well as the external dependencies libsurvive, *Eigen* and *boost*.

4.1.3 Tracking Data Processing Workflow

The developed TFF prototype measures user movements, processes them through filters and feeds the processed data back to the user in the form of an updated scene on the HMD display. This processing sequence is illustrated in figure 4.3.

4.1.4 Applications and Components

All listed applications and components below, with the exception of the SteamVR plugin, have been implemented with the help of third party libraries by me. The SteamVR plugin has been copied and modified, see section 4.5 for details.

TFF.VRA: This is the prototype of the VRA, it enables to use SVR with an alternative tracking data source, receives tracking data via websocket and hosts the VE that is used to conduct the user study. It manages user interaction with the virtual world,

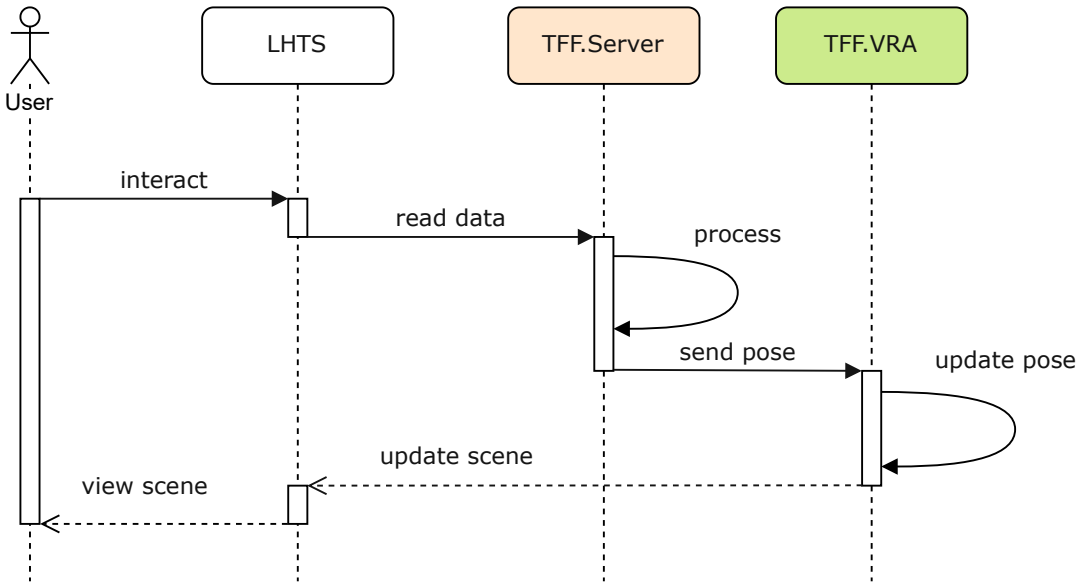


Figure 4.3: TFF processing sequence
The application *TFF.Server* captures user interactions from the LHTS with libsurvive, processes the data through filter algorithms and sends the new pose estimate to the *TFF.VRA*. The application *TFF.VRA* updates the pose, renders and updates the scene, and in the end the LHTS views the updated scene to the user through the HMD.

updates poses of tracked objects based on received tracking data and renders the scene shown within the HMD.

StreamVR: This unreal plugin is part of the UE and is copied to the VRA project and modified in order to use tracking poses from other sources then SVR. It is loaded at runtime of the VRA, retrieves TFF poses through the unreal plugin *TFF.Plugin* and applies them to the tracked HMD and KC's.

TFF.Plugin: This unreal plugin is loaded at runtime of the VRA, it loads the websocket library *TFF.Client* and uses the websocket client to connect to and receive tracking data from the websocket server application *TFF.Server*. It also prints transmission metrics generated by the *TFF.Client* to the *Unreal* log.

TFF.Client: This library contains a websocket client that connects to and receives tracking data from the websocket server, and calculates transmissions metrics of received updates.

TFF.Server: This standalone prototype application opens a websocket server and accepts connections from clients. It initializes and runs the tracking source, which is libsurvive in our case, reads and processes the tracking data through filter algorithms and pushes the updates to all connected clients that have subscribed for

tracking data. The applications runs three separate threads, the first thread gets data from the tracking source, the second processes the data through filters and the third sends them to connected clients. The application can be controlled though Command Line Interface (CLI) parameters and commands sent by a websocket client.

TFF.Process: This library reads data from a tracking source, processes the data through filters and provides the result as output. It contains a dummy or passthrough filter implementation, and the DESP and ESKF filter implementations.

TFF.Websocket: This library holds the shared websocket implementations for the *FF.Server* and the *TFF.Client*. It contains the functionality to open a websocket server, connect with a client and transmit data in the form of JavaScript Object Notation (JSON) strings.

TFF.Shared: This library contains shared implementations for tasks like concurrency, analysis and reporting.

4.1.5 Frameworks, tool and libraries

This section describes all 3rd party frameworks, tools and libraries used for this work.

Eigen: A C++ template library for linear algebra, it provides a rich set of tools for matrix and vector computation. It is an open-source library and released under the Mozilla Public License 2.0. It can handle complex and high-performance operations efficiently, and provides data structures and functionality to operate with vectors, matrices and quaternions. The version *3.3.9* is used for this work to implement the filter algorithms.

boost: A C++ set of libraries that provide structures and tools for linear algebra, logging, pseudorandom number generation, multithreading, regular expression and many more. It is portable across different platforms operating systems and hardware architectures. Boost is open-source, peer-reviewed, considered to be very stable, and published under the Boost Software License. This work uses the version *1.75.0* and uses the features logs, json, unit tests, regex patterns, date time, filesystem, threading, websocket communication, serialization and program arguments.

Matlab: Matlab is an numerical computing environment and programming language. It is used for data analysis, visualization and algorithm development and provides a wide range mathematical functions such as linear algebra, statistics, calculus, fourier analysis, interpolation, extrapolation and differential equations. Matlab was utilized for this work to process libsurvive tracking data, prototype filter algorithms and create visualizations of tracking data through various plots.

UE: The UE is a powerful game development framework and tool that allows to create interactive 3d applications. It provides support for virtual reality and augmented reality development, allowing developers to create immersive, interactive experiences for users. The engine is built on top of *C++* and features the visual scripting language Blueprint that allows to quickly prototype complex game logic without having to write code. The version *5.0.2* has been used to implement the VRA and conduct the user study.

libsurvive: An open-source library to access tracking data of LHTS systems (see chapter 2.3.5).

CMake: A cross-platform, open-source build system, used to control the software compilation process with platform- and compiler independent configuration files. CMake provides a set of built-in commands for performing common build tasks, such as compiling source code, linking libraries, and installing files and allows to find and use external libraries, making it easy to incorporate third-party code into projects. It is used in conjunction with native build environments. The application *TFF.Server* and the libraries *TFF.Client*, *TFF.Process*, *TFF.Websocket*, *TFF.Shared* were built using Cmake the build system *Ninja*.

4.2 libsurvive

This section introduces the external library libsurvive used as tracking data source for this work. Specifically the build steps for the version used, the libsurvive CLI and API, and provided tracking properties are documented. It is an experimental library and is used for this work to access not only position and orientation, but also acceleration and angular velocity of the IMU for each tracked device of the LHTS. The library needs to be built in order to use its executables and DLLs. This work relies on the quality of the data provided by libsurvive. The library uses a complex built-in Kalman filter with multiple states and an iterative view-point optimization algorithm, which allows libsurvive to filter raw light cap data before the poser estimates a position and rotation of a tracked device. For this work the libsurvive Kalman filter, as well as the use of IMU data for pose estimation is disabled, which removes all those optimizations leading to noisy data and therefore more inaccurate poses.

4.2.1 Build

libsurvive is an experimental library and due to problems with crashes a specific version, which turned out to be the most stable, was used. The libsurvive version used for this work can be built using the following steps:

- Clone libsurvive repository and change directory:

```
git clone https://github.com/cntools/libsurvive.git
cd .\libsurvive\
```

- Checkout specific revision:

  `git checkout d47c15a6a788e50092e936224332ddd6b911c216`

- Create the project and start Visual Studio:

  `.\make.ps1`
  `.\build-win\libsurvive.sln`

- Build solution:

  Run *Build → Build solution* **in Visual Studio**

- After the build the executables can be found at:

  `.\build-win\Release`

- The libraries can be found at:

  `.\build-win\src\Release`

4.2.2 Parameters

libsurvive can be configured using parameters, it follows a short introduction of the parameters used for this work:

force-calibrate: Forces calibration and resets the origin of the coordinate system to the current position of the head mounted display.

record-stdout: Prints the tracking data to stdout, this is used for testing and debugging.

use-imu: Determines if the IMU data is used as part of the pose solver. This parameter is disabled for this work, in order to verify the impact of TFF algorithms.

use-kalman: Determines if the libsurvive kalman filter is used as part of the pose solver. This parameter is disabled for this work, in order to verify the impact of TFF algorithms.

record: Records the tracking data to a text file. This parameter is used to record a tracking scenario.

playback: Playback a recorded tracking sample file. This parameter is used to playback tracking scenarios.

4.2.3 Executable and Tracking Sample

The libsurvive executable `survive-cli.exe` provides a direct way to access tracking data through its CLI. Tracking data can be written to the standard output or a text file. This allows to create tracking samples by recording a track scenario. The recorded tracking samples are used to implement and test filter algorithms. A libsurvive tracking sample contains configured options, as well as light cap, IMU, device, pose and velocity data. Each tracking property entry contains a time stamp, device identifier and the property values. The values of an entry are separated with a white space. The listing 4.1 shows a tracking sample with an IMU, pose and velocity entry.

```
12.262440 T20 i 3 2230941945 -5544.000000 -2624.000000 5980.000000
    -826.000000 -143.000000 1000.000000  0.000000 0.000000 0.000000 230
12.262461 T20 POSE 1.084901 1.378156 0.606235 0.344806 0.900690 -0.255425
    0.068010
12.262465 T20 VELOCITY -0.001734 0.210562 -0.012106 -0.071972 0.004855
    0.368428
```

Listing 4.1: libsurvive tracking sample

4.2.4 API

The libsurvive's Dynamic Link Library (DLL) must be loaded and linked at runtime to access its API. libsurvive provides a high-level and a low-level API. This work uses the low-level API because provides access to IMU data. Listing 4.2 shows a *C++* code example that demonstrates the use of the low-level API and how to run libsurvive. The *TFF.Process* library registers to this API for log, pose and IMU updates.

4.2.5 Tracking Data

It follows a description of the data provided by the low-level API used for this work. Pose and IMU updates provide the struct `SurviveObject` (so) and a `timecode`. The run time is obtained with `survive_run_time(so->ctx)` and represents the time duration that libsurvive is running in seconds. Tracked devices are identified with `so->codename`, libsurvive uses the code name *T20* for the HMD, and *KN0* and *KN1* for the KCs. The unique serial number (`so->serial_number`) of a tracked device can be used to distinguish devices in case of an ambiguous code name. The pose of a device is available in local (`so->IMUPose`) and global (`so->OutPose`) space. Applying the translation pose (`so->head2imu`) allows to convert poses from the local to the global coordinate system. Each pose consists of a position represented as vector of cartesian coordinates and a rotation represented as quaternion. Time differences between updates can be calculated with `timeCode / so->timebase_hz`, which provides a more accurate run time in seconds. IMU updates expose the vector `accelgyro`, which holds the acceleration and angular velocity. libsurvive provides variances as vectors for position (`so->tracker->obs_pos_var`), rotation (`so->`

```c
#include "survive.h"
// libsurvive hooks
static void logCallback(SurviveContext *ctx, SurviveLogLevel logLvl, const
    char *msg);
static void imuCallback(SurviveObject *so, int mode, FLT *accelgyro,
    survive_timecode timeCode, int id);
static void poseCallback(SurviveObject *so, survive_long_timecode_type
    timeCode, const SurvivePose *pose);
...
// Low level API access
int main(int argc, char *argv[]) {
     SurviveContext *ctx = survive_init(argc, argv);
     // Register hooks
     survive_install_log_fn(ctx, logCallback);
     survive_install_imu_fn(ctx, imuCallback);
     survive_install_pose_fn(ctx, poseCallback);

     while (survive_poll(ctx) == 0) {
          survive_close(ctx);
     }
}
```

Listing 4.2: libsurvive low level API example

`tracker->obs_rot_var`), acceleration (`so->tracker->acc_var`) and angular velocity (`so->tracker->gyro_var`). When the Kalman filter and the use of IMU data for pose estimation are enabled, libsurvive also provides a velocity (`so->velocity.Pos`) and an angular velocity (`so->velocity.AxisAngleRot`).

4.2.6 Coordinate Systems

libsurvive uses a local and a global coordinate system upon calibration the origin of the global system is set to the location of the HMD. The IMU acceleration and angular velocity are referenced within the local system. In order to use them, one has to apply the integrated acceleration and angular velocity to the local pose and translate the result to the global system. The global position p_g is given by

$$p_g = r_l \otimes p_t + p_l, \tag{4.1}$$

where r_l denotes the local rotation, p_t the translation vector and p_l the local position. The global rotation r_g is given by

$$r_g = r_l \otimes r_t, \tag{4.2}$$

where r_l denotes the local rotation and r_t the translation quaternion.

4.3 Capture and Process Tracking Data

I developed the library *TFF.Process*, it captures tracking data provided by a source, processes them through filters and provides the result as output. Figure 4.4 illustrates the most important classes and interfaces that are involved. It follows an overview of the interfaces and classes, and an outline of the general procedure.

4.3.1 Interfaces

IAnalyzer: This interface specifies functionality to add an `IMetric` object with `addMetric(...)`, remove it with `removeMetric(...)` and subscribe for reports of attached metrics with `onReport(...)`.

IController: This interface specifies functionality to register callbacks for log and filter update events with `onLog(...)` and `onFilterUpdate(...)`. It also allows to start and stop the process of capturing and processing tracking data with `start(...)` and `stop()`. The current thread can be blocked until the process has stopped with `join()`.

ISource: This interface specifies controls for a tracking source. It allows to start gathering tracking data with `start(...)`, stopping the source with `stop()` and to register callbacks for log events of a tracking source with `onLog(...)`.

IProcessor: The functionality of this interface allows to process a tracking update through a filter with `tryProcessUpdate(...)` and to obtain the result upon successful completion with `getFilterData(...)`.

IProcessorRepository: Defines functionality to retrieve a processor for a specific device with `getProcessor(...)` and to set the underlying filter processor for all devices with `setFilterType(...)`. Available filter types are `Passthrough`, `Desp` and `Eskf`.

IMetric: The internal calculations of a metric are triggered with `onPulse(timeDelta : ...)` where `timeDelta` represents the time since the last pulse in milliseconds and a report is returned with `getReport()`.

4.3.2 Data Object Classes

SourceDevice: This class holds properties of a tracked device provided by a tracking source. These values are device id, serial number, accelerometer and gyroscope bias, acceleration and gyroscope scale, observation variance for position and rotation, and acceleration and gyroscope variance.

SourceData: This class holds the run time, update type, source device and other tracking properties provided by the tracking source. Supported tracking properties are

global pose (`PoseData`), local pose (`PoseData`), IMU measurements (`ImuData`), local to global transformation pose (`PoseData`).

FilterData: This data structure holds the output of a processor. The `FilterData` object holds the `SourceData` object, the output pose (`PoseData`) and several other properties depending on the processor.

PoseData: This class holds the position and rotation of a pose.

ImuData: Stores the acceleration and angular velocity of the IMU.

4.3.3 Classes Overview

Controller: This class implements the interface `IController`, it is constructed with an instance of `ISource`, `IProcessorRepository` and `IAnalyzer`. On the function `start(...)` the `Controller` runs two threads. The first thread runs the tracking source object `ISource` with the function call `start(...)` and passes the configuration arguments and a `ConcurrentQueue` object. The `ISource` object pushes `SourceData` objects into `ConcurrentQueue` with the function `push(...)` for each new tracking update. The second thread waits for and dequeues these updates with the function `waitAndPop(...)`, gets the `IProcessor` object of the device the update is related to from the `IProcessorRepository` with the function `getProcessor(...)` and tries to process the update with `tryProcessUpdate(...)`. If the processing succeeds the thread calls the function `getFilterData(...)`, gets the object `FilterData` and passes it to all callback functions that have been registered with `onFilterUpdate(...)` by signaling the filter update event. Additionally the `ControllerMetric` is updated for each filter update.

LibSurviveSource: This class implements the interface `ISource`, it registers to events of the libsurvive low-level API (see 4.2.4) and runs libsurvive on the function `start(...)`. It creates a `SourceData` objects for each libsurvive update and pushes it into the `ConcurrentQueue` object.

Processer: An abstract base class for a processor implementation, which implements the interface `IProcessor`. It provides basic properties and functionality used by all implementations of a processor. Each `Processor` is related to a tracked device object `SourceDevice`, applies the filter logic and stores the filter state.

PassthroughProcessor: Derives from the base class `Processor` and implements a passthrough logic, which transfers the tracking data of the `SourceData` object to a `FilterData` object on `tryProcessUpdate(...)` without any modifications of the underlying data.

DespProcessor: A `Processor` implementation that applies the DESP to the tracking data on `tryProcessUpdate(...)`.

ErrorStateKalmanProcessor: A `Processor` implementation that applies the ESKF to the tracking data on `tryProcessUpdate(...)`.

ProcessorRepository: This class stores the `Processor` objects for the specified filter type of each tracked device object `SourceDevice`. On the function call `getProcessor(...)` the corresponding `Processor` object for a device is returned, if a `Processor` for a device doesn't already exist it will be created and initialized. On the function `setFilterType(...)` a new filter type is set and all existing `Processor` objects are cleared.

Analyzer: This class implements the interface `IAnalyzer`, it creates reports for all attached `Metric` objects on a periodic pulse and invokes all the callback functions that have been registered with `onReport(...)`.

ControllerMetric: This class calculates update metrics for tracked devices, it keeps track of the number pose and IMU updates, and creates a report on `onPulse(...)` that can be retrieved with `getReport(...)`.

ConcurrentQueue: A thread safe queue implementation with the ability to block a thread, wait for new items, and to cancel the block on thread termination.

4.3.4 Procedure

1. *Run*: A `Controller` object is constructed with a tracking source (`ISource`), an `IProcessorRepository` and an `IAnalyzer` object. A `ControllerMetric` object is created on initialization and attached to the `IAnalyzer` with the function `addMetric(...)`. The `Controller` is then started with the function `start(...)`.

2. *Receive*: On start the `Controller` runs the object `ISource` with `run(...)`. The `ISource` object pushes updates, for each tracked device object `SourceDevice`, in the form of a `SourceData` object into the passed `ConcurrentQueue` with `push(...)`.

3. *Process*: Next the `Controller` dequeues the `ConcurrentQueue` object with the function `waitAndPop(...)` and gets the `IProcessor` object, for the device the update is related to, from the `IProcessorRepository` with `getProcessor(...)` and processes this `SourceData` object through the `Processor` with the function `tryProcessUpdate(...)`. The function applies the filter implementation and `getFilterData(...)` provides a `FilterData` object on success.

4. *Analyze*: The `Controller` also updates the `ControllerMetric` on successful processing. The `Analyzer` produces reports on a specified interval and writes them to the standard output.

5. *Propagate*: The `Controller` propagates the filter result on successful processing by signaling the filter update event, which invokes all callback functions that have been registered with `onFilterUpdate(...)`.

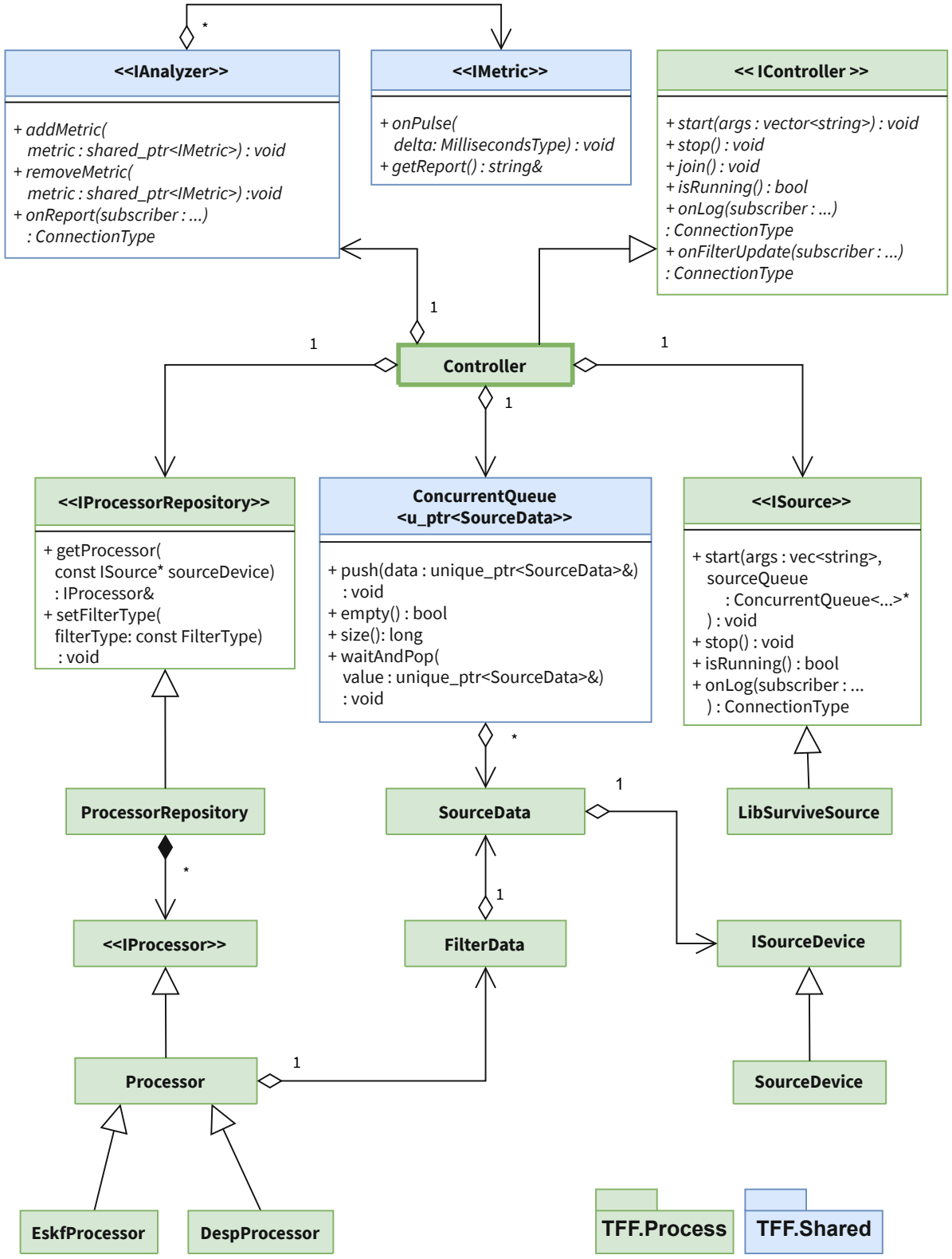


Figure 4.4: Classes involved in capturing and processing tracking data

4.4 Transmit Tracking Data

The application *TFF.Server*, receives, processes and propagates tracking data via websocket to all connected clients. The external client library *TFF.Client* is linked by the UE plugin *TFF.Plugin* and allows to receive tracking data from a websocket server. The library *TFF.Websocket* holds the shared websocket implementations to host a server, it connects to a server with a client and transmits data between the two. Figure 4.5 gives an overview of the relevant classes and their packages, and figure 4.6 gives a more detailed overview of the relevant websocket classes. It follows an overview of the involved classes, and an outline of the general procedure.

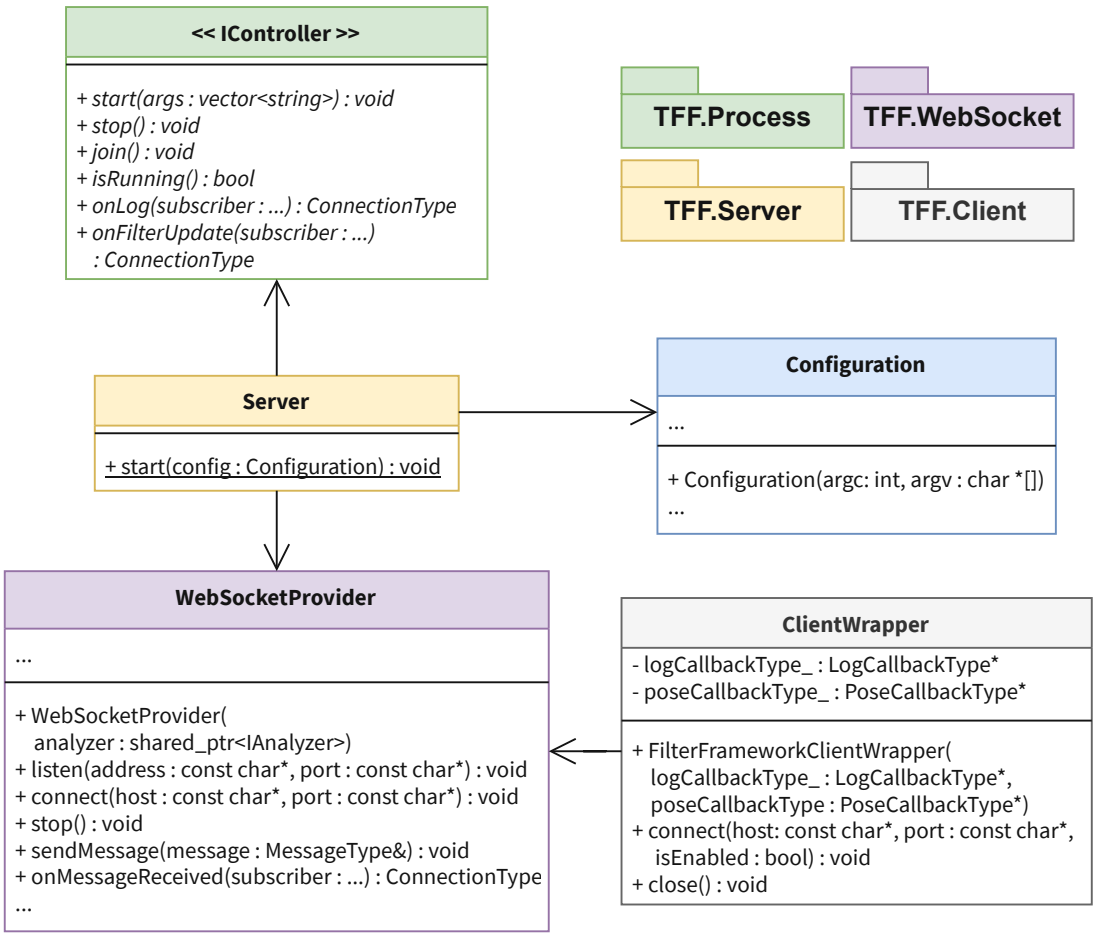


Figure 4.5: Websocket server and client classes

4.4.1 Configuration, Server and Client Wrapper Classes

Configuration: The configuration class parses the command line arguments and exposes the corresponding settings to other components. It uses boosts program arguments feature.

Server: The function `start(...)` of this class reads the `Configuration` object, creates and initializes the `Controller` object and its dependencies, subscribes for filter update events of the `Controller` with the function `onFilterUpdate(...)` and runs it with `start(..)`. It creates a `WebsocketProvider` object to open a websocket server and accepts incoming connections with the function `listen(...)`. Once a client has connected, a `ServerSession` object is created, which handles the communication between the server and the client. Filter updates in the form of `FilterData` objects from the `Controller` are serialized as JSON string and propagated to websocket clients with the function `sendMessage(...)`.

ClientWrapper: This class is as wrapper for the `WebsocketProvider` class and abstracts external dependencies. It is designed to be used with the UE and avoids problems with namespaces and macros. Is is initialized with callback functions for log and pose update events. It connects to the websocket server with the function `connect(...)`, requests pose updates from the server, deserializes incoming messages and invokes the callback function `poseCallback_(...)` on incoming pose updates.

4.4.2 Websocket Classes

WebSocketProvider: This class allows to open a websocket server with `listen(...)` or connect to one with `connect(...)`. It provides callbacks for the events connect, disconnect, message received, log entry and a function to send a message to connected clients or the server. On `connect(...)` the provider connects to a server and creates a `ClientSession` object which handles the communication with the server. On `listen(...)` the provider creates a `Listener` object which listens for incoming websocket connections. The `Listener` creates a `ServerSession` object for each incoming connection which handles the communication between the server and the client. The function `onMessageReceived(...)` registers callbacks for message received events and the function `sendMessage(...)` sends a message through an open websocket connection to the server or to clients. The websocket and all open connections are closed on the function `stop()`.

SharedState: This class is used to share information and notify other components of occurred events. It provides notifications for connect, disconnect, message received, message send and log entry events, and allows to invoke message send events with the function `signalMessageSend(...)` and message received events with function `signalMessageReceived(...)`.

Listener: This class implements logic to open and bind a websocket endpoint, and to listen for incoming connections on `run()`. It creates a new `ServerSession` object for each incoming connection.

Session: An abstract class that implements basic functionality to asynchronously read from and write to an open websocket connection with `readAsync()` and `writeAsync(...)`. It listens to message send events of the `SharedState` object with `OnMessageSend(...)`, and queues and sends these messages asynchronously. It signals session disconnects and invokes received message events with `signalMessageReceived(...)` of the `SharedState` object.

ServerSession: This class derives from the base class `Session`, it accepts and runs a websocket connections with a client on `run(...)`, and notifies the `SharedState` object about an established connection.

ClientSession: This class derives from the base class `Session`, it resolves and connects to a websocket endpoint, runs a websocket connection with the server on `run(...)`, and notifies the `SharedState` object about an established connection.

TransmissionMetric: This class calculates transmission metrics, it counts the number of received updates for each device, calculates the latency between server and client for each message, averages it over the specified time interval, determines the minimum and maximum latency and creates a report on `onPulse(...)` that can be retrieved with `getReport(...)`.

4.4.3 Procedure

1. *Parse configuration:* The application creates a `Configuration` object and passes the CLI parameters, the object parses them and provides access to the specified settings. If a settings is not specified a default value is returned.
2. *Run server*: A `server` objects is created and its function `start(...)` is invoked with the `Configuration` object. The function wraps the logic to run a tracking source, receive filter updates, open a websocket server, accept incoming connections and distribute pose updates to clients.
   a) *Run controller:* The `Controller` object and its dependencies are created, initialized and set up with settings provided by the `Configuration` object. A filter update callback function is registered with `onFilterUpdate(...)`. The `Controller` is then started with `start(...)` to receive filter updates (see chapter 4.3).
   b) *Run websocket server:* A `WebsocketProvider` object is created and the function `listen(...)` is used to host and run a websocket server. It listens for incoming connections, creates a session for each client, listens for incoming messages and propagates pose updates to all connected clients that have requested updates.

3. *Connect client:* The wrapper class `ClientWrapper` connects to the websocket server with `connect(...)`, requests filter updates and provides callbacks for log entries and incoming pose updates. It is designed to be used as a client with the UE and acts as a wrapper for the `WebsocketProvider` object.

4. *Transmit:*

   a) *Server:* The filter update callback function registered prior on the `Controller` with `onFilterUpdate(...)` serializes the passed `FilterUpdate` object on invocation as JSON string and passes it to the `WebSocketProvider` object with `sendMessage(...)`. Each `ServerSession` object sends this message then to the connected client, if it has requested pose updates.

   b) *Client:* The `ClientWrapper` objects deserializes the JSON string message on the arrival of a pose update and invokes the pose update callback function `poseCallback_(...)`.

5. *Disconnect client:* A session between a client and server closes when the client disconnects or the server shuts down.

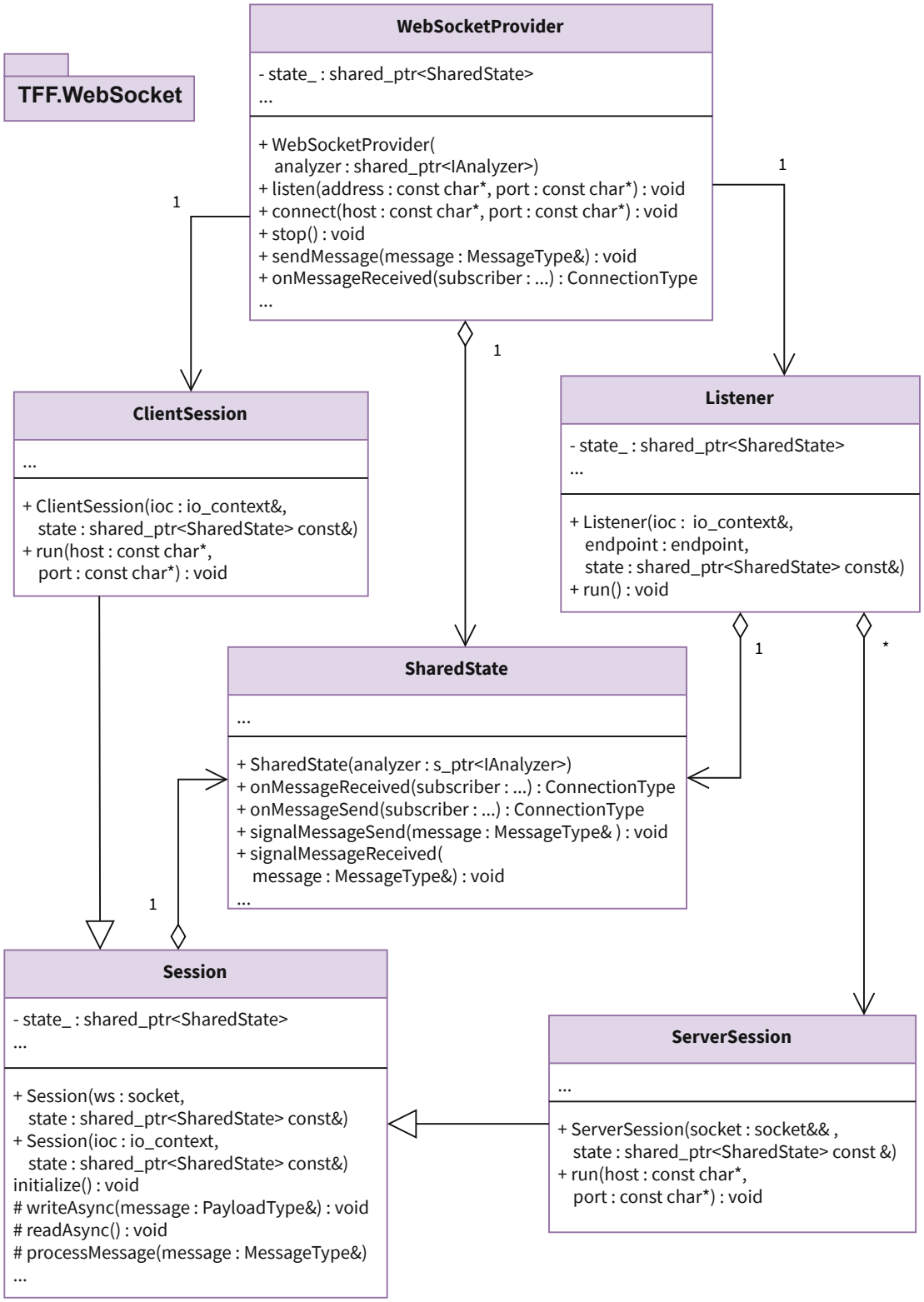


Figure 4.6: Websocket classes

4.5 Apply and Visualize Tracking Data

The application *TFF.VRA*, hosts the VE used to conduct the user study and logs reports for analysis. It relies on the *TFF.Plugin* to connect to the *TFF.Server* and receive pose updates, and uses the modified *SteamVR* plugin to apply those poses to objects within the VE. The figure 4.7 illustrates the most important classes and interfaces that are involved. The application is built with the UE version *5.0.2* and is based on the *Virtual Reality Game* template.

The UE plugin *TFF.Plugin* copies the required dependencies to the UE build output and loads the required DLLs on runtime. Due to compatibility issues of the UE with the C Standard General Utilities Library and the requirement of an independent filter framework, a websocket approach was implemented to transfer tracking poses to the UE. The UE websocket component had latency issues causing messages to be delayed by up to 40 ms, therefore a custom websocket approach was implemented using *Boost* websocket.

The plugin *SteamVR* needs to be modified to inject poses provided by an alternative source other them SVR. The integration requires to copy the *SteamVR* UE plugin folder

`C:\Program Files\Epic Games\UE_5.0\Engine\Plugins\Runtime\Steam\SteamVR`

to the projects plugin folder

`..\Unreal_Project\Plugins`

and to enable it under

Unreal Editor → **Plugins** → **Project** → **Virtual Reality** → **SteamVR**

with the UE Editor. The UE classes `SteamVRHMD` and `SteamVRInputDevice` are modified to receive pose updates from the *TFF.Plugin*.

4.5.1 TFF Plugin Classes and Interfaces

IModuleInterface: This UE interface specifies the functionality to load and unload a module with `StartModule()` and `ShutdownModule()`. An UE module can be accesses from another UE components with `FModuleManager::GetModulePtr<T>(...)`.

ITffSystem: This interface allows other components to get the pose of a tracked device with the function `TryGetPose(...)`. It also specifies controls to set and get the current filter system with `SetFilterSystem(...)` and `GetFilterSystem()`. Available filter systems are the TFF *Passthrough*, *DESP* and *ESKF*, as well as the *SteamVR* system which does not use tracking data provided by the TFF.

FTffSystem: This class implements the interface `ITffSystem`. It sets up a `ClientWrapper` object and its callback functions and connects to the websocket server with `connect(...)`. It received pose data from the server, stores the pose for each tracked device internally and provides it to other components with `TryGetPose(...)`

FTffPluginModule : This class implements the interface `IModuleInterface` and loads external dependencies such as the *TFF.Client* at runtime. It creates an instance of the class `FTffSystem`, establishes and runs a connection with the server on `StartupModule(...)`, executes a cleanup on `ShutdownModule(...)` and exposes the `FTffSystem` object to other components through the function `GetTffSystem()`.

Pose: A Data class that holds the pose and the code name of a tracked device.

4.5.2 *SteamVR* Plugin Classes

SteamVRHMD : This class is part of the UE *StreamVR* plugin and handles the HMD, it has been modified in order to inject poses from the TFF. On startup this class loads the module `FTffPluginModule`, gets the object `ITffSystem` with the function `GetTffSystem()` and stores it locally. The modified function `UpdatePoses(...)` uses the stored object `ITffSystem` to check the current filter system with `GetFilterSystem(...)`. If the filter system is set to anything other than *StreamVR*, the modified function updates the pose of the tracked HMD with the result of the `ITffSystem` objects function `GetPose(...)`.

SteamVRInputDevice : This class is part of the UE *StreamVR* plugin and handles the KCs, it has been modified in order to inject poses from the TFF. It loads the module `FTffPluginModule` in the constructor and stores the object `ITffSystem` using `GetTffSystem()`. The modified function `GetControllerOrientationAndPosition(...)`, maps the internal identifiers to the correct code name and uses the stored object `ITffSystem` to check the current filter system with `GetFilterSystem(...)`. If the filter system is set to anything other than *StreamVR*, the modified function updates the pose of the tracked KC with the result of the `ITffSystem` objects function `GetPose(...)`.

4.5.3 Procedure

1. *Initialization:* On startup the application *TFF.VRA* creates instances of the modules *TFF.Plugin* and *StreamVR*. The module `FTffPluginModule` loads the external *TFF.Client* dependency with `LoadDependencies()` and creates a `FTffSystem` object. The *SteamVR* objects `SteamVRHMD` and `SteamVRInputDevice` load the `FTffPluginModule` at startup, get the `IFTffSystem` object with `GetFilterSystem()` and store it internally.

2. *Connect to websocket server:* The module `FTffPluginModule` connects to the websocket server with the `FTffSystem` objects function `connect()` on `StartupModule()`. This creates a `ClientWrapper` object, registers the pose and log callback on creation, connects with `connect(...)`, and runs the connection within a separate thread. If the connection to the server drops or server is not available the thread tries to reconnect automatically.

3. *Receive pose updates:* Once a connection to the server is established the object `ClientWrapper` receives and deserializes the messages, and invokes a pose callback for each one. The `FTffSystem` object stores the received pose for each device internally on a callback function call.

4. *Apply and visualize pose updates:* The poses of SVR devices are updated at regular intervals. The object `SteamVRHMD` updates the pose of the HMD with `UpdatePoses(...)` and each `SteamVRInputDevice` object updates the pose of a KC with `GetControllerOrientationAndPosition(...)`. If the current filter system is set to anything other than `StreamVR`, the `FTffSystem` objects function `GetPose(...)` is used to get and apply a pose received from the server. Applying this new pose results in a visualization update within the VE on the next render update.

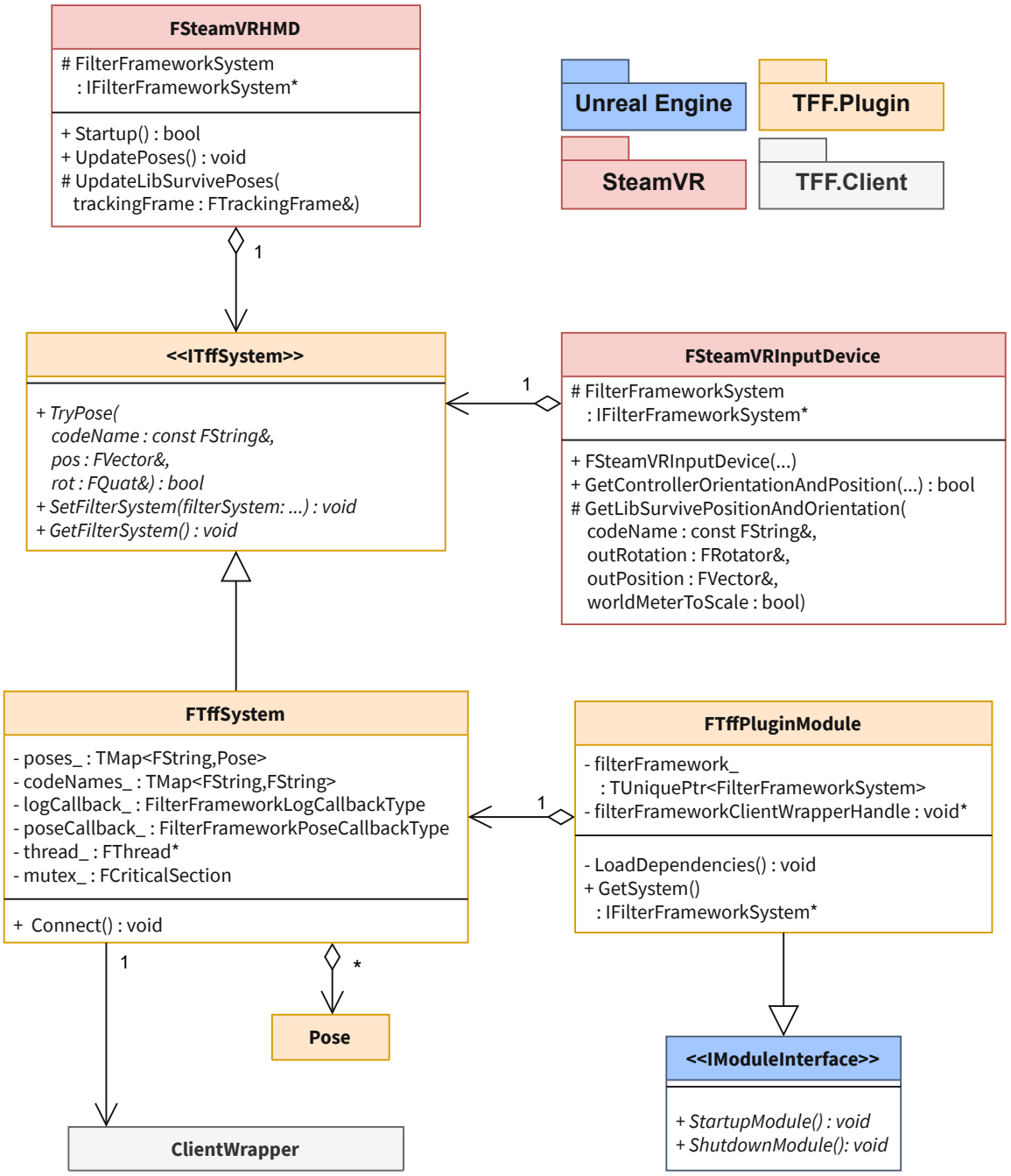


Figure 4.7: UE integration class diagram

4.6 Evaluation

The evaluation is designed to optimize and validate the performance of the TFF and its implemented filter algorithms. The TFF relies on libsurvive as mentioned in chapter 4.2 and enables to apply the DESP (see chapter 3.3) and the ESKF algorithm (see chapter 3.4) to libsurvive tracking data. Important tracking properties (see chapter 4.6.1), the existence of tracking errors (see chapter 4.6.2) and the impact of filter algorithms are verified. A VE is created with the UE, which makes it possible to test filter implementations from the users perspective and compare them to SVR.

The evaluation consists of three parts, a direct assessment of performance measurements, an analysis of the filter outputs, and the conduction of a user study within a VE:

TFF performance: The TFF measures and records the tracking properties sampling rate and latency for each device as quality measure to be able to verify that they do not negatively impact the tracking performance.

Filter performance: The TFF allows to save the output of filter algorithms to a text file. The stored data is then imported into Matlab and plotted to analyze it. This allows to run TFF with different filter parameters and select suitable ones by analyzing their impact on the filter output. The created plots also allow to review the occurrence of tracking errors.

User study: A user study is conducted to check the tracking systems accuracy, the subjective occurrence of tracking errors, and the ability of a filter algorithm to enhance tracking performance and therefore improve the user experience. A VE is created with the UE, which makes it possible to evaluate the performance of the filter algorithms from the user's perspective inside the virtual world. The user has to complete several tasks for each filter system, which covers different aspects of interactions within the virtual world, to evaluate the filter performance. All tasks have to be completed in one session for each filter system, the order for each filter system is randomized. The user's feedback is obtained after each session.

4.6.1 Performance Measurements

The TFF provides two different performance metrics, the controller and the transmission metric. Both metrics have been implemented to simply assess the capability of the TFF to provide and deliver the number of required updates in time to the UE.

The controller metric is part of the TFF server implementation and measures the sampling rate. This metric gives feedback about the provided pose and IMU updates of libsurvive to distinguish between server and client problems. It measures the number of pose and IMU updates for each tracked device within a specified interval and writes them to the standard output.

The transmission metric is part of the TFF client implementation and is used by the UE to measure the latency between the TFF server and client. This latency including the processing time of an update by a filter. It was implemented to ensure the TFF capability to process and transmit the number of required updates to the UE within a few milliseconds. Upon the arrival of a libsurvive update, the system adds the current time as timestamp to the update information, this timestamp is appended to the pose update message that is sent to the UE. The TFF client then counts the number of received updates for each device, calculates the time difference, averages it over the specified time interval, determines the minimum and maximum latency and prints a report to the UE log.

The transmission metrics is run over a time span of 10 minutes with all available tracking devices, namely the HMD and both KC, within the virtual world to assess the systems capability to transmit all the updates to the UE within a few milliseconds.

4.6.2 Tracking Errors

The evaluation checks the existence of tracking errors, which ideally should be mitigated by a filter algorithm. This chapter discusses if and how the types of errors introduced in chapter 2.3.2 can manifest in libsurvive.

Static errors are spatial distortion, spatial jitter and drift and can occur if the tracked object is still.

Spatial distortion is a repeatable pose error where the real pose is not properly represented within the VE. Partial occlusion could cause spatial distortion in certain areas where the line of sight to the LHTS is broken. The LHTS is very accurate and spatial distortion is generally not an issue, but could occur if the user bows down to pick up an object and breaks the line of sight of the tracked object to the LHTS with his body.

Spatial jitter is caused by tracking noise and occurs when the tracked object within the VE appears to be shaking but is actually still. The output of libsurvive is inherently noisy, and it exhibits spatial jitter. However, by applying a filter, it is possible to significantly reduce this jitter.

Stability or drift is a slow but steady change in position or orientation for a tracked object within the VE but in reality the object is still. This error occurs if the optical system is unable to estimate a pose and only IMU data is used to predict a pose.

Dynamic errors can be categorized as latency and latency jitter while the tracked object is moving, or any other type of error not caused by static inaccuracy.

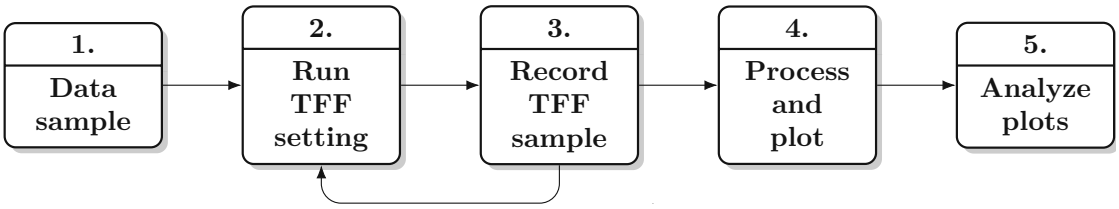


Figure 4.8: Filter performance analysis

Latency influences the time it takes to represent an actual movement within the VE. A high latency breaks the immersion with a noticeable delay between movement and representation and can cause nausea for the user.

Latency jitter is a variation in latency meaning the time from an actual movement to its representation within the VE varies. If the variation is significantly enough the user may experience twitching or spatial jitter along the path the object is moving.

Other types of error that can not be explained by static inaccuracy or latency, including sensor and algorithm prediction errors.

Latency and latency jitter due to transmission issues are generally not present in libsurvive unless the system is working at its capacity and updates cannot be processed in time or updates are delayed during transmission within TFF. The occurrence of other dynamic errors due to TFF and libsurvive pose estimation algorithms is a possibility. Pose distortion can be caused by the libsurvive poser algorithm and latency can occur due to excessive smoothing of filter algorithms.

4.6.3 Filter Performance Review

The evaluation of the filter performance is done by analyzing their impact on simulated and recorded tracking data. libsurvive and the TFF can be controlled through a CLI, these interfaces are used to create plots based samples with Matlab. The recorded samples used for this assessment are based on recordings of a conducted task performed within the VE and the simulated samples are generated movement data.

It follows a description of this procedure as illustrated in Figure 4.8:

1. Simulated tracking data is generated and written to a text file or movements of tracked devices are recorded to a text file as data samples with libsurvive by setting the parameter `--record file_path`.

2. The generated tracking data or the recorded file is then used by the TFF. This allows to run the same tracking scenario through different filters and parameter configurations.

3. Setting the TFF parameter **`-R file_path`** writes the filter output to a text file. The TFF can be executed multiple times with different settings to create different filter result samples that are based on the same tracking scenario.

4. Multiple TFF samples files can be loaded, processed and stored as binary file with Matlab. A Matlab script is used to load these cached samples files into memory to draw plots and compare the different results side by side. This allows to create custom plots for the different tracking properties provided by each filter. Figure 4.3 demonstrates a simple Matlab custom plot script to compare libsurvive raw to libsurvive Kalman filter positions.

5. The plots allow to analyze the filter output and compare the data side by side. They are used to review the manifestation of tracking errors and verify the impact of filters and their parameters.

```
force = false;
usesubplot = true;
playback_file_path = "sample_data.txt";
% Load sample data
raw = imp.get("raw", playback_file_path,...
     "--filter Passthrough --pt-use-imu 0",...
     "--use-imu 0 --use-kalman 0 --playback "+ playback_file_path +" --
         playback-factor 0", force);
libsurvive = imp.get("libsurvive", playback_file_path,...
     "--filter Passthrough --pt-use-imu 0",...
     "--use-imu 1 --use-kalman 1 --playback "+ playback_file_path +" --
         playback-factor 0", force);
% Setup plot configuration
pos_plot.file_name = file_name + "_pos";
pos_plot.axes = ["X","Y","Z"];
% Raw position
pos_plot.data.rawP = raw.output.T20.P;
pos_plot.legend.rawP = "[Raw]";
% libsurive position
pos_plot.data.lsP = libsurvive.output.T20.P;
pos_plot.legend.lsP = "[LsEskf]";
% Plot position
imp.plot_data("[Position]", usesubplot, pos_plot);
```

Listing 4.3: Matlab plot

4.6.4 Virtual Environment and Tasks

The virtual environment is created with the UE (see figure 4.9). It consists of a play area where the user can familiarize himself with the navigation and interaction within the virtual world. All the objects used for tasks are available for the user to play around with before a task has to be done. Additionally a short description before each task is displayed and the user can inspect the task before it starts. The task starts as soon as

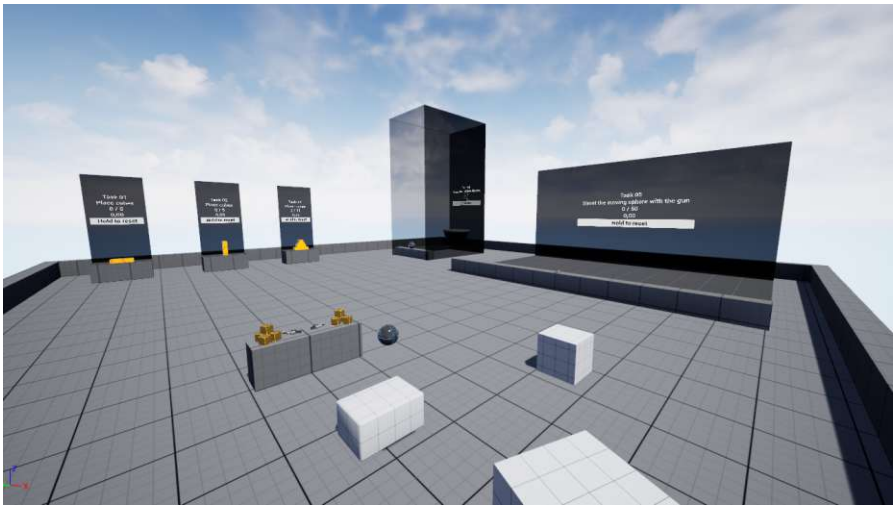


Figure 4.9: Virtual environment

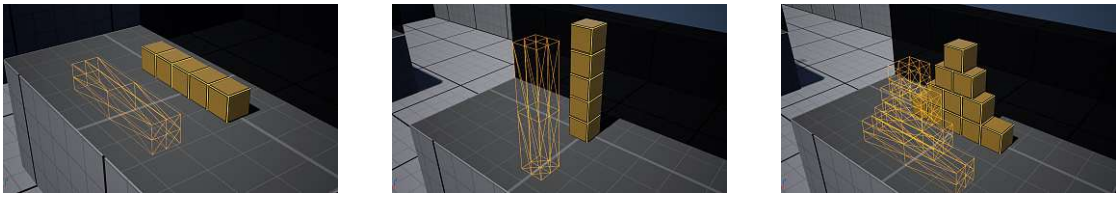
the user interacts with it. A display panel at the task shows the progress for each task once it has started.

The user has to complete five tasks for each filter system:

Place cubes: Three tasks evaluates close up interactions by picking up cubes and placing them within a highlighted volume on a table. The first task requires to stack cubes horizontally (see figure 4.10a), the second task to stack them vertically (see figure 4.10b) and the third task to stack them horizontally and vertically into the highlighted volume (see figure 4.10c). Cubes are highlighted by changing their color from yellow to green once they have been properly placed (see figure 4.11).

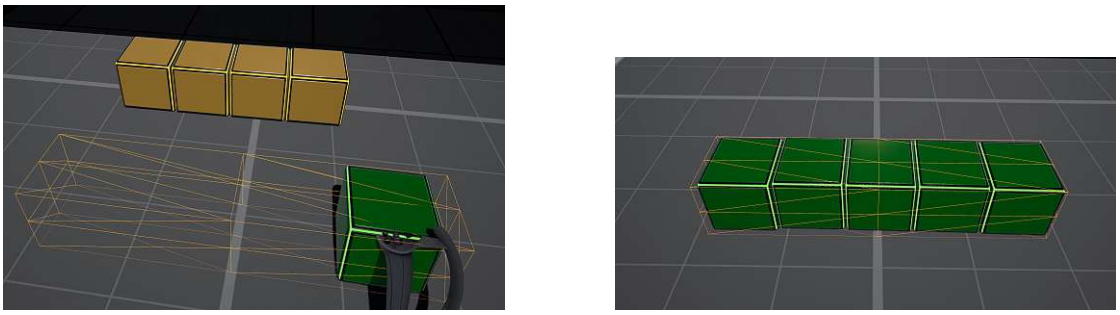
Throw a ball: The fourth task evaluates faster movements, it checks the ability to throw objects into a target by throwing a ball into a ring (see figure 4.12a). The user has to do this three times, when the ball is thrown into the ring or hits the ground it gets reset to its original position.

Shoot a target: The fifth task evaluates the ability to interact with a moving target by shooting projectiles 50 times at a randomly moving sphere with a gun (see figure 4.12b). The sphere appears when the task starts and moves from one randomly selected location to the next. It speeds up while moving to its next location and slows down before reaching it. After the task has been completed the sphere is removed.



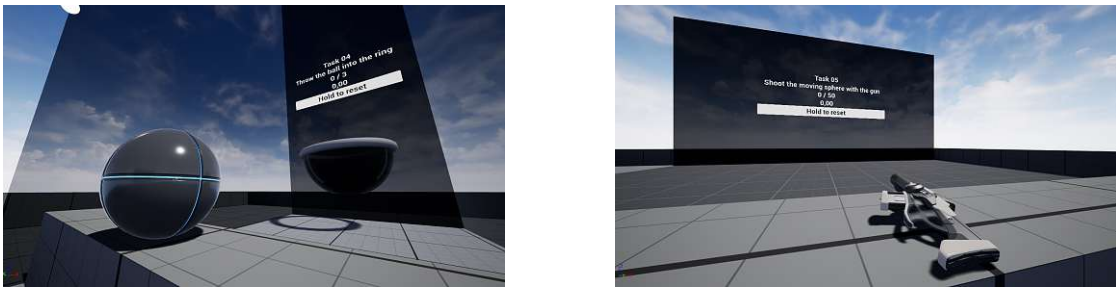
(a) Task 1 (b) Task 2 (c) Task 3

Figure 4.10: Place cubes



(a) Cube placed (b) Task completed

Figure 4.11: Interaction feedback



(a) Task 4 (b) Task 5

Figure 4.12: Throw a ball and shoot a moving target

4.6.5 User Study

The subjective evaluation of the TFF performance and the user acceptance is done in the form of a user study. The SVR tracking, as well as the filtered libsurvive tracking is being tested by letting the user perform tasks within a VE (see chapter 4.6.4). The user is surveyed after each filter system about the difficulty of each task and potentially occurred errors. Additionally the completion time and the number of interactions for each task is measured and recorded. The whole experiment takes the user approximately 20 minutes to complete. The user study is structured into five parts (see figure 4.13) where some of them have to be repeated for each filter system.

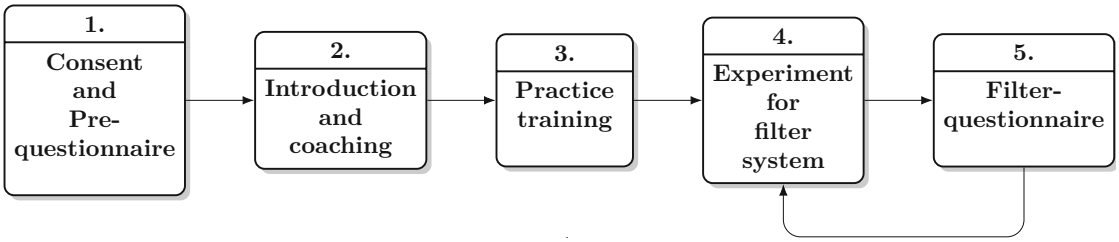


Figure 4.13: User study procedure

In the first part the user needs to read and sign a consent form and answer questions of a pre-questionnaire (see figure 4.14) with questions about the user's age, gender and prior experience with computer games and virtual reality. The question Q_1 is answered with the age of the user in years. The gender in Q_2 is categorized as male, female and other. The questions Q_3 and Q_4 have to be answered with the Likert scale:

1.	2.	3.	4.	5.
Not experienced at all	Slightly experienced	Moderately experienced	Very experienced	Extremely experienced

Q_1 What is your age?

Q_2 What is your gender ?

Q_3 How experienced are you with computer games ?

Q_4 How experienced are you with virtual reality ?

Figure 4.14: Pre-questionnaire

In the second part the user gets a brief introduction into potentially occurring tracking errors and how they might manifest (see chapter 4.6.2), to help to identify them during the test session. The user also gets a description of the tasks to perform within the VE, followed up by an introduction to the virtual reality system. At the third part the user is assisted to put the HMD and the KC on. This part allows the user to get familiar with the system, explore the VE, and learn how to navigate and interact with objects within the virtual world. During the fourth part each filter system is evaluated. The user has to perform a variety of tasks within the VE for each filter system. These tasks consist of stacking cubes, throwing a ball into a ring and hitting a moving target with the projectiles of a gun (see chapter 4.6.4). The time and number of interactions it takes to complete a task is recorded as quantitative data. The fifth part is a follow up for each evaluated filter system, at which the user gives subjective feedback in the form of a questionnaire. The questionnaire is grouped into four parts, the first group asks if

the user experienced dizziness or nausea (see figure 4.15), the second part consists of feedback about the perceived difficulty of each task (see figure 4.16), and the last two groups gather feedback about potentially experienced tracking errors of the HMD (see figure 4.17) and the KC (see figure 4.18). The question W_1 is answered with the Likert scale,

1.	2.	3.	4.	5.
Not dizzy at all	Slightly dizzy	Moderately dizzy	Very dizzy	Extremely dizzy

,

and the question W_2 with,

1.	2.	3.	4.	5.
Not nauseous at all	Slightly nauseous	Moderately nauseous	Very nauseous	Extremely nauseous

.

All T questions, about how challenging the tasks were perceived to be, are answered with the Likert scale:

1.	2.	3.	4.	5.
Not challenging at all	Slightly challenging	Moderately challenging	Very challenging	Extremely challenging

The questions H_6 and C_6 are answered with free text, and the rest of the H and C questions have to be answered with the Likert scale:

1.	2.	3.	4.	5.
Not at all	Rarely	Sometimes	Often	Almost always

After finishing the questionnaire for a filter system, the user continues to the fourth part and repeats every task for the next filter system. The order of the filter systems to be tested is randomized. Neither the instructor nor the user knows which system is currently active. The quantitative data is collected as soon as the user starts the task by interacting with it and is saved to a JSON file upon completion.

4.6.6 Data Acquisition

The UE stores a report in the form of a JSON file for each test session within the VE at the file location `Saved/Report-{GUID}.json`. The listing 4.4 shows the structure of the report, it contains all the collected quantitative data. The report contains a generated

W_1 Did you experience any dizziness ?

W_2 Did you experience any nausea ?

Figure 4.15: Well-being post-questionnaire

T_1 How challenging was the horizontal placement of the cubes in the first task?

T_2 How challenging was the vertical placement of the cubes in the second task?

T_3 How challenging was the horizontal and vertical placement of the cubes in the third task?

T_4 How challenging was it to throw the ball into the ring in the fourth task?

T_5 How challenging was it to hit the moving target with projectiles in the fifth task?

Figure 4.16: Task post-questionnaire

H_1 Have you experienced reoccurring distortions of your view for specific poses within the virtual world? (Spatial distortion)

H_2 Have you experienced any shaking of your view within the virtual world while holding still? (Spatial jitter)

H_3 Have you experienced any drifts in position or orientation of your view within the virtual world? (Stability or drift)

H_4 Have you experienced time delays between the movement of your head and the actual motion shown in the head mounted display? (Latency)

H_5 Have you experienced a variation in latency between movements of your head and the displayed motion in the head mounted display? (Latency jitter)

H_6 Have you experienced any other errors while using the head mounted displays that cannot be classified by any of the above error descriptions? (Dynamic error)

Figure 4.17: HMD post-questionnaire

C_1 Have you experienced reoccurring distortions for specific poses of any controller within the virtual world? (Spatial distortion)

C_2 Have you experienced any shaking of a controller within the virtual world while holding your hand still? (Spatial jitter)

C_3 Have you experienced drifts in position or orientation of a controller within the virtual world? (Stability or drift)

C_4 Have you experienced time delays between the movement of one of your hand and the actual motion of the controller within the virtual world? (Latency)

C_5 Have you experienced a variation in latency between movements of your hand and the related motion of the controller within the virtual world? (Latency jitter)

C_6 Have you experienced any other errors while using a controller that cannot be classified by any of the above error descriptions? (Dynamic error)

Figure 4.18: KC post-questionnaire

Globally Unique Identifier (GUID), the time of the reports creation and the status of each task for each filter system. Each task can be identified by the filter system and the task name, and contains the number of interactions, the time for completion and a flag that indicates if the task has been successfully completed.

Listing 4.4: JSON report

```
{
        "Time": "2022.12.08-11.53.18",
        "GUID": "34B08D134D88304C0E2DDDAC73A55455",
        "Tasks": [
                        {
                                "System": "Unreal",
                                "TestName": "Task␣01",
                                "Interactions": 33,
                                "Time": 120.23,
                                "Finished": true
                        },
                        ...
                ]
}
```

CHAPTER 5

Results and Discussion

This chapter documents and discusses the results of the TFF implementation. First we take a look at its ability to deliver the filter updates in time to the UE. Next we analyze the results of the TFF using generated as well as libsurvive tracking data. This chapter also discusses some issues related to the data that libsurvive provides. Finally we take a look at the subjective experience by analyzing the results of the conducted user study.

5.1 TFF Performance

As introduced in the section 4.6.1, we measure the TFF's capability to deliver the number of tracking updates provided by libsurvive in time to the UE.

The controller metric measures the number of pose and IMU updates provided by libsurvive for each tracked device within a specified interval. The result for a time interval of 1000 ms is,

Table 5.1: Controller metric report for a time interval of 1 second.

Device name	Number of pose updates	Number of IMU updates
HMD	100	984
Left KC	100	250
Right KC	100	252

and shows that libsurvive provides around 100 pose updates per second for each device, 1000 IMU updates per second for the HMD and 250 IMU updates for a KC.

The result of the transmission metric, which is run over a time span of 10 minutes for each filter with all available tracking devices, namely the HMD and both KCs, assesses the systems capability to transmit all the updates to the UE within a few milliseconds.

The transmission metric for the ESKF results in,

Table 5.2: ESKF Transmission metric report for a time interval of 10 minutes.

Device name	Number of updates	Average latency in ms	Minimum latency in ms	Maximum latency in ms
HMD	651390	0.200442	0	3.002
Left KC	163285	0.189505	0	3.001
Right KC	210359	0.176549	0	3.001

and for the DESP filter in:

Table 5.3: DESP Transmission metric report for a time interval of 10 minutes.

Device name	Number of updates	Average latency in ms	Minimum latency in ms	Maximum latency in ms
HMD	639950	0.165407	0	4
Left KC	170547	0.173136	0	3.001
Right KC	208643	0.162875	0	3.528

The transmission time for both filters includes the processing time. Looking at both tables above, the average latency is less than a millisecond, and the maximum peak latency less than or equal to 4 milliseconds. This shows that the TFF is capable of processing and transmitting the tracking data, provided by libsurvive for the hardware introduced in section 4.1.1, to the UE within the required time period of a few milliseconds.

5.2 Filter Performance with Generated Tracking Data

As introduced in section 4.6.3, this section documents the generation and application of simulated tracking data. We denote generated signals as true, noisy measurement signals as input and the result of a DESP or ESKF as filtered signal. The generated input tracking data represents position, orientation, acceleration and angular velocity data with added noise, it is generated with Matlab and written to a text file. In order to use this data with the TFF, a simulator data source has been implemented, which retrieves the data from the text file instead of libsurvive. It reads and provides every entry from the file as a pose or IMU update, which is then processed by a filter implementation. The result is then imported into Matlab and plotted. This allows to analyze the ability of filter algorithms to improve the noisy input data and compare it with the true signal. Generated tracking data also allows to create a controllable scenario, which makes it possible to modify and adapt the underlying data in order to quickly identify issues with filter algorithms. However, the use of simulated tracking data removes many real-world

problems and edge cases, limiting its application and reducing expressiveness when used alone.

5.2.1 Generated Tracking Data

We generate two sets of simulated tracking data, one where the object is moving and another one where the object is stationary. The moving object data is used to evaluate the performance of each filter and the stationary object data to verify the ESKF's correction of the acceleration and angular velocity, and the proper estimation of the accelerometer's and gyroscope's bias.

We define

$$f = 1000 \quad d = 10 \quad N = f \cdot d, \tag{5.1}$$

where f represents the frequency, d the duration of the data set in seconds and N the total number of data samples. Our series of timestamps t is then given by

$$t = \frac{1 \dots N}{f}. \tag{5.2}$$

For a moving object we define the true position kinematics as,

$$f_p(t) = sin(x^2) \tag{5.3}$$
$$f_v(t) = 2x\ cos(x^2) \tag{5.4}$$
$$f_a(t) = 2(cos(x^2) - 2x^2 sin(x^2)). \tag{5.5}$$

The function for the position is denoted as $f_p(t)$, for the velocity as $f_v(t)$ and for the acceleration as $f_a(t)$. Furthermore we define the 3-dimensional vectors for the axes x, y, z, of the position $p(t)$, velocity $v(t)$, gravity $g(t)$ and acceleration $a(t)$ as,

$$p(t) = [f_p(t) \quad f_p(t+1) \quad f_p(t+2)] \tag{5.6}$$
$$v(t) = [f_v(t) \quad f_v(t+1) \quad f_v(t+2)] \tag{5.7}$$
$$g(t) = [0 \quad 0 \quad -1] \tag{5.8}$$
$$a(t) = [f_a(t) \quad f_a(t+1) \quad f_a(t+2)] + g(t). \tag{5.9}$$

We define the true rotation kinematics of a moving object as,

$$f_r(t) = cos(x^2) \tag{5.10}$$

where $f_r(t)$ represents the rotation function and the rotation vectors $r(t)$ around the axes x, y, z are then given by,

$$r(t) = [f_r(t) \quad f_r(t+1) \quad f_r(t+2)]. \tag{5.11}$$

The corresponding quaternion representation $q(t)$ is given by,

$$q(t) = q\{r(t)\}. \tag{5.12}$$

The rotation's angular velocity vectors $\omega(t)$ are calculated with [17]:

$$\omega(t) = \frac{2}{\Delta t}\begin{bmatrix} q_w(t)q_w(t+\Delta t) - q_x(t)q_w(t+\Delta t) - q_y(t)q_z(t+\Delta t) + q_z(t)q_y(t+\Delta t) \\ q_w(t)q_y(t+\Delta t) + q_x(t)q_z(t+\Delta t) - q_y(t)q_w(t+\Delta t) - q_z(t)q_x(t+\Delta t) \\ q_w(t)q_z(t+\Delta t) - q_x(t)q_y(t+\Delta t) + q_y(t)q_z(t+\Delta t) - q_z(t)q_w(t+\Delta t) \end{bmatrix} \tag{5.13}$$

In order to test the filters with noisy data, we define the following 3-dimensional noise vectors with a normal distributed noise,

$$p_n \sim \mathcal{N}(0, 0.005) \quad a_n \sim \mathcal{N}(0, 0.001) \quad a_w \sim \mathcal{N}([0.05\ 0.01\ -0.06], 0.001) \tag{5.14}$$

$$r_n \sim \mathcal{N}(0, 0.005) \quad \omega_n \sim \mathcal{N}(0, 0.001) \quad \omega_w \sim \mathcal{N}([-0.06\ 0.01\ 0.05], 0.001), \tag{5.15}$$

and add them to the generated true kinematics vectors in order to get our noisy input signal,

$$p_{in}(t) = p(t) + p_n(t) \tag{5.16}$$

$$a_{in}(t) = q\{r(t)\}^{-1} \otimes a(t) + a_n(t) + a_w(t) \tag{5.17}$$

$$r_{in}(t) = r(t) + r_n(t) \tag{5.18}$$

$$q_{in}(t) = q\{r_{in}(t)\} \tag{5.19}$$

$$\omega_{in}(t) = w(t) + \omega_n(t) + \omega_w(t). \tag{5.20}$$

Note that the acceleration $a(t)$ must be rotated by the inverse rotation $q\{r\}(t)^{-1}$ in order to simulate the values that the IMU would provide and that the generated tracking data is designed to provide nine IMU updates in between every pose update.

For a stationary tracked object we define the position kinematics,

$$f_p(t) = 1, \quad f_v(t) = 0, \quad f_a(t) = 0, \tag{5.21}$$

and the rotation kinematics,

$$f_r(t) = 1, \quad \omega(t) = 0, \tag{5.22}$$

and create the noisy input signal the same way we did for a moving object.

5.2.2 DESP

The DESP is applied to the input signal, and the filtered results are plotted with Matlab. In order to find the optimal filter parameter α, different plots are created, and the results are compared side by side. The figure 5.1 shows a sample of DESP filtered position

results, and the figure 5.2 shows filtered rotation results. Both figures illustrate DESP filtered results using a value of 0.1 for the filter parameter α. This value results in strong smoothing, while smaller values introduce a noticeable lag, causing the filtered signal to lag behind, and greater values result in a more noisy signal.

Looking at the result of the DESP filter a few things become apparent. The filtered signal, while still being noisy, is an improvement compared to its noisy input. Looking at the prediction results in between simulated optical updates, which can be identified as group in between jumps where consecutive points are close to each other, it can be seen that the slope of the signal is clearly lagging behind. The DESP is capable to filter the input signal to some degree but adds a lag to the filtered signal. The algorithm only provides one parameter to filter the level and trend, it is therefore not possible to filter them independently.

We can summarize the results for simulated tracking data as follows:

- The result is an improvement compared to its noisy input.
- The slope of the predicted estimate is lagging behind. This causes a lag for faster movements.
- Only one parameter is available to filter both level and trend.

5.2.3 ESKF

Similar to the DESP, we apply the ESKF to the input signal and plot the results with Matlab. The variance parameters used for the ESKF are equal to the variances used to generate the data, as introduced in chapter 5.2.1. Before looking into the position and rotation results, we verify the correction of the acceleration and angular velocity, and the proper estimation of the accelerometer's and gyroscope's bias. In order to do that we use the generated tracking data of a stationary object, apply the ESKF and plot the results. The figure 5.3 shows that the generated acceleration is correctly rotated and the gravity removed. Furthermore the arbitrary defined acceleration bias of $[-0.05\ 0.01\ 0.05]$ has been approximately estimated by the ESKF, which leaves us with a noisy but corrected ESKF acceleration of about zero. The figure 5.4 shows that the ESKF approximately estimates the arbitrary defined gyroscope bias of $[-0.06\ 0.01\ 0.05]$ and corrects the provided angular velocity, resulting in a noisy but corrected angular velocity close to zero.

Next we take a look at the filtered signal of a moving object. The figure 5.1 shows a sample of ESKF filtered position results, and the figure 5.2 filtered rotation results. It can be seen that the filtered results are close to the true signal, this means that the ESKF is capable of filtering noisy tracking data. Analyzing the prediction results of simulated IMU updates in between simulated optical updates shows that the trend is correctly estimated.

We can summarize the results for simulated tracking data as follows:

- The result is a significant improvement compared to its noisy input.
- The filter is able to estimate the accelerometer and gyroscope bias correctly.
- The filter is able to process fast and slow movement data correctly.

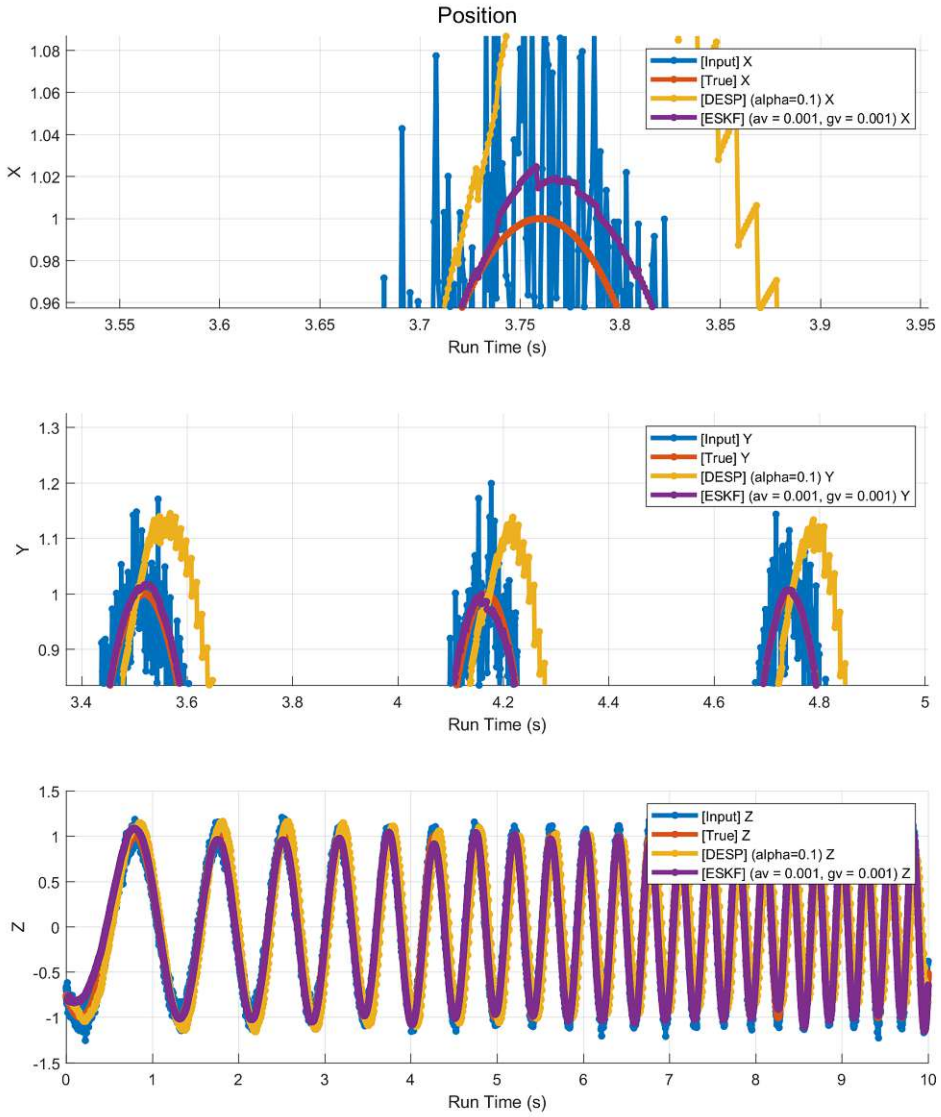


Figure 5.1: Filtered position of a generated tracking data sample.
This figure illustrates the filtered position of a generated tracking data sample. The different zoom levels highlight different aspects of the tracking data. The plots of X and Y show that the DESP filtered result is still very noisy, introduces a significant lag on the signal, and that the slope of the prediction is clearly lagging behind. It also shows that the ESKF filters the noisy input correctly and predicts the slope of the signal accurately. The last plot of Z gives an overview of the entire sample, it shows that the DESP lags behind while the ESKF can filter the signal properly even for faster movements.

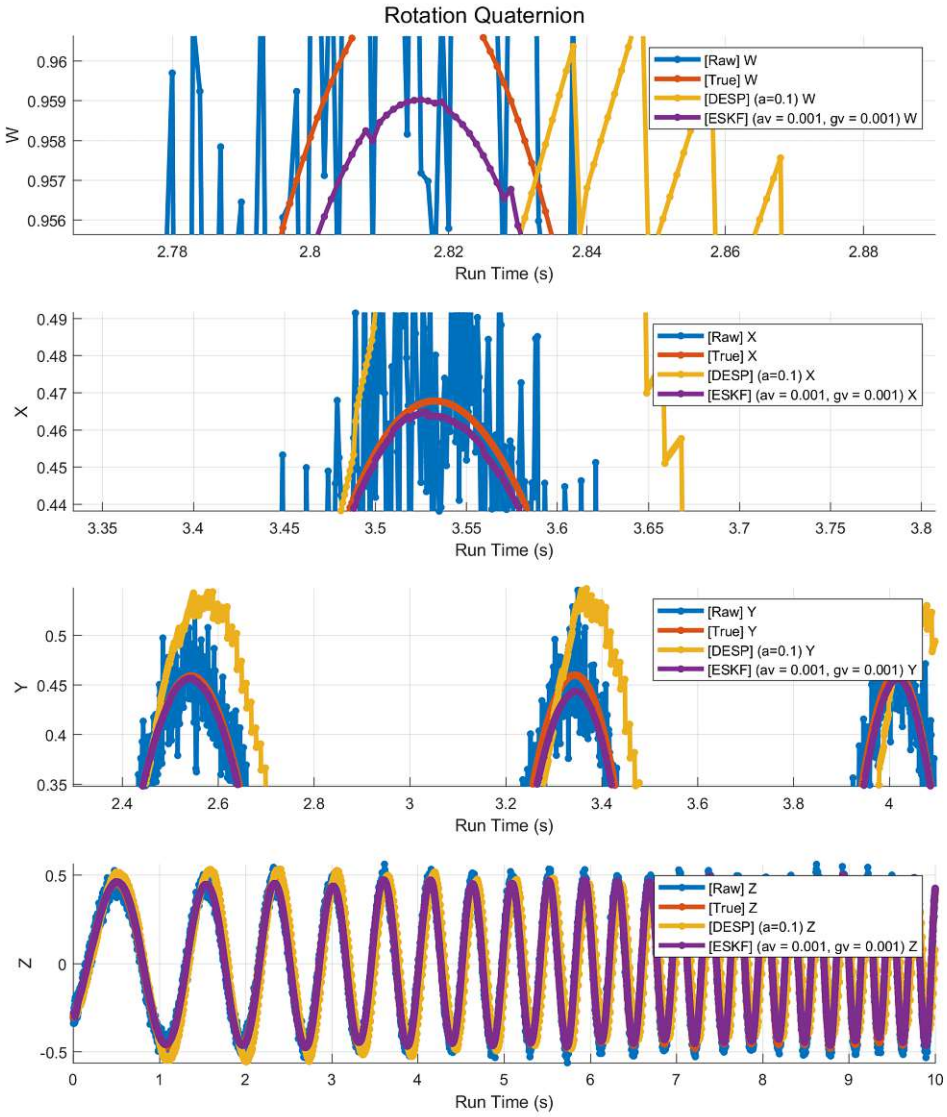


Figure 5.2: Filtered rotation, represented as quaternion, of a generated tracking data sample.

This figure illustrates the filtered rotation, represented as quaternion, of a generated tracking data sample. Again, different zoom levels highlight different aspects of the tracking data. The results are similar to those of the filtered position. The plots of W and X show that the DESP result is very noisy, lags behind and that the slope of the prediction is lagging behind too. The ESKF on the other hand filters the signal properly and predicts the slope accurately. The plot of Z shows the same result of the entire signal, and also shows the results for faster movements at the end of the sample.

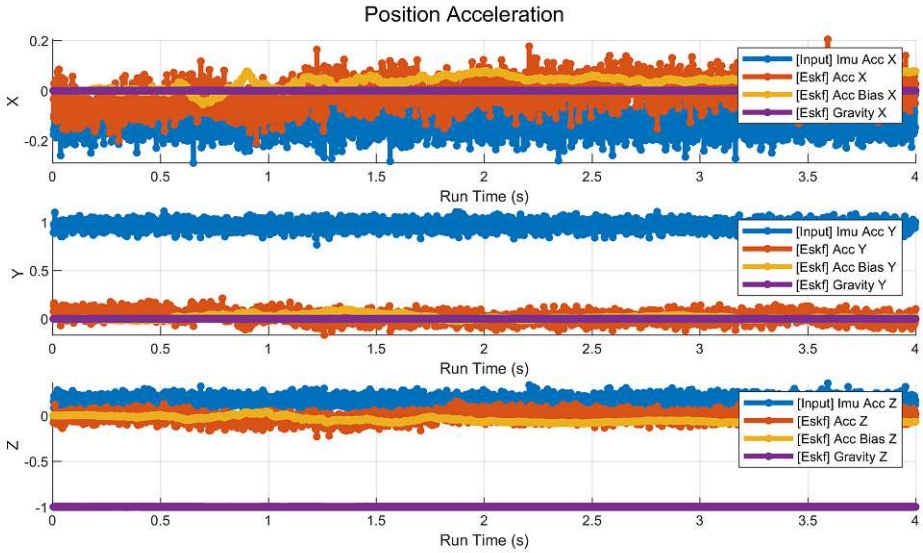


Figure 5.3: Generated acceleration of a still object sample.

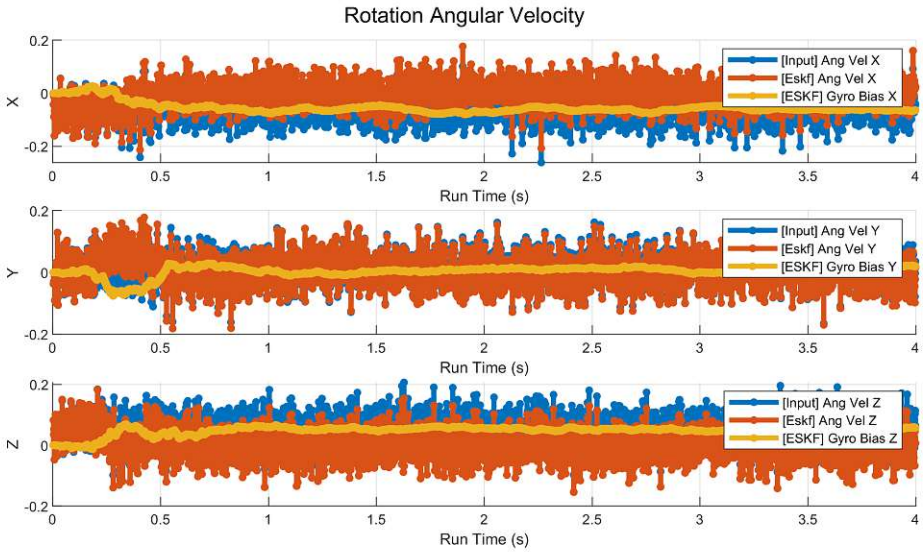


Figure 5.4: Generated angular velocity of still object sample.

5.3 Filter Performance with libsurvive Data

5.3.1 libsurvive Data

Before discussing the results of the specific filter implementations, we take a look at the data provided by libsurvive and discuss the related issues. libsurvive provides a *Run Time* which marks the time of the provided pose or IMU update, and more precise *Internal Time* representing the time of the IMU measurement or pose estimation. The two *Internal Time* clocks which provide the time stamps for IMU measurements and pose estimations are not aligned by default. libsurvive aligns these time measurements only if the internal Kalman filter is enabled, which does not apply in our case. This causes issues as the *Internal Time* for poses lie back in time, compared to the *Internal Time* for IMU measurements (see figure 5.5), because of this we only use the *Internal Time* stamps for IMU measurements.

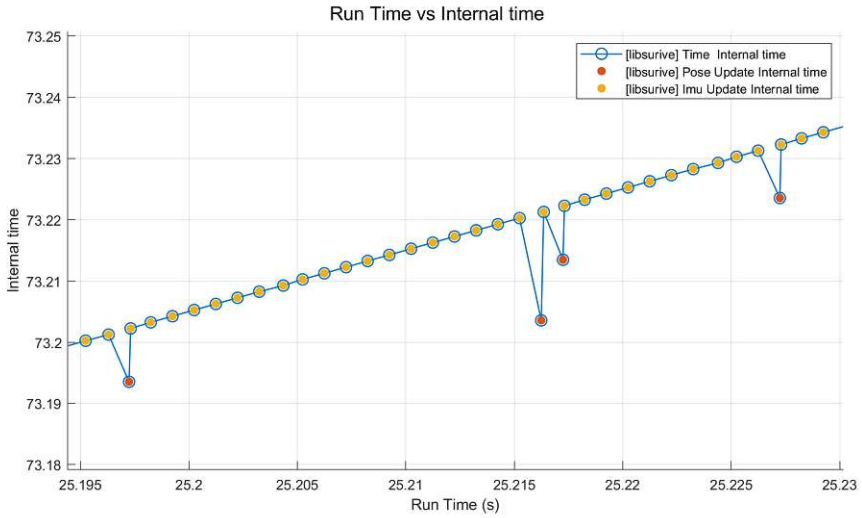


Figure 5.5: Internal time to run time comparison.
This figure maps the two *Internal Time* clocks for IMU and pose updates to libsurvive's *Run Time* for comparison. It can be seen that the *Internal Time* for pose estimations lies back in time.

Furthermore the LHTS provides measurements for each beacon and libsurvive estimates a pose, but the time interval between these measurements is not consistent, which causes the sampling rate to vary. This is an issue for the DESP, diminishing its ability to estimate between pose updates, as it requires a stable sampling rate. In order to prevent overshoots the implementation limits the parameter h to a maximum of 1 and uses the time stamp of the first IMU update after each pose update as prediction start.

In order to integrate the accelerometer readings for the ESKF they have to be rotated into the reference frame. Valve's Index tracking devices do not use magnetometers

therefore the estimated orientation of libsurvive is used to rotate these readings. The libsurvive Kalman filter is applied at the level of lightcap data, and without it we are left with a less accurate orientation (see figure 5.6), which causes acceleration integration errors that lead to an inaccurately estimated velocity. The discrepancy between the

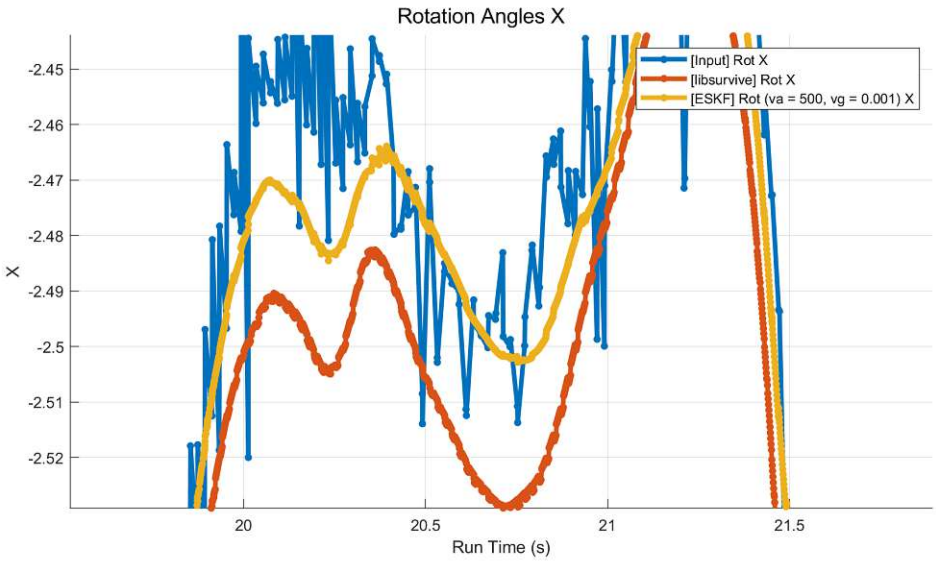


Figure 5.6: Rotation discrepancies.
This figure demonstrates the discrepancies of the rotation provided by libsurive and shows the angle on the axis x for comparison. [Input] shows the raw signal with disabled optimizations. [Eskf] shows the result of the ESKF filtered raw signal. [libsurvive] shows the corrected signal with enabled optimizations.

position velocity and the integrated velocity causes a breakout of the acceleration bias, which adds a lag to the integrated velocity (see figure 5.7) causing overshoots on turning points. This subsequently applies also to the position. In order to mitigate this effect a higher acceleration bias variance has been used.

5.3.2 DESP

The DESP is applied to the input signal and the results are plotted and then evaluated with Matlab. Again, different plots for the control parameter α were created and compared side by side to find the optimal value for the given sample. The figure 5.8 shows a sample of DESP filtered position results and the figure 5.9 shows rotation results. The optimal value for the filter parameter α turned out to be 0.1, as illustrated in the two figures, it filters the signal but also introduces a negligible lag. The results are similar to those of the simulated tracking data introduced in chapter 5.2.2.

The main difference is the unstable sampling rate of the input signal, which worsens the

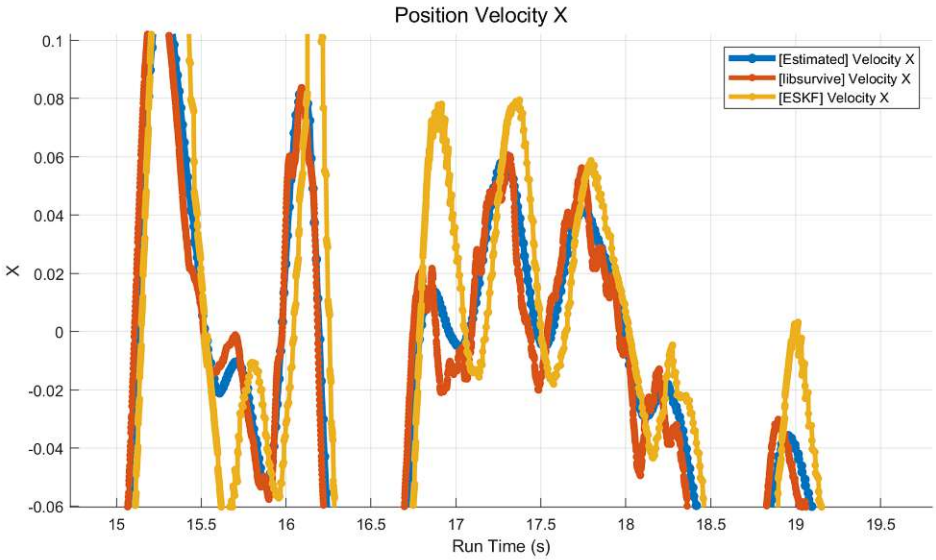


Figure 5.7: Integrated velocity comparison.
This figure illustrates the lag of the integrated velocity due to integration errors for a sample on the x axis.

predicted estimate between pose updates. The filter results are not stable enough to be used for the user study, especially the noisy rotation results are causing dizziness when used within the VE.

In addition to the results documented in chapter 5.3.2, we can add the following points:

- The irregular sampling rate and improperly aligned timestamps cause the prediction to not work properly.
- The results are too noisy, causing dizziness. Especially the noisy rotation is an issue for the subjective experience.

Therefore, the filter is discarded for the user study.

5.3.3 ESKF

A stationary input signal is used for the ESKF to evaluate the correct accelerometer and gyroscope bias estimation and to verify the proper rotation of the acceleration. The figure 5.10 illustrates the rotation of the input acceleration signal provided by the IMU. It is first rotated by the pose rotation and then adjusted by subtracting the gravity and the estimated accelerometer bias. The resulting acceleration is approximately zero, which is as expected for a stationary signal. The figure 5.11 shows the result of the

gyroscope's angular velocity subtracted by the estimated bias. The result of the angular velocity for a stationary signal is, as expected, close to zero.

The stationary variances of position and orientation are around 10^{-8}, and those of the accelerometer and gyroscope around 10^{-6}. However, the variances for position and orientation of tracked devices in movement deviate greatly from the stationary variances. This can also be observed using raw libsurvive tracking within the VE, holding still greatly improves the subjective stability of the position and orientation. The optimal parameters for the ESKF were determined by trial and error on a sample with movement data. We started out with a variance of 0.001 for all parameters and ended up with:

$$p_n = 0.001 \quad r_n = 0.001 \quad a_n = 0.01 \quad \omega_n = 0.01 \quad a_w = 100 \quad \omega_w = 0.01 \tag{5.23}$$

Due to errors of the orientation mentioned in section 5.3.1, we reduce the trust in the acceleration and angular velocity by increasing their variances compared to the position and orientation. The consequences of the resulting bias breakouts are mitigated by increasing the bias variances of the angular velocity and the acceleration.

Looking at the results of the rotation for the HMD (see figure 5.9), we can see a clear improvement to the input signal. Especially the stability of the rotation is important for subjective experience. Moving on to the results of the position for the HMD (see 5.8) we can see the introduction of a negligible lag on the filtered signal, but also an improvement compared to the noisy input signal. The increased acceleration bias prevents the result from drifting, but also makes it more noisy. As a consequence, the deviation of the rotation results in an inaccurate velocity, which requires an increase of the acceleration bias variance and prevents proper filtering of the position signal. The results of the KCs are very similar to those of the HMD and are not further illustrated here.

The results allow this filter to be used for the user study and can be summarized as follows:

- The filtering of the rotation is a significant improvement compared to its noisy input signal.
- The filtering of the position is a minor improvement compared to its noisy input signal.
- The filter is not able to compensate for the inaccurate rotation, causing integration errors of the velocity. The resulting bias breakouts of the ESKF cause drifts. The measures taken to mitigate this effect add a negligible delay and prevent proper filtering of the position.
- The end result is subjectively good enough to be evaluated within the user study.

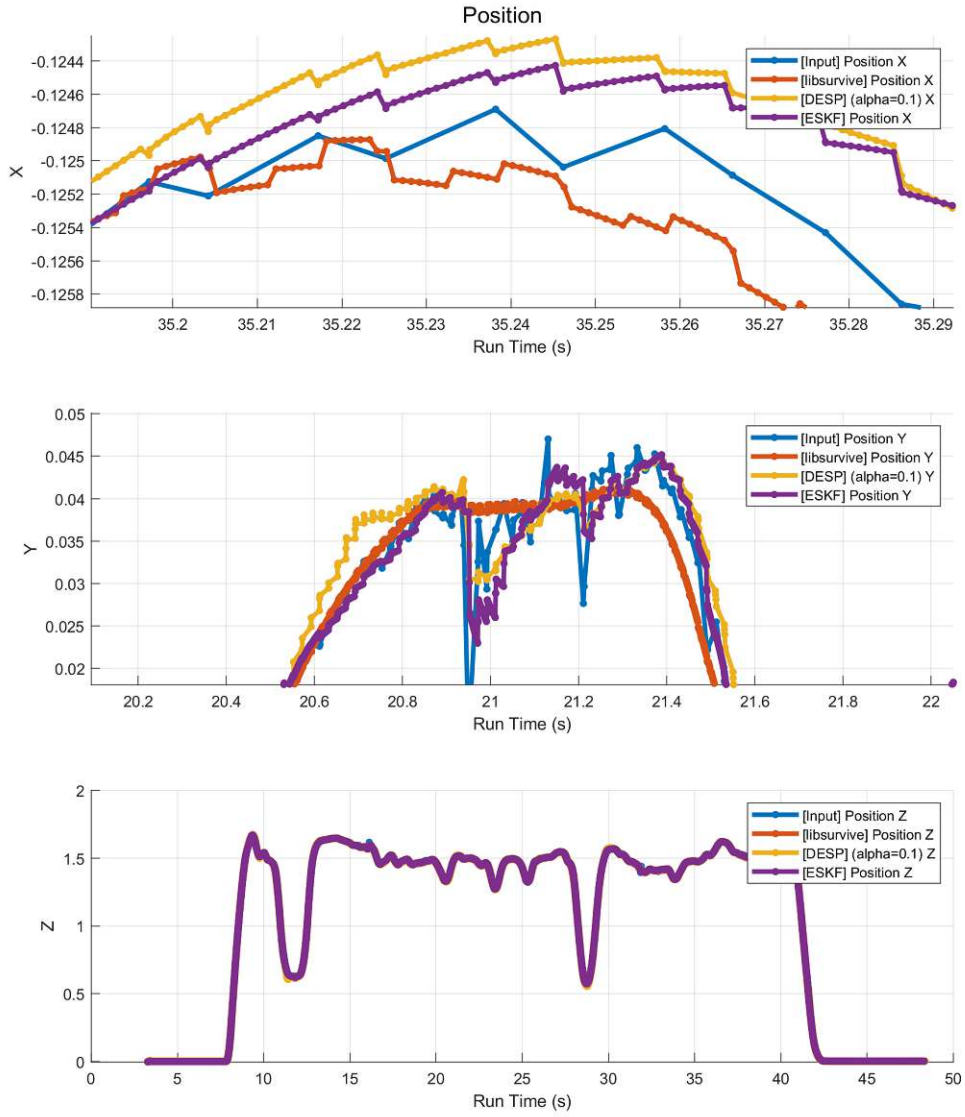


Figure 5.8: Filtered positions sample of libsurvive HMD tracking data.

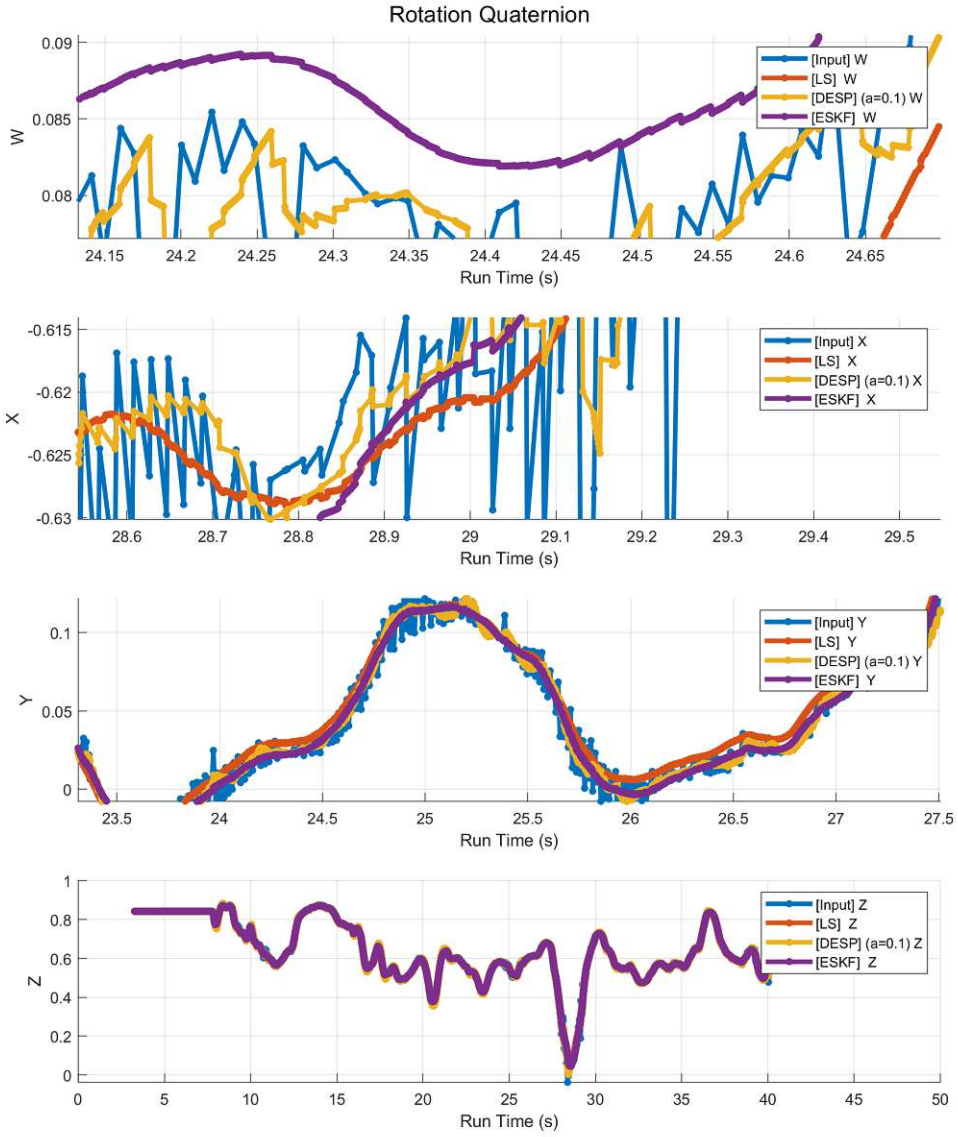


Figure 5.9: Filtered rotation sample of libsurvive HMD tracking data.

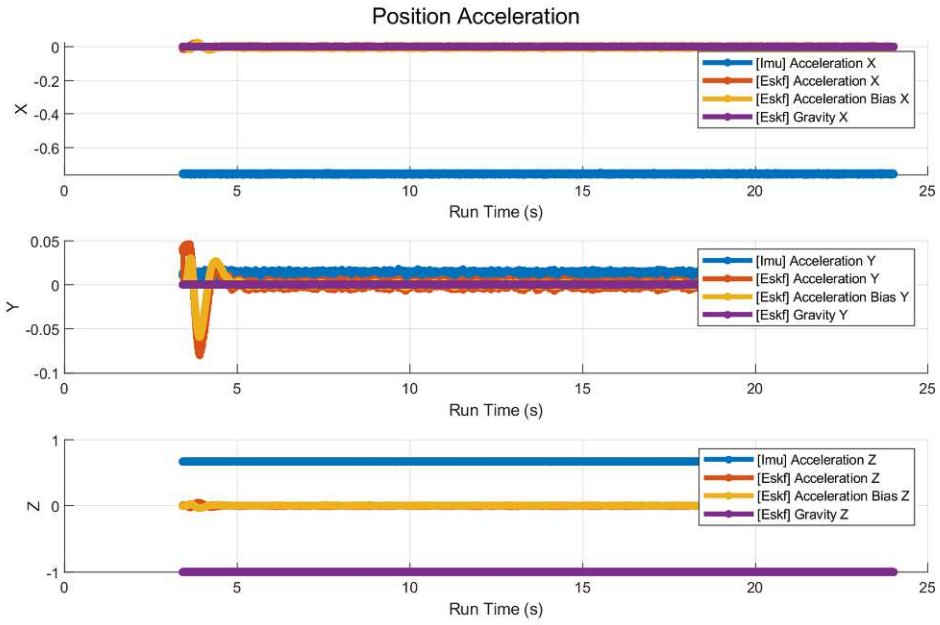


Figure 5.10: libsurvive acceleration of a stationary object sample.

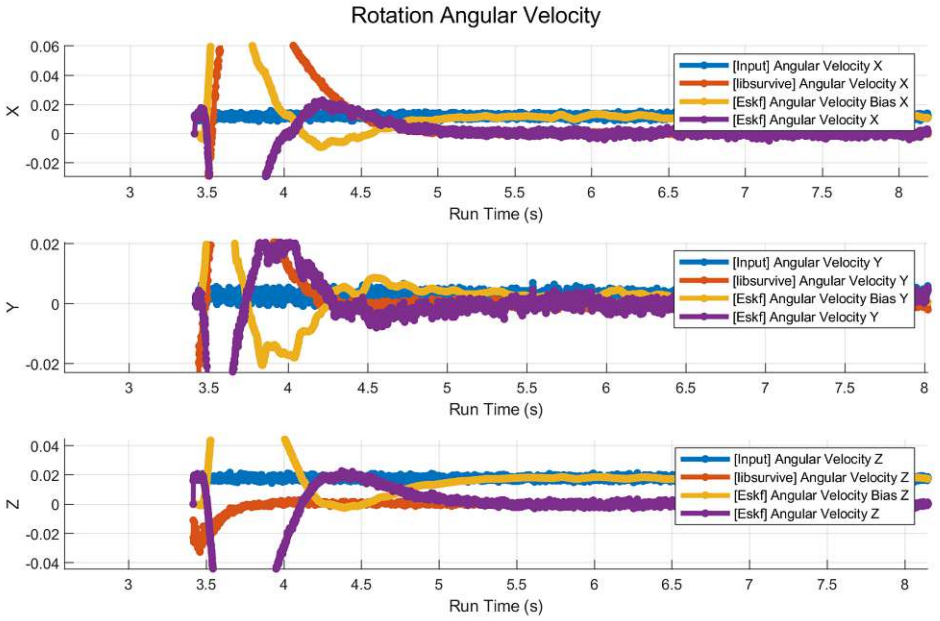


Figure 5.11: libsurvive angular velocity of stationary object sample.

5.4 User Study

This section discusses the results of the conducted user study (see section 4.6.5) within the VE (introduced in section 4.6.4) to investigate the systems performance and user acceptance. This work utilizes a within-subjects factorial design in which the independent variables are the tracking systems or filters and the tasks to complete. A subject refers to a participant within this context and the design allows for the manipulation of independent variables within subjects. This means that each participant is tested in all conditions of the experiment. The dependent variables are the number of interactions with each task and the completion time for each task.

It has to be noted that the filter parameters of the ESKF were adapted for the user study to further reduce the noise of the underlying tracking data delivered by libsurvive. This was necessary because the performance of the cube placement tasks was impacted by the remaining noise, which made it more difficult to place the cubes. As a result, there is a negligible delay and a slightly less accurate but more stable position and orientation. The following ESKF parameters were used for the experimental user study:

$$p_n = 0.01 \quad r_n = 0.01 \quad a_n = 0.001 \quad \omega_n = 0.001 \quad a_w = 100 \quad \omega_w = 0.01 \tag{5.24}$$

5.4.1 Subjects

19 people have participated in the user study, 6 of these participants identified as female and 13 as male. The age of the participants ranged between 28 to 65. 12 participants stated that they had moderate or more experience with computer games and 7 had moderate or more experience with virtual reality. The order in which the systems were tested was unknown and randomized for each participant. 10 participants started with SVR and 9 with the ESKF applied to the libsurvive tracking data. One person stated that they felt extremely dizzy while conducting the tasks for a given system, but they still were able to finish the session.

5.4.2 Collected Data and Statistical Tests

The quantitative variables, task completion time and task interactions, are recorded by the VRA. The participants' age, which is also a quantitative variable, is gathered by the questionnaire. The task completion time has a continuous value, and the number of task interactions as well as the participants' age have discrete values. Furthermore, the tracking system and the participants' gender have categorical nominal values. The tracking system is collected by the VRA and related to the quantitative variables, and the participants' gender is queried through the questionnaire. The rest of the categorical variables are collected through the questionnaire and have ranked ordinal values. Namely computer game experience, virtual reality experience, dizziness, nausea, HMD and KC's tracking errors, and task difficulty.

Paired t-test

The paired t-test allows to compare collected pairs of dependent quantitative data. It is used to compare the means of two groups with multiple samples, where each observation in one group is paired with an observation in the other group, in order to determine if there is a significant difference between the means of the two groups. The null hypothesis H_0 assumes that the mean $\overline{x_d}$ of the random variable $x_d = x_1 - x_2$, which represents the differences between each pair of observations, is normally distributed and zero. The alternative hypothesis H_1 assumes that the mean $\overline{x_d}$ is unequal to zero.

$$H_0 : \mu_d = \mu_1 - \mu_2 = 0 \tag{5.25}$$

$$H_1 : \mu_d \neq 0 \tag{5.26}$$

The t-statistic t is given by the mean $\overline{x_d}$ divided by the standard deviation s_d of differences for each pair of observations n (sample size):

$$t_{df} = \frac{\overline{x_d}}{s_d/\sqrt{n}} \tag{5.27}$$

The *degrees of freedom df* are defined as $n - 1$. The result of the t-test is given by the t-statistic t and the p-value p. The p-value represents the probability of obtaining a test-statistic by chance alone, the lower its value the more significant is the test. A p-value of 0.05 indicates that there is less than a 5% chance that the difference in the mean values was due to chance. A high t-value and a low p-value indicate the rejection of the null hypothesis. The null hypothesis is rejected if the absolute value of the calculated t-statistic is greater than the critical value of a t-distribution with $n-1$ degrees of freedom for a chosen significance level α. Not rejecting the null hypothesis means that there is no sufficient evidence to claim a significant difference, while rejecting the null hypothesis means that there is a significant difference between the means of the two groups.

Wilcoxon Signed-Rank Test

The Wilcoxon signed-rank test is a non-parametric test, used to compare the difference of two groups with multiple samples. It allows to compare the medians of two dependent distributions and is able to compare ranked values from a Likert scale. The test statistic is obtained as follows:

- First, the difference between the values of each dependent pair of observations is calculated.
- Next, a rank from 1 to the number of samples n is assigned to the absolute difference of each value pair.
- Pairs of equal values with a difference of zero are removed.
- All pairs with equal absolute differences form a group, the average of all the ranks within a group is calculated, and assigned to each pair as rank value.

H_1 The number of interactions when performing tasks is less with SVR, than with libsurvive and applied ESKF.

H_2 The time needed to complete tasks is less with SVR, than with libsurvive and applied ESKF.

H_3 The task performance is perceived less difficult with SVR, than with libsurvive and applied ESKF.

H_4 Participants perceive fewer tracking errors during a test run with SVR, than with libsurvive and applied ESKF.

H_5 The use of SVR causes less dizziness for users, than libsurvive with applied ESKF.

H_6 The use of SVR causes less nausea for users, than libsurvive with applied ESKF.

Figure 5.12: Tracking hypotheses

- Finally the signed ranks, which are the sums of absolute rank values for all negative and positive differences, are calculated separately.

The test statistic W, which is the smaller signed rank value, is then compared to the critical value from a Wilcoxon table for a chosen significance level α. The null hypothesis H_0 is rejected if the test statistic is less than or equal to the critical value, indicating sufficient evidence to claim a significant difference between the two groups. [23]

5.4.3 Goals and Hypotheses

It is the goal of this user study to evaluate the performance and user acceptance of libsurvive with the applied ESKF compared to SVR. The performance is measured in terms of the number of interactions and the completion time for each task. The user acceptance is assessed by perceived task difficulty, tracking errors, and experienced dizziness and nausea. The hypotheses for this experiment have been selected with the prior knowledge from the performance analysis of applying the ESKF to libsurvive tracking data in mind (see section 5.3.3), and are listed in figure 5.12.

5.4.4 Quantitative Evaluation Results

This section compares SVR to libsurvive with disabled internal optimizations and applied ESKF. The comparison aims to verify if there are significant differences between the two systems using paired t-tests. The tests compare the average completion time and number of interactions for each system.

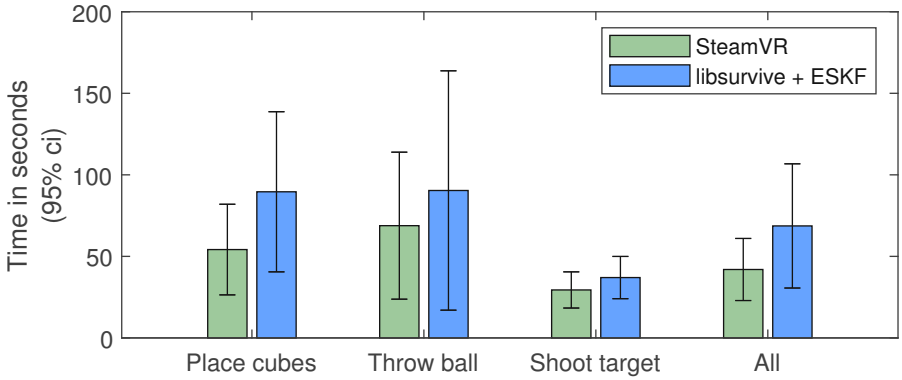


Figure 5.13: Average completion time by task or group

Task Completion Time

Figure 5.13 illustrates the average task completion time for all tasks 1 to 5, as well as for cube placement tasks 1 to 3, task 4 (throwing a ball) and task 5 (shooting a moving target), separately. The average task completion time for all tasks indicates a significant difference in favor of SVR between the two systems, with ($t_{18} = -3.2932, p = 0.0040415$). For the averaged result of tasks 1 to 3, SVR significantly outperforms libsurvive and applied ESKF with ($t_{18} = -3.32, p = 0.0038095$). This difference can likely be attributed to the remaining residual noise present in libsurvive tracking data of the KCs, which makes it more challenging to accurately place cubes and consequently leads to an extended time for cube placement. For task 4, there was no significant difference detected between the systems with ($t_{18} = -2.0566, p = 0.054519$). For task 5, SVR is also significantly better than libsurvive and applied ESKF with ($t_{18} = -2.5186, p = 0.021461$). This is most likely caused by libsurvive related tracking errors of the HMD and the KC, which are affecting the hand-eye coordination and making it harder to follow and hit the moving target.

Task Interactions

Figure 5.14 illustrates the average task interactions for all tasks 1 to 5, as well as for cube placement tasks 1 to 3, task 4 (throwing a ball), and task 5 (shooting a moving target), separately. The average task interactions for all tasks indicates a significant difference in favor of SVR for the two systems with ($t_{18} = -3.6084, p = 0.0020093$). For the averaged result of tasks 1 to 3, there was no significant difference detected between the systems with ($t_{18} = -1.9479, p = 0.067189$). Again for task 4, there was no significant difference detected between the systems with ($t_{18} = -0.65727, p = 0.51932$). For task 5, SVR was significantly better than libsurvive and applied ESKF with ($t_{18} = -3.8832, p = 0.0010894$). Again, this is most likely caused by libsurvive related tracking errors of the HMD and the KCs, which affect the hand-eye coordination, make it harder to hit the moving target, and cause more misses.

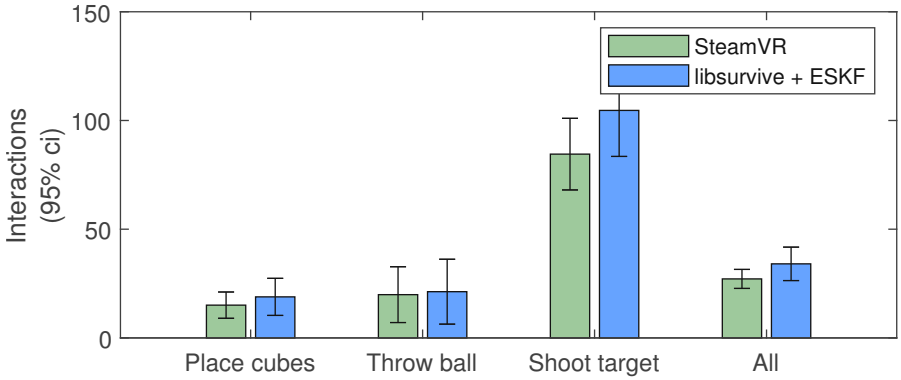


Figure 5.14: Average interactions by task or group

5.4.5 Subjective Evaluation

This section once again compares SVR to libsurvive with disabled internal optimizations and applied ESKF. Similarly to the previously discussed analysis of the gathered quantitative performance data, the goal is to compare the two systems and verify if there are significant differences. This time, the Wilcoxon signed-rank test is utilized to analyze the subjective feedback provided by the participants in the form of Likert scales, with a significance level of 0.05. The questions and possible answers were introduced in section 4.6.5. The participants provided feedback on the perceived dizziness, nausea, difficulty and tracking errors.

Dizziness and Nausea

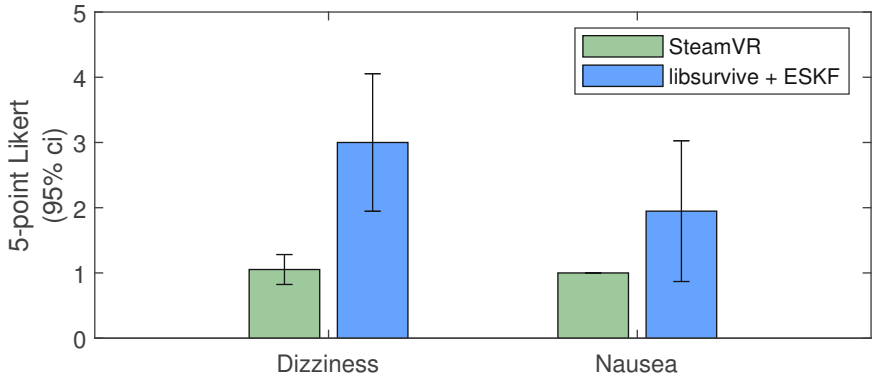


Figure 5.15: Average dizziness and nausea.

The results of the perceived average dizziness and nausea are illustrated in figure 5.16. The participants experienced significantly less dizziness, with ($W = 0, p = 0.00022311$), and significantly less nausea, with ($W = 0, p = 0.0019531$), when using SVR compared

to libsurvive with applied ESKF. The dizziness and the more severe nausea is most likely caused by a discrepancy between executed movements and their perceived representation in the VE. Even libsurvive with enabled optimizations introduces dizziness or a feeling of "something is off", this is probability worsened with disabled optimizations and the measures taken to reduce jitter with the ESKF.

Task Difficulty

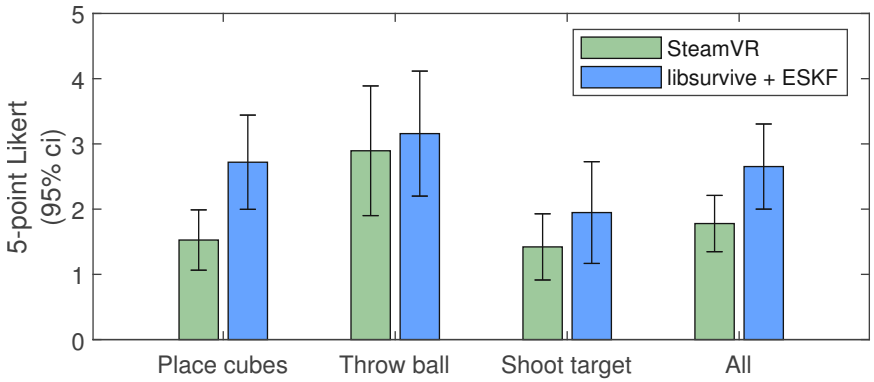


Figure 5.16: Average task difficulty.

Figure 5.16 illustrates the results for the perceived average task difficulty of all tasks 1 to 5, the place cube tasks 1 to 3, the throw a ball task 4, and the shoot a moving target task 5. All tasks total, were perceived significantly less difficult using SVR with ($W = 0, p = 0.0039062$). This also applies to the cube placements tasks 1 to 3 with ($W = 0, p = 0.00097656$). Especially the close-up interactions required to place the cubes can become more difficult with the presence of tracking errors. The occurrence of spatial distortions and the remaining residual noise in libsurvive tracking data for the HMD and KCs can make it more challenging to accurately place cubes. Again similar to the quantitative data results, there was no significant difference detected in the task difficultly between the two systems for the task 4 (throw a ball) with ($W = 21.5, p = 0.37305$). The difficultly of task 5 (shoot a moving target) was perceived significantly less challenging when using SVR with ($W = 0, p = 0.015625$). Again, this is most likely due to spatial distortions of the HMD and KCs, as well as dizziness introduced by latency, both affect the hand-eye coordination.

HMD Tracking Errors

This section presents the results of the perceived tracking errors by participants using the HMD. Figure 5.17 shows the average perceived tracking errors with the HMD for all tracking errors, as well as for spatial distortion, spatial jitter, drift or stability, latency, and latency jitter, separately. The overall perceived tracking errors for the HMD were significantly lower when using SVR, compared to libsurvive with applied ESKF, with

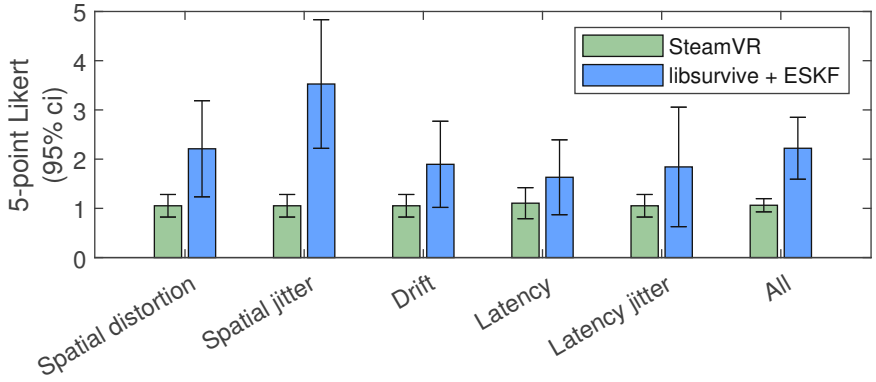


Figure 5.17: Average HMD tracking errors

($W = 0, p = 0.00019187$). The perception of spatial distortions was significantly less for SVR with ($W = 0, p = 0.00024414$). The pose estimation algorithm of libsurvive with disabled optimizations can cause spatial distortions, especially while the user is moving fast or the HMD is partially occluded. Participants experienced significantly more spatial jitter while using libsurvive and ESKF compared to SVR with ($W = 0, p = 0.00017376$). The measures taken to address the previously discussed acceleration integration errors reduce the ESKF's ability to eliminate the existing jitter from libsurvive tracking data. Drift or stability issues were significantly less perceived on SVR with ($W = 0, p = 0.0019531$). The perception of a latency between movement and their representation in the VE was significantly less for SVR with ($W = 3.5, p = 0.027344$). The measures taken to reduce the position jitter most likely introduce a minor delay, which is only noticeable during fast movements. Participants also perceived significantly less latency jitter when using SVR with ($W = 0, p = 0.0078125$).

KC Tracking Errors

This section presents the results of participants' perceived tracking errors using the KCs. Figure 5.18 shows the perceived average tracking errors of KCs for all tracking errors, as well as spatial distortion, spatial jitter, drift or stability, latency, and latency jitter. Similar to the HMD, participants perceived significantly fewer tracking errors for KCs when using SVR compared to libsurvive and applied ESKF with ($W = 0, p = 0.00019187$). Participants perceived significantly fewer spatial distortions for KCs when using SVR with ($W = 0, p = 0.00024414$). Similar to the HMD, the pose estimation algorithm of libsurvive with disabled optimizations can cause spatial distortions, particularly when a KC is partially occluded. Spatial jitter was perceived significantly less often when using SVR with ($W = 0, p = 0.00015195$). Again similar to the HMD, the measures taken to address the previously discussed acceleration integration errors reduce the ESKF's ability to eliminate the existing jitter from libsurvive tracking data for KCs. Drift or stability errors were perceived more often with libsurvive and applied ESKF than with

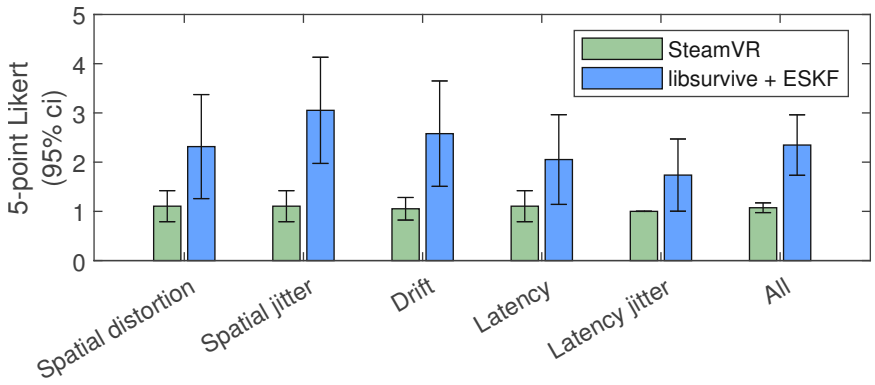


Figure 5.18: Average KC tracking errors

SVR, with ($W = 0, p = 6.1035e - 05$). If a KC was occluded the rotation could start to spin, this was a common issue with libsurvive and could occur when participants bent over to pick up a cube. Latency was perceived significantly less for the SVR system with ($W = 0, p = 0.00048828$). Again similar to the HMD, the measures taken to reduce the position jitter most likely introduce a minor delay, which is only noticeable during fast movements of a KC. The latency was most noticeable for a KC during task 4 (Throw a ball). Finally, participants also perceived significantly less latency jitter when using SVR with ($W = 0, p = 0.00097656$).

CHAPTER 6

Conclusion and Outlook

6.1 Tracking data

The LHTS with Valve's Index has been used as data source for this work, in order to get real tracking data of an optical/inertial system. SVR only provides filtered tracking data and does not provide IMU measurements. The library libsurvive has been used as an alternative to access LHTS data, including IMU measurements. The libsurvive optimizations were disabled in order to work with noisy tracking data. The IMU measurements are necessary for the ESKF, because it has the ability to fuse optical and inertial tracking data. libsurvive is experimental and has a few problems with disabled optimizations, as mentioned in section 5.3.1. The internal Kalman filter of libsurvive filters the lightcap data prior to the estimation of a pose. Disabling this internal filter causes additionally to the expected spatial jitter, spatial distortions and inaccurate poses. The resulting inaccurate rotation causes problems for the integration of the acceleration.

libsurvive and its output with disabled optimizations is unstable. The noise and spatial distortions turned out to be too much for both filters to compensate. There are two ways to deal with the LHTS and libsurvive, either start with the implementation of a custom poser and a much more complex Kalman filter algorithm, similar to what libsurvive does, or find another tracking source that has fewer issues and verify if the introduced filters are capable of producing better results.

6.2 Virtual Reality Application

The subjective feedback from users regarding the VE was very positive, and the tasks were perceived as entertaining and fun. However, for future work, there are a few areas that should be improved. A better collision handling system could be implemented to improve close-up interactions, such as the placement of cubes. Currently, it is possible to

grab an object and push it inside another object. Upon release, the two objects bounce of each other, which caused frustrations for some participants. Additionally, a better visualization for interactions could be implemented to indicate when an object can be grabbed. The area of effect for a grab interaction extends the volume of the KC model, which was not apparent for participants. The task 4 (throw a ball) was perceived as too difficult for most users, because the ball itself had no momentum on its own. This lack of momentum made it more challenging to accurately throw the ball. Additionally, the distance to the hoop was perceived as too far for some participants.

6.3 Double Exponential Smoothed Prediction

The results for simulated tracking data (see section 5.2.2) show that the filter could improve the noisy input, but introduces a lag in the signal. However, the result is still very noisy.

The results for libsurvive tracking data (see section 5.3.2) turned out to be to noisy and caused extreme dizziness due to the introduced lag. The prediction in between updates turned out to be not working properly for libsurvive tracking data due to the the varying sampling rate. The results of the DESP are not sufficient to filter libsurvive tracking data. The DESP was therefore discarded for the user study.

The DESP is able to remove high frequency noise but not to the extend that the results would be sufficient enough for a VRA. A control parameter α with a value of 0.1 already introduces a lag in the signal, yet the result was still too noisy. This applies to both simulated and libsurvive tracking data. Another problem of the filter is that it only provides one parameter for filtering level and trend. The sample rate of a signal makes a difference for the results, the more data is available the better an exponential filter can estimate the underlying trend and remove high frequency noise. For future work, exploring other exponential filters could lead to better results. The potential to apply different control parameters to the level and trend, as well as the independence from a fixed sampling rate, could potentially lead to improved outcomes. However, if the input data is to noisy other exponential filters might still not provide acceptable results.

6.4 Error-state Kalman Filter

The ESKF results for simulated tracking data (see section 5.2.3) show a significant improvement compared to its noisy input. Both the position and rotation results are close to the true signal, this holds true even for faster movements. The algorithm accurately estimates the arbitrary chosen accelerometer and gyroscope biases and can estimate the velocity by integrating the simulated acceleration. The resulting velocity is accurate and enables to estimate the position in between simulated optical updates.

The results for libsurvive tracking data (see section 5.3.3) show an improvement compared to the noisy input. The rotation results show a significant improvement. However,

the position shows only a minor improvement due to the issues discussed in section 5.3.1. The incorrect rotation leads to an incorrect velocity, which ultimately leads to a drift in the position. The measures taken to mitigate this drift prevent a proper filtering of the position.

The results of the experimental user study show that there is a significant difference between SVR and libsurvive with applied filter.

Looking at the results for tracking errors, the hypothesis $H4$ is supported. Participants perceived significantly less tracking error for the HMD and KCs while using SVR. When examining the perceived tracking errors for libsurvive with applied ESKF, a few things become apparent. Spatial jitter was the most dominant perceived error, which occurred for both the HMD and the KCs. The measures taken to prevent position drifts, as discussed earlier, reduced the ESKF's ability to remove the remaining jitter from the position data. Another two major tracking errors were spatial distortion and the related inaccurate orientation of the HMD. Some participants could even detect that the orientation of the HMD was not always correct, as it gave them the impression that they are standing crooked within the VE. Latency was experienced by some participants, which was mostly noticeable during the task 4 (throw a ball). Faster movements of throwing a ball for this task made the latency become more noticeable. Drifts could also occur for the KCs, with occluded KCs showing orientation spinning.

Moving on to the perceived dizziness and nausea, the results also support the hypothesis H_5 and H_6 as SVR caused significantly less dizziness and nausea. The emergence of dizziness and nausea was due to a few factors. The delay between a movement and its representation could often not be consciously identified, but was perceived as dizziness or nausea. Even the use of libsurvive with enabled optimizations can cause a slight sense of dizziness, and the measures to remove drifts for the position worsened this delay. Furthermore, the spatial distortion of the HMD and the inaccurate orientation also contribute to the feeling of dizziness and nausea.

The results also support the hypothesis H_1 and confirm that the average number of interactions for all tasks is significantly less with SVR than libsurvive with disabled optimizations and applied ESKF. However, there was no significant difference detected for the tasks 1 to 4. The number of interactions for task 1 to 3 are not as affected, because participants pick up a cube and release it once it is in the right spot, reducing the number of required interactions. The time factor to place a cube is more important, as we will see later. The task 4 was more challenging in general, but once a participants had figured out how to throw the ball, the number of interactions and the completion time, as we see later, were not significantly different.

The user study results support the hypothesis H_2 and confirm that the average completion time for all task is again significantly lower with SVR. The only exception was task 4 (throw a ball), where no significant difference could be detected. The differences can be attributed to the remaining noise, the occurring spatial distortions but also the incorrect rotation. Participants took more time in order to be more accurate and to

counteract the feeling of dizziness.

Finally, the hypothesis H_3 is also supported by the findings and shows that performing tasks with SVR was significantly less challenging. Again, the only exception here was task 4 for which no significant difference could be detected. This means that, as excepted, the occurrence of tracking errors made performing a task more difficult for participants.

As already mention in section 6.1, for future work on the ESKF another tracking data source is required. The results with simulated tracking data show that the filter is capable of significantly improving the tracking data, but falls short with libsurvive tracking data because of the mentioned issues.

List of Figures

List of Tables

Acronyms

API Application Programmming Interface. 13, 65, 82, 84, 85, 87

CCD Charge Couple Device. 10

CLI Command Line Interface. 81, 82, 84, 92, 101

DESP Double Exponential Smoothed Prediction. vii, ix, 29, 31, 63, 65–67, 81, 87, 95, 99, 110, 112, 113, 115, 116, 118, 119, 134, 137, 139

DLL Dynamic Link Library. 84, 95

DOF Degrees Of Freedom. ix, 7, 13, 20

ESKF Error-state Kalman Filter. vii, ix, 50–52, 55, 57, 58, 63, 65, 69–72, 81, 88, 95, 99, 110, 111, 113, 115, 116, 118–121, 125, 127–131, 133–137, 139

FOV Field of View. 5

GUID Globally Unique Identifier. 108

HMD Head Mounted Display. 64, 78–80, 84, 85, 96, 97, 100, 105–107, 109, 110, 121–123, 125, 128, 130–132, 135, 137, 138

IMU Inertial Measurement Unit. 11, 13, 50–52, 55, 65, 67, 68, 70, 72–74, 82–85, 87, 88, 99, 100, 109, 110, 112, 113, 118, 120, 133

INS Intertial Navigation System. 9, 137

JSON JavaScript Object Notation. 81, 91, 93, 106, 108

KC Knucke Controller. 64, 78, 80, 84, 96, 97, 100, 105, 106, 108–110, 121, 125, 128, 130–132, 134, 135, 138

LED Light-Emitting Diode. 11, 12

LHTS Lighthouse Tracking System. vii, ix, 11–13, 64, 65, 77, 78, 80, 82, 100, 118, 133, 137

MMSE Minimum mean square error. 44

ODE Ordinary Differential Equation. 20

PDF Probility density function. 58, 59

PSD Position Sensing Detector. 10, 11

PTS Prediction Time Steps. 66–69

SVR Steam Virtual Reality. 64, 66, 78–80, 95, 97, 99, 104, 125, 127–133, 135, 136

TFF Tracking Filter Framework. vii, ix, xii, 1, 2, 13, 63–66, 77–80, 83, 95, 96, 99–102, 104, 109, 110, 137

UE Unreal Engine. 2, 64, 66, 78, 80, 82, 90, 91, 93, 95, 96, 98–100, 102, 106, 109, 110, 137

UKF Unscented Kalman Filter. 58, 59, 61

VE Virtual environment. ix, 1–6, 63, 64, 66, 79, 95, 97, 99–101, 104–106, 120, 121, 125, 130, 131, 133, 135

VRA Virtual Reality Application. ix, 66, 78–80, 82, 125, 134

Bibliography

[1] Ronald Azuma and Gary Bishop. „Improving static and dynamic registration in an optical see-through HMD“. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, pp. 197–204.

[2] Woodrow Barfield and Claudia Hendrix. „The effect of update rate on the sense of presence within virtual environments“. In: *Virtual Reality: Research, Development and Application* vol. 1, no. 1 (1995), pp. 3–16.

[3] David Berger. *Libsurive Architecture*. 2020. URL: https://github.com/cntools/libsurvive/wiki/Architecture (visited on 08/16/2020).

[4] Gary Bishop, Greg Welch, et al. „An introduction to the kalman filter“. In: *Proc of SIGGRAPH, Course* vol. 8, no. 27599-23175 (2001), p. 41. URL: https://courses.cs.washington.edu/courses/cse571/03wi/notes/welch-bishop-tutorial.pdf.

[5] Bruce L Bowerman and Richard T O'Connell. „Forecasting and time series: An applied approach. 3rd“. In: (1993).

[6] Robert Goodell. Brown. „Statistical forecasting for inventory control“. In: *McGraw-Hill,New York* (1959). ISSN: 01922262.

[7] Robert Grover Brown, Patrick YC Hwang, et al. *Introduction to random signals and applied Kalman filtering*. Vol. 4. Wiley New York, 2012.

[8] Steve T Bryson. „Effects of lag and frame rate on various tracking tasks“. In: *SIGGRAPH93 Course*. 3. 1993, pp. 16.1–16.12.

[9] Frank M Cardullo and Yorke J Brown. „Visual System Lags: The Problem, The Cause, The Cure“. In: *IMAGE V Conference*. 1990, pp. 31–42.

[10] David Berger Charles Lohr et al. *Libsurive*. 2021. URL: https://github.com/cntools/libsurvive (visited on 10/24/2021).

[11] Jan G De Gooijer and Rob J Hyndman. „25 years of time series forecasting“. In: *International journal of forecasting* vol. 22, no. 3 (2006), pp. 443–473.

[12] Michael F Deering. „Data complexity for virtual reality: Where do all the triangles go?“ In: *Proceedings of IEEE Virtual Reality Annual International Symposium*. IEEE. 1993, pp. 357–363.

[13] Sheldon M Ebenholtz. „Motion sickness and oculomotor systems in virtual environments“. In: *Presence: Teleoperators & Virtual Environments* vol. 1, no. 3 (1992), pp. 302–305.

[14] Peter Ferschin, Ingeborg Tastl, and Werner Purgathofer. „A comparison of techniques for the transformation of radiosity values to monitor colors“. In: *Proceedings of 1st International Conference on Image Processing*. IEEE. 1994.

[15] Thomas A Funkhouser and Carlo H Séquin. „Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments“. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 247–254.

[16] W. S. Stiles G. Wyszecki. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. John wiley & sons, 1982. ISBN: 0-471-02106-7.

[17] Mario Garcían. *Angular velocity from Quaternions*. 2022. URL: https://mariogc.com/post/angular-velocity-quaternions (visited on 02/27/2022).

[18] Morton Leonard Heilig. „El cine del futuro: the cinema of the future.“ In: *Presence Teleoperators Virtual Environ.* vol. 1, no. 3 (1992), pp. 279–294.

[19] J Helman. „Performance Requirements and Human Factors“. In: *SIGGRAPH*. Vol. 95. 6. 1995, pp. 1.19–1.32.

[20] Richard Holloway and Anselmo Lastra. „Virtual environments: A survey of the technology“. In: *SIGGRAPH95 Course*, no. 8 (1995), A.1–A.40.

[21] C. C Holt. „Forecasting trends and seasonals by exponentially weighted moving averages“. In: *ONR Memorandum* (1957). ISSN: 1467-1107. DOI: 10.1016/j.catcom.2010.10.019.

[22] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018. URL: https://otexts.com/fpp2/.

[23] Wouter Jansen et al. „Automatic calibration of a six-degrees-of-freedom pose estimation system“. In: *IEEE Sensors Journal* vol. 19, no. 19 (2019), pp. 8824–8831.

[24] Jan-Keno Janssen. *Virtual-Reality-Headset Valve Index im Test*. 2020. URL: https://www.heise.de/tests/Virtual-Reality-Headset-Valve-Index-im-Test-4717307.html (visited on 08/13/2020).

[25] Rambabu Kandepu, Bjarne Foss, and Lars Imsland. „Applying the unscented Kalman filter for nonlinear state estimation“. In: *Journal of process control* vol. 18, no. 7-8 (2008), pp. 753–768.

[26] Robert S Kennedy et al. „Profile analysis of simulator sickness symptoms: Application to virtual environment systems“. In: *Presence: Teleoperators & Virtual Environments* vol. 1, no. 3 (1992), pp. 295–301.

[27] Ji-Hwan Kim Ji-Hwan Kim et al. „Virtual Reality History, Applications, Technology and Future“. In: *Digital Outcasts* (2013). ISSN: 0010-4825. DOI: http://dx.doi.org/10.1016/B978-0-12-404705-1.00006-6. URL: https://www.cg.tuwien.ac.at/research/publications/1996/mazuryk-1996-VRH/TR-186-2-96-06Paper.pdf.

[28] Oliver Kreylos. *Lighthouse tracking examined*. 2016. URL: http://doc-ok.org/?p=1478.

[29] Ben Lang. *Next-gen Lighthouse Base Station Could Bring rapid cost reductions*. 2016. URL: https://www.roadtovr.com/next-gen-lighthouse-base-station-bring-rapid-cost-reductions/ (visited on 11/29/2016).

[30] Ben Lang. *Valve Shows off Miniscule Lighthouse Sensors*. 2016. URL: https://www.roadtovr.com/valve-shows-off-miniscule-lighthouse-sensors/ (visited on 01/25/2016).

[31] Jr. LaViola, Joseph J. „Double Exponential Smoothing: An Alternative to Kalman Filter-Based Predictive Tracking“. In: *Proceedings of the Workshop on Virtual Environments*. 2003, pp. 199–206. ISBN: 1-58113-686-2. DOI: 10.1145/769953.769976. URL: http://cs.brown.edu/people/jlaviola/pubs/kfvsexp_final_laviola.pdf.

[32] Tomasz Mazuryk and Michael Gervautz. „Two-step Prediction and Image Deflection for Exact Head Tracking in Virtual Environments“. In: *Computer Graphics Forum*. Vol. 14. 3. Wiley Online Library. 1995, pp. 29–41.

[33] Matja Mihelj, Domen Novak, and Samo Begu. *Virtual Reality Technology and Applications*. 2014. ISBN: 978-94-007-6909-0. DOI: 10.1007/978-94-007-6910-6.

[34] Mark R Mine. „Characterization of end-to-end delays in head-mounted display systems“. In: *The University of North Carolina at Chapel Hill, TR93-001*, no. 8 (1993).

[35] Paul Michael Newman. *EKF based navigation and SLAM*. 2016. URL: http://aims.robots.ox.ac.uk/wp-content/uploads/2016/02/AIMSEstimationNotes.pdf (visited on 03/31/2019).

[36] R. Pausch. „Human factors of virtual reality design.“ In: *SIGGRAPH93 Course*, no. 43 (1993), pp. 1.4.1–1.4.4.

[37] Clare Regan. „An investigation into nausea and other side-effects of head-coupled immersive virtual reality“. In: *Virtual Reality* vol. 1, no. 1 (1995), pp. 17–31.

[38] Michal Reintein. *From Bayes to Extended Kalman Filter*. 2015. URL: http://people.ciirc.cvut.cz/~hlavac/TeachPresEn/55AutonomRobotics/2015-05-04ReinsteinBayes-ekf.pdf (visited on 03/31/2019).

[39] JP Rolland, Y Baillot, and AA Goon. *A survey of tracking technology for virtual environments. Fundamentals of Wearable Computers and Augmented Reality, W. Barfield, T. Caudell, Eds*. 2001.

[40] H. Scheirich. „Stereoscopics - Principles and Techniques.“ In: Diploma Thesis, Vienna University of Technology, Austria. 1994.

[41] Mel Slater and Martin Usoh. „Representations systems, perceptual position, and presence in immersive virtual environments“. In: *Presence: Teleoperators & Virtual Environments,* vol. 2, no. 3 (1993), pp. 221–233.

[42] Mel Slater, Martin Usoh, and Anthony Steed. „Depth of presence in virtual environments“. In: *Presence: Teleoperators & Virtual Environments* vol. 3, no. 2 (1994), pp. 130–144.

[43] Joan Sola. „Quaternion kinematics for the error-state Kalman filter“. In: *arXiv preprint arXiv:1711.02508* (2017). URL: https://arxiv.org/pdf/1711.02508.pdf.

[44] K Stanney. *Handbook of virtual environments. Lawrence Earlbaum.* 2001.

[45] The SteamVR Tracking team. *SteamVR Tracking Technology Update.* 2017. URL: https://steamcommunity.com/games/steamvrtracking/announcements/detail/1264796421606498053 (visited on 06/17/2015).

[46] Colin Ware and Ravin Balakrishnan. „Target acquisition in fish tank VR: The effects of lag and frame rate“. In: *Graphics Interface.* 1994, pp. 1–7.

[47] Greg Welch and Eric Foxlin. „Motion tracking: No silver bullet, but a respectable arsenal“. In: *IEEE Computer graphics and Applications* vol. 22, no. 6 (2002), pp. 24–38.

[48] Alan Yates. *SteamVR's "Lighthouse" for Virtual Reality and Beyond.* 2015. URL: https://www.youtube.com/watch?v=xrsUMEbLtOs (visited on 05/19/2015).