TU WIEN Informatics

Low-cost Motion Capture Suit using Inertial Sensors

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Ahmed Othman BSc
Matrikelnummer 01325531

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Hannes Kaufmann

Wien, 31. Oktober 2023

Ahmed Othman BSc	Hannes Kaufmann

TU WIEN Informatics

Low-cost Motion Capture Suit using Inertial Sensors

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Ahmed Othman BSc
Registration Number 01325531

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Hannes Kaufmann

Vienna, 31st October, 2023

Ahmed Othman BSc	Hannes Kaufmann

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

Erklärung zur Verfassung der Arbeit

Ahmed Othman 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, 31. Oktober 2023

Ahmed Othman BSc

Acknowledgements

Motion capture is a topic I have been interested in for years. So, I would like to thank Prof. Hannes Kaufmann for allowing me to select such an interesting topic for my master thesis which pushed me to dig deeper into this subject and subsequently learn new valuable skills.
In addition to my two brothers, I want to especially thank my parents who have uplifted me and pushed me throughout my entire education and even through the difficult times. For this, I will be eternally grateful.
Finally, I would like to dedicate this work to the memory of my father.

Kurzfassung

Motion Capture Technologien existieren bereits seit mehreren Jahrzehnten und werden in verschiedenes Gebieten für unterschiedliche Zwecke verwendet. In der Unterhaltungsindustrie hat es die Erstellung von realistischen und komplexen Animationen, die manuell sonst sehr schwierig und zeitaufwändig zu erstellen wären, stark erleichtert. Über die Jahre wurden verschiedene Motion Capture Technologien erfunden und sie alle habe ihre Stärken und Schwächen.
Inertial Motion Capture ist eine kostengünstige Alternative, die sich auf Inertialsensoren für die Schätzung der Orientierung und Position eines Objektes im dreidimensionalen Raum verlässt. Inertialsensoren bestehen aus einem Gyroskop und einem Beschleunigungssensor und sind oft in inertiale Messeinheiten enthalten. Durch ihre Erforschung und Entwicklung in den vergangenen Jahren sind diese kleiner, leichter, günstiger, stromsparender geworden und bieten höhere Abtastraten. Dies macht sie ideal für die Entwicklung eines Motion Capture Systems. Die ausgegebenen Messwerte von diesen Sensoren leiden aber unter verschiedenen Verzerrungen, was ein Kalibrierungsverfahren notwendig macht, um diese Verzerrungen zu minimieren.
In dieser Arbeit habe ich eine komplett kabellose und konfigurierbare Inertial Motion Capture Lösung entwickelt mit einem roboterassistierten Kalibrierungsverfahren. Diese Motion Capture Lösung besteht aus mehreren selbstgebauten Bewegungstracker, die an dem Benutzer befestigt sind und die Messdaten kabellos an einen Computer versendet. Ich habe einen quaternion-basierten Extended Kalman-Filter als eine Sensordatenfusionsmethode implementiert, welches die Sensordaten verwendet, um die Orientierung des Bewegungstrackers zu schätzen.
Da ich keinen Magnetometer in meiner Trackinglösung verwende, ist es schwierig eine gute Genauigkeit für die Schätzung des Gierwinkels zu erreichen. Diese Genauigkeit wird beeinträchtigt durch akkumulierte Driftfehler über einen längeren Zeitraum bei der Schätzung dieses Winkels. Deswegen ist meine Motion Capture Lösung nur geeignet für das Aufnehmen kurzer Animationen für humanoide 3D Charaktere. Zum Schluss werde ich meinen entwickelten Bewegungstracker einer auf den Markt bereits angebotenen Trackinglösung gegenüberstellen. Die Ergebnisse zeigen, dass mit Ausnahme des Gierwinkels für die geschätzte Orientierung akzeptable Resultate erbracht werden.

Abstract

Motion capture technology has now existed for several decades and has been used in many different fields for a variety of purposes. In the entertainment industry, it has facilitated the creation of realistic and complex animations that, if created manually, would be too difficult and time-consuming. Different motion capture technologies have been invented over the years and they all have their strengths and weaknesses.
Inertial motion capture is a low-cost alternative that relies on inertial sensors to estimate the orientation and position of a tracked object in three-dimensional space. Inertial sensors are a combination of a three-axis gyroscope and a three-axis accelerometer and are often contained in devices called inertial measurement units. In recent years they have become smaller, more lightweight, cheaper, less power-consuming and offer high sampling rates which makes them ideal for building a motion capture system. However, the outputted measurements from these sensors suffer from distortion which means that there needs to be a calibration procedure in place in order to minimize these distortions from the measurements.
In this work, I developed a completely wireless configurable inertial motion capture solution with a robot-assisted calibration procedure. This motion capture solution consists of multiple motion trackers that are attached to the capture subject and wirelessly transmit the motion data to a receiving computer. I implemented a quaternion-based Extended Kalman filter as a sensor fusion method that uses the inertial data to estimate the orientation of the motion tracker.
Due to the absence of a magnetometer sensor in this tracking solution, it is difficult to maintain a good enough accuracy when estimating the yaw angles of the motion trackers which leads to accumulated drifting errors over time. Therefore, this solution is only suitable for recording short animations for humanoid 3D characters. Furthermore, I compared my developed motion trackers to an existing commercially available tracking solution and the results indicate, with the exception of the low accuracy of the yaw angle estimation, acceptable orientation estimates.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

In this diploma thesis, I will present a low-cost and low-weight motion capture solution using inertial measurement units (IMUs).
Motion capture is used across different fields and industries where the goal is to record and sample the motions of humans or animals in three-dimensional space [KW08]. This provides a major advantage when trying to create realistic and complex animations in a fast and efficient way [Rah18] or when trying to analyze the motion data for research purposes. There are several available motion capture technologies that have been developed over the years such as optical, mechanical or magnetic motion capture systems where each technology has its advantages and disadvantages in terms of accuracy and cost [KW08].
Inertial motion capture solutions have emerged as low-cost alternatives that are also more versatile, portable, easy to setup and do not suffer from expensive post-processing which is ideal for real-time applications [VP20].
Because of their low cost, small weight and size, micro-electromechanical system inertial sensors have become increasingly popular in recent years. They offer high sampling rates and allow the user to extract position and orientation information [CGD+19]. IMUs, which contain these inertial sensors, can be placed on the user's body or interactable objects and transfer the sensor data to a virtual environment via a wireless connection such as Wi-Fi. Unfortunately, measurements extracted from these sensors are often distorted which could lead to noticeable errors when estimating the orientation of the sensor. Therefore, a calibration procedure is necessary in order to minimize these errors [TPM14].

1.2 Aim of the Work

The aim of this work is to track the user's movement using a self-developed inertial motion capture suit that will transfer the orientation data to a receiving motion capture application running on a computer and contains a full rigged 3D character. This application will communicate with all motion trackers simultaneously and receive real-time inertial measurement unit (IMU) data. Each of these IMUs will be connected to a microcontroller powered by a rechargeable battery, which are all encompassed in a self-designed 3D printed chassis. For the microcontroller, I will use the cheap and low-power consuming ESP32, which has an integrated Wi-Fi module and, for the IMU, I will use the MPU-6050. Additionally, I will design a Printed Circuit Board (PCB) which will allow me to efficiently and compactly assemble each motion tracker.
I plan to track the following body parts: head, left upper arm, left lower arm, right upper arm, right lower arm, chest, tailbone, left upper leg, left lower leg, right upper leg and right lower leg
Each motion tracker will be attached to all the above-mentioned body parts or other objects the user wishes to track. The IMU data will be transferred via Wi-Fi using User Datagram Protocol (UDP) communication to the receiving application. Then, I can estimate the orientation of each individual object or body segment. In order to estimate the orientation of an IMU sensor, I implemented a quaternion-based Extended Kalman filter which takes the IMU data as input and outputs reliable rotation data parameterized as a quaternion.
Due to the mentioned distortions of the IMU measurements, I will introduce a robot-assisted calibration procedure that aims to estimate the sensor errors of each IMU which can be then incorporated as additional parameters in my attitude estimation approach. The self-designed robot has an arm with an IMU attached that can rotate around the roll, pitch and yaw axes which is an important part of the data collection process in order to estimate the calibration parameters.
Finally, I plan to provide an extensive technical evaluation of my motion capture system where I will compare my motion trackers to an existing commercially available tracking solution (HTC Vive tracker). Then I will analyze the accuracy of the full-body motion capture results.

1.3 Structure of the Work

This thesis has the following structure. In chapter 2, I will explain important theoretical concepts that are utilized in this thesis. I will give an overview on motion capture and inertial sensors. Then, I will explain aspects of sensor fusion and the Kalman filter algorithm. I will also cover angular velocity integration methods and rotation parameterization using Euler angles and quaternions. Finally, I will give a brief overview on the numerical minimization algorithm which I will use in my proposed calibration procedure. In chapter 3, I will dive into the related work and present already existing inertial motion capture solutions as well as state-of-the-art research papers describing

the usage of IMUs for human motion tracking. Chapter 4, will present the methodology behind my software and hardware implementation of my proposed inertial motion capture system. Chapter 5 and 6 will discuss the implementation details for the hardware and software solutions described in this thesis, respectively. In chapter 7, I will provide a technical evaluation of my inertial motion capture system and highlight its strengths and weaknesses. Finally, in chapter 8, I will summarize the project and provide suggestions for possible improvements for future work.

CHAPTER 2

Theoretical Background

In this chapter, I will describe important theoretical concepts that are utilized in this thesis. First of all, I will give a general overview on motion capture and describe the different technologies that exist along with their advantages and disadvantages. Then, I will describe inertial sensors and explain how accelerometers and gyroscopes work on a higher level. Afterwords, I delve into the topic of sensor fusion and describe the complementary filter and the Kalman filter. In the Kalman filter section, I will also highlight the differences between the regular Kalman filter and the Extended Kalman filter algorithm. Subsequently, I will describe numerical methods for performing angular velocity integration, primarily using Euler's method and the Runge-Kutta method. In the following section, I will explain how the orientation of a rigid body can be parameterized in three-dimensional space using Euler angles and quaternions. Finally, I will write about the Levenberg–Marquardt algorithm which I will use in my calibration algorithm.

2.1 Motion Capture

Motion capture can be seen as the recording and sampling of human motion or the motion of animals and inanimate objects as data in three-dimensional space. The development of contemporary motion capture technology can be attributed to research in the fields of medical science, military and computer generated imagery (CGI) where it used for various purposes [KW08]. In the movie, gaming and animation industry, it is used as a mean to speed up the animation process and achieve high quality animations that would otherwise be too complex to create manually [Rah18]. There are different categories of motion capture systems which I will describe in the following sections.

2.1.1 Optical Motion Capture

Optical motion capture systems usually consist of a computer that controls multiple cameras responsible for capturing the motions of a subject. Often times, markers are

placed on the subject that are either reflective, which are also called passive markers, or light-emitting, which are also referred to as active markers. For that, it is effective if the subject wears a full-body unitard made from a stretchy fabric [KW08].
Passive markers, which are fabricated with reflective materials, have either spherical, semi-spherical or circular shapes. Sizes and shapes of these markers vary based on camera resolutions and capture subjects. For instance, markers that are placed on the subject's face are smaller than the ones placed on the subject's torso. Light reflected by passive markers come from light-emitting diodes (LEDs) that are mounted on the cameras [KW08].
Active markers themselves consist of LEDs. There are systems where all LEDs are illuminated simultaneously and there are ones where only selected LEDs are illuminated at the same time, which eliminates the requirement to identity each individual marker at a given time. The modulation of the amplitude or frequency of the LEDs allows them to be identified by the system.
In order to triangulate the 3D position of a marker, at least two cameras are needed. Accuracy of the estimated 3D position is increased by adding more cameras to the optical system [KW08].
The advantages of optical motion capture are

- high optical data accuracy,
- high capture rate,
- the simulations capture of multiple subjects,
- the easily changeable configuration of the marker system depending on the project and
- the possibility for the subjects to move freely in the capture volume.

The disadvantages of optical motion capture are

- expensive data post-processing,
- the high possibility of occlusion of the markers by other subjects or objects in the capture volume,
- the need to control the lighting inside the capture volume and
- the expensive hardware required for these types of systems [KW08].

Figure 2.1 shows two optical motion capture systems being used, where one system uses active markers and the other uses passive markers [Ver21].
Optical motion capture can also be performed without the use of markers. This is referred to as **markerless motion capture** which was enabled by research advancements

Figure 2.1: An optical motion capture system: Upper image shows a performer wearing passive markers and the lower image shows a performer wearing active markers [Ver21].

in the filed of computer vision. The subject's movement is analyzed in multiple video streams by computer vision algorithms to detect the human forms and decompose them into single, isolated parts, which are then used for tracking [Nog12]. An example can be seen in Figure 2.2.

2.1.2 Magnetic Motion Capture

A magnetic motion capture system consists of multiple tracking sensors placed on a subject which are then used to measure their spatial relationship to a magnetic transmitter. Since each sensor can output its translation and orientation, post-processing is reduced compared to optical systems which makes magnetic systems very convenient for real-time applications. Additionally, the problem of occlusion is eliminated which is also an advantage over optical motion capture systems. Similar to optical systems, multiple subjects can be recorded simultaneously with multiple setups. Another big advantage is their lower cost compared to optical systems.
Their big disadvantage, however, is the possible distortion of the output by magnetic and electrical interference caused by electronics or metal objects in the environment [KW08]. Sometimes even the building's own structure can cause these kinds of interferences to the magnetic system[Nog12]. One can divide magnetic systems into two groups.

Figure 2.2: Markerless motion capture was used during filming of the Lord of the Rings trilogy. The movements analyzed in the upper image are used to generate the CGI character Gollum in the lower image which mimics the movements performed by the actor [Nog12].

The first group relies on direct current (DC) electromagnetic fields, which are highly sensitive to copper and aluminium, and the second group utilized alternating current (AC) electromagnetic fields, which are sensitive to steel and iron [KW08].
Additionally, the batteries, which need to be recharged every few hours, and the wiring needed to power the tracking sensors can be limiting to the movement of the capture subject. Also, the sampling rate is lower than what is possible with optical systems and the magnetic data can suffer from noise. Magnetic systems also lack the easiness of changing the configuration of the tracking sensors and their capture volume is generally smaller than optical systems [KW08].
Figure 2.3 shows an example of a magnetic motion capture suit [BRRP97].

2.1.3 Mechanical Motion Capture

A mechanical motion capture system measures the joint angles of a subject wearing an exo-skeleton, which is an articulated device consisting of straight rods linked to potentiometers at the joints of the body. Additional types of mechanical systems are digital armatures and data gloves [KW08].
Mechanical motion capture systems can be used in real-time applications, are low cost, easily transportable, do not suffer from occlusion or magnetic and electrical interferences. Also, if the mechanical system is wireless, it can provide a large capture volume [KW08].
Another advantage theses types of systems have, are the similarity of the interface to stop motion systems which are commonly used in the film industry. This similarity allows for an easy transition between the technologies [Nog12].
A big issue of mechanical motion capture systems is their inability in measuring accurate global translation, which they measure using accelerometers. For instance, if the subject jumps, the recorded data will not follow the subject and instead stay on the floor. Often

Figure 2.3: A magnetic motion capture suit. [BRRP97].

times, magnetic sensors are added to increase the accuracy of the system. Another disadvantage is the restriction of the subjects movement by the exo-skeleton, which can also break if the subject performs movements that can damage the device, like rolling on the floor. Additionally, mechanical systems suffer from low sampling rate and fixed sensor configuration [KW08].
Figure 2.4 shows an example of a mechanical motion capture suit [Rah18].

Figure 2.4: A mechanical motion capture suit (exo-skeleton). [Rah18].

2.1.4 Inertial Motion Capture

An inertial motion capture system aims to estimate the orientation of body segments using skin-mounted IMUs which contain inertial sensors. The estimation accuracy using these sensors has improved significantly with recent advancements in the development of IMUs

and signal processing techniques [VP20]. The process of calculating the position and orientation from inertial sensor measurements is also referred to as *dead-reckoning* [KHS17]. There are significant advantages of inertial motion capture systems compared to other systems such as optical motion capture. They are significantly cheaper, more versatile and portable. Optical motion capture is often confined to a singular environment, whereas an inertial system can be used in a variety of environment such as outdoors, in the laboratory, in the workplace, in clinics or even at home. Additionally, they are not that complicated or time consuming to set up compared to other motion capture systems explained above and they also do not suffer from expensive post-processing, which makes them ideal for real-time applications [VP20].
However, the biggest disadvantage of the sensors used in an inertial system are their drifting errors, which often times can only be reduced by adding additional sensors [VP20]. Since this work is about my attempt at developing an inertial motion capture system, I will talk more deeply about their properties and components in the following sections and chapters.

2.2 Inertial Sensors

The combination of a three-axis gyroscope and a three-axis accelerometer is also referred to as an inertial sensor. An IMU is a device that contains such an inertial sensor [KHS17]. Nowadays, with the developments of Microelectromechanical systems (MEMS), IMU devices became much smaller, cheaper and have a reduced power consumption, which made them more widely used in applications such as robotics, smartphone navigation or Augmented and Virtual Reality [TN18].
MEMS are a combination of electrical and mechanical components in the micrometer scale. They are created by a combination of semiconductor and micro-fabrication technologies, where micro machine processing techniques are used to integrate the mechanical structures, electronics and sensors into a common silicon substrate [Dad14].

2.2.1 Accelerometer and Gyroscope

Whereas the accelerometer sensor is used for measuring acceleration, the gyroscope is used for measuring angular velocity as can be seen in figure 2.5 [Dad14].

An accelerometer makes use of Newton's second law $F = ma$, which states that the acceleration a of a body is in the same direction as and directly proportional to the force F acting on a body and inversely proportional to the body's mass m [Dad14]. Using this principle, the accelerometer sensor can be seen as a mass suspended by springs. When the mass is being displaced due to some movement, that displacement is measured using a displacement pickoff which provides a signal proportional to the force acting on the mass in the direction of the input axis. Figure 2.6 illustrates this principle [Woo07].

A gyroscope takes advantage of the Coriolis effect. It explains that a body rotating in a frame of reference at angular velocity ω and a mass moving with velocity v, has a force F that can be calculated by $F_c = -2m(\omega \times v)$. In order to measure the Coriolis

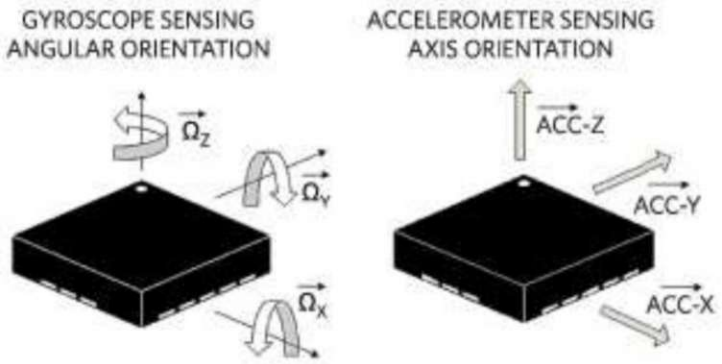


Figure 2.5: Right: Gyroscope measuring angular velocity. Left: Accelerometer measuring acceleration along its sensitive axis. [Dad14].

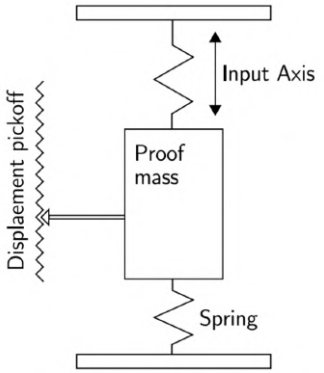


Figure 2.6: Shows the spring arm system of an accelerometer [Woo07].

effect MEMS gyroscopes contain vibrating elements. The simplest vibrating element geometry contains a single mass that can vibrate along a certain drive axis as can be seen in figure 2.7. Whenever the gyroscope rotates, a secondary vibration is triggered along the perpendicular sense axis because of the Coriolis force. By measuring this secondary vibration the angular velocity can by derived [Woo07].

2.2.2 Coordinate frames

It is also important to define the different coordinate frames when talking about the measurements outputted by gyroscopes and accelerometers [KHS17]:

- **Inertial frame**: Linear acceleration and angular velocity measured by the IMU are with respect to this stationary coordinate frame, where the origin is positioned at the center of the earth.
- **Body frame**: All inertial measurements are resolved in this frame, which is the coordinate frame of the IMU device and is aligned with its chassis. The origin

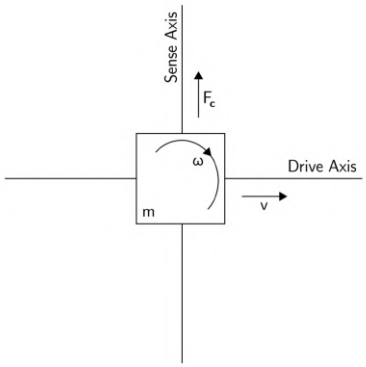


Figure 2.7: A gyroscope containing a vibrating single mass [Woo07].

coincides with the center of the accelerometer triad.

- **Navigation frame**: This coordinate frame is the local geographical frame where we want to navigate. In order to do that, we need to express the position and orientation of the body frame with respect to the navigation frame as seen in figure 2.8 [KHS17]. In our application the navigation frame is stationary, since it does not need to be moved or rotated.

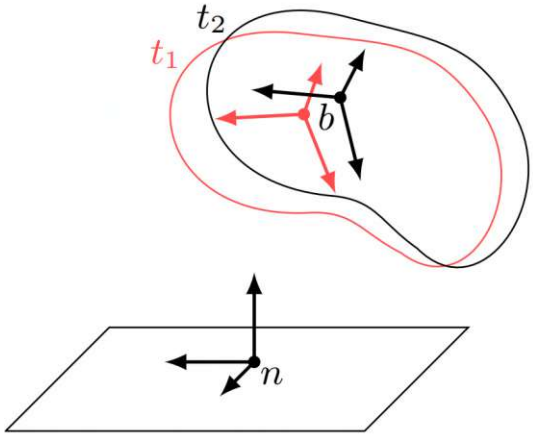


Figure 2.8: Expressing position and orientation of the moving body frame with respect to the navigation frame. t_1 and t_2 represent two different time steps. This is also referred to as pose estimation. [KHS17].

2.3 Sensor Fusion

When readings from multiple sensors are available that would jointly provide more information on the measured system than a sole sensor would provide by itself, a computational methodology called sensor fusion can be applied to combine the measurements from these

multiple sensors. Sensor fusion algorithms find their use in multiple applications such as navigation and localization in robotics, understanding traffic scenes in autonomous cars or analyzing biosignals in biomedical engineering [HS20].
The main objective of any sensor fusion algorithm is taking measurements from multiple sensors and estimating one or more quantities of interest. One can summarize by saying that a sensor fusion system consists of three main components (as can be seen in Figure 2.9) [HS20]:

1. **Multiple senors** for measuring observable quantities
2. **One or multiple models** for relating these measured quantities to quantities of interest
3. **Estimation algorithm** which combines these models and the measured quantities and estimates the quantities of interest [HS20]

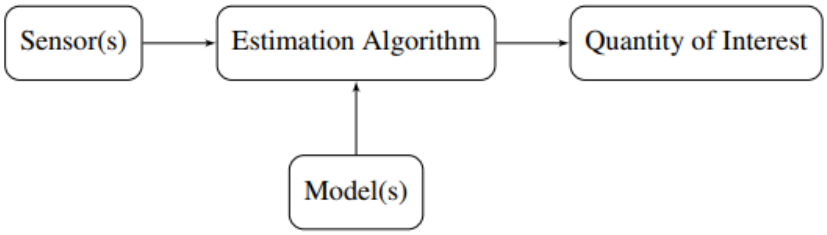


Figure 2.9: Shows the three main components of a sensor fusion system [HS20].

In our case, the multiple sensors are the accelerometer and gyroscope and the quantity of interest, which we want to estimate, is the orientation of the IMU device.
In the following sections, I will explain two sensor fusion algorithms, the complementary filter and the more advanced Kalman filter, both of which are used in this project.

2.3.1 Complementary Filter

Given accelerometer and gyroscope measurements, the complementary filter tries to estimate the attitude at time t based on measurements $y_{1:t}$. Since both gyroscope and accelerometer can provide information on orientation of the sensor, the complementary filter can provide an estimate based on the advantages and disadvantages that are known about both of these sensors [KHS17].
The orientation calculated from accelerometer measurements are noisy and suffer from vibrations but have a higher accuracy and can therefore be trusted over long periods of time. Orientations calculated from the gyroscope, on the other hand, suffer from drift but are accurate and can be trusted on a short time scale.
The complementary filter exploits these properties when providing an orientation estimate. When looking at it in the frequency domain, the gyroscope measurements have desirable

properties at high frequencies and thus a high-pass filter can be applied on those measurements. On the other hand, the accelerometer measurements have desirable qualities at low frequencies and therefore a low-pass filter can be applied on these measurements. The sum of the high-pass filter and the low-pass filter must equal to one, which is also the rationality behind the name complementary filter [KHS17].
This concept can be illustrated in the one-dimensional case, where we have an angle θ_{acc} calculated from accelerometer measurements and an angle θ_{gyro} calculated from gyroscope measurements. The estimated angle using the complementary filter $\hat{\theta}$ can be calculated using the following formula

$$\hat{\theta}_t = (1 - \gamma)\ \theta_{acc,t} + \gamma\ \theta_{gyro,t}. \tag{2.1}$$

γ is the only parameter that needs to be chosen beforehand. Choosing a large γ would mean that the angle estimated by the gyroscope contributes more to the overall estimated angle than the accelerometer, which in our case is the desired effect. Choosing a γ closer to 1 would correspond to applying a high-pass filter to the gyroscope measurements and a low-pass filter to the accelerometer measurements [KHS17].

2.3.2 Kalman Filter

The Kalman filter, which was published in the 1960s by Rudolf Emil Kalman and has since revolutionized the field of estimation, can be seen as a recursive predictive filter that relies on the use of state space techniques and recursive algorithms [Kle04].
The Kalman filter can estimate the state of a dynamic system, that can suffer from noise which is often assumed to be white noise. The filter then improves the estimated state by incorporating measurements related to that state. These measurements, however, also suffer from disturbances [Kle04].
The filter is composed of two core steps, which are

1. the *prediction* step
2. and the *correction* step.

The first step predicts the dynamic model while the second step corrects it using the observation model, where the goal is to minimize the error covariance of the estimator. This is an iterative process where each time step the state of the previous time step is used as an initial value. Figure 2.10 shows the prediction and correction step of the Kalman fitler [Kle04].
The state vector, dynamic model and observation model are the basic components of the Kalman filter [Kle04]:

- **The state vector** describes the state of the dynamic system and simultaneously represents its degrees of freedom. It contains the variables of interest, which are not measured directly but inferred from the measurements. The state vector takes

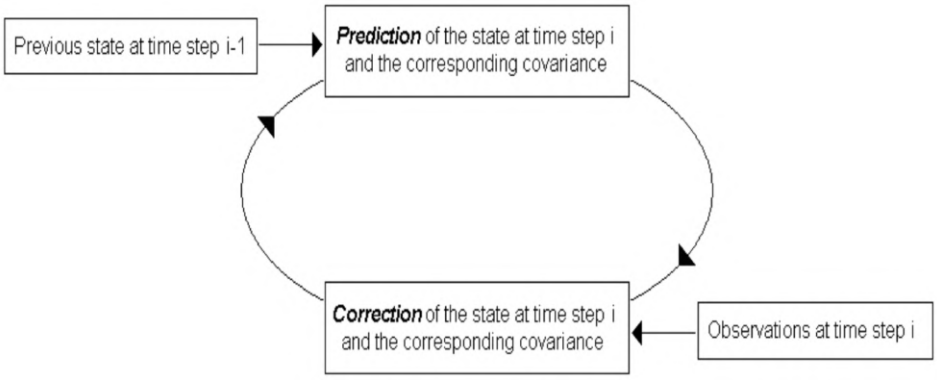


Figure 2.10: Shows the core building blocks of the Kalman filter [Kle04].

two values. The first value is the a priori value which is the predicted value. The second value is the a posteriori value, which is the corrected value [Kle04].

- **The dynamic model** can be seen as a description of the transformation of the state vector over time and it is usually a system of differential equations, which can be written as

$$x'(t) = \frac{d}{dt}x(t) = f(x(t), m(t)) \tag{2.2}$$

or in the case of a linear system as

$$x'(t) = \frac{d}{dt}x(t) = A \cdot x(t) + w(t) \text{ [Kle04].} \tag{2.3}$$

$x(t)$ is the state vector and A is the dynamic matrix. $w(t)$ is the process noise which is the assumed white noise with a normal distribution [WB01] that has the covariance matrix $Q(t)$. In a linear system A is constant [Kle04].

- **The observation model** describes the relationship between the measurements and the state. When the system is linear, the measurements can be represented by a system of linear equations that depend on the state variables. This can be written as

$$z(t) = h(x(t), v(t)) = H \cdot x(t) + v(t), \tag{2.4}$$

where H is the observation matrix, $z(t)$ is the vector containing the observations at time t and $v(t)$ is the measurement noise that has the covariance matrix $R(t)$ and also has a normal distribution [WB01]. Similar to the dynamic matrix, H is constant in a linear system [Kle04].

The Kalman Filter Algorithm

Here, I want to give a broad overview of the discrete Kalman filter algorithm.

Prediction Step: This step consists of the time update equations which are responsible

for projecting the current state and error covariance forward in time in order to get the a priori estimates for the subsequent time step. The two time update equations are

$$\begin{aligned} \hat{x}_k^- &= A\hat{x}_{k-1} + Bu_k \\ P_k^- &= AP_{k-1}A^T + Q \end{aligned} \tag{2.5}$$

where k is the time step, A is the state transition matrix, $\hat{x}_k^-$ is the a priori state estimate, Q is the process noise covariance, P_k^- is the a priori estimate error covariance and P_k is the a posteriori estimate error covariance. u_k is the optional control input and the matrix B relates this control input to the state vector [WB01].

Correction Step: This step consists of the measurement update equations which are responsible for using new measurements to get improved a posteriori estimates. The three time measurement update equations can be written as

$$\begin{aligned} K_k &= P_k^- H^T (HP_k^- H^T + R)^{-1} \\ \hat{x}_k &= \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \\ P_k &= (I - K_k H)P_k^- \end{aligned} \tag{2.6}$$

where K_k is the Kalman gain, $\hat{x}_k$ is the a posteriori state estimate at step k, R is the measurement noise covariance, H is the observation matrix, z_k is the measurement at step k and I is an identity matrix [WB01].
$(z_k - H\hat{x}_k^-)$ is also refered to as the measurment innovation or residual and it measures the difference between the predicted measurement $H\hat{x}_k^-$ and the actual measurement z_k. An innovation of zero means total agreement between the two values. K is also refered to as the blending factor [WB01] and signifies how large the correction will be [Roj03].

The Extended Kalman Filter Algorithm

The Kalman filter algorithm explained above is only suitable for linear systems. In order to make the algorithm work for non-linear systems, it has to be modified. The Kalman filter used to solve non-linear problems is called the Extended Kalman filter [Kle04].
In case of a linear system, the dynamic matrix A and the observation matrix H are constant and can therefore be pre-computed. This reduces the amount of time consuming calculations and thus makes it more efficient than having a non-linear system, where the dynamic matrix and observation matrix are functions of the state and therefore are subjected to change during every time step [Kle04].
The algorithm for the Extended Kalman filter an be described as follows:

Prediction Step: The two time update equations are

$$\begin{aligned} \hat{x}_k^- &= f(\hat{x}_{k-1}, u_k, 0) \\ P_k^- &= A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T. \end{aligned} \tag{2.7}$$

Correction Step: The three time measurement update equations can be written as

$$\begin{aligned} K_k &= P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \\ \hat{x}_k &= \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0)) \\ P_k &= (I - K_k H_k) P_k^- . \end{aligned} \tag{2.8}$$

There are few differences to note here between this Extended Kalman filter algorithm and the one explained previously.
The jacobians A, W, H and V have the subscript k attached to them to indicate that they change at each time step k and have to be recomputed. W is a matrix that represents the jacobian of partial derivatives of f with respect to w and the V matrix represents the jacobian of partial derivatives of h with respect to v [WB01].
Also notable is that, for the Kalman gain equation, H_k only magnifies the relevant components of the measurements. This means that if a one-to-one mapping between the state and the measurement does not exist, the Kalman gain is affected by H_k so that it only magnifies parts of the residual $z_k - h(\hat{x}_k^-, 0)$ that would affect the state. If a one-to-one mapping does not exist over all the measurements then the algorithm will diverge, making the process unobservable [WB01].
Figure 2.11 shows a high-level diagram explaining the steps with the equations needed for the Extended Kalman filter algorithm [WB01].

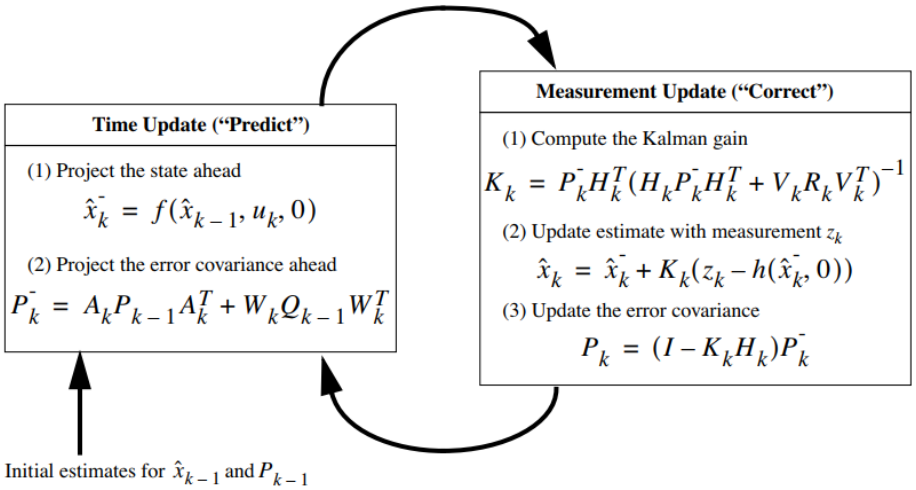


Figure 2.11: Shows the complete Extended Kalman Filter algorithm [WB01].

2.4 Angular Velocity Integration

The integration of angular velocity is a common application in different fields such as aerospace navigation, computer graphic or robotics. For example, in the case of an inertial navigation system, as the one described here in this thesis, the angular velocity measured

from the gyroscope embedded in an IMU is used to approximate the global orientation of this device. In order to do so, the coordinate transformation matrix between the body-fixed coordinate system and the spatial coordinate system has to be calculated by integrating angular velocity-rotation differential equations in real time [ZS11].
Since the angular velocities measured from the gyroscope are not constant in time, the differential equations to be solved are non-linear making an analytical solution impossible. Therefore, numerical methods have to be applied to solve these non-linear differential equations [ZS11].
There are different ways of integrating angular velocity and I am going to explain two numerical methods in the following sections. These are the Euler method and the Runge-Kutta method.

2.4.1 Euler Method

In order to approximate the solution of a differential equation, Euler's method computes the slope of a tangent line between two points. The tangent in this case represents the derivative of a function at a certain point and can therefore be used to approximate the subsequent point. The recursive formula for Euler's method can be written as

$$\begin{aligned} \dot{y}_n &= f(x_n, y_n) \\ y_{n+1} &= y_n + h\dot{y}_n, \quad for\ n = 0, 1, 2... \text{ [Mol18]} \end{aligned} \tag{2.9}$$

In our case, y_n represent the rotation angle at a certain time step t. $\dot{y}_n$ is the derivative of the rotation angle at that time step, which is the angular velocity. y_{n+1} is rotation angle at the subsequent time step, which we are trying to approximate. h is the step size and in our example corresponds to the time difference between the two time steps. Considering these conditions, the angular velocity integration using Euler's method can be rewritten as

$$\phi(t + \Delta t) = \phi(t) + \omega(t)\Delta t, \tag{2.10}$$

where ϕ is the rotation angle and ω is the angular velocity.

2.4.2 Runge-Kutta Method

Euler's method is generally not recommended when integrating angular velocity as it lacks the accuracy and stability provided by more advanced numerical methods. One of these methods is the Runge-Kutta method [PTFV92].
There are different versions of the Runge-Kutta method. The first-order Runge-Kutta method corresponds to Euler's method explained above, while the more accurate second-order Runge-Kutta method, corresponds to the midpoint method, where the slope at the midpoint of the interval h is considered in the evaluation. However, in this work, the fourth-order Runge-Kutta method, which is more accurate than the second-order method, is used for solving these non-linear differential equations. The formula for this

method can be written as

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \text{ [PTFV92].} \end{aligned} \tag{2.11}$$

Similar to Euler's method, when integrating the angular velocity using the fourth-order Runge-Kutta method, the formula can be rewritten as

$$\begin{aligned} k_1 &= \omega(t, \phi(t))\Delta t \\ k_2 &= \omega(t + \frac{\Delta t}{2}, \phi(t) + \frac{k_1}{2})\Delta t \\ k_3 &= \omega(t + \frac{\Delta t}{2}, \phi(t) + \frac{k_2}{2})\Delta t \\ k_4 &= \omega(t + \Delta t, \phi(t) + k_3)\Delta t \\ \phi(t + \Delta t) &= \phi(t) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}. \end{aligned} \tag{2.12}$$

Figure 2.12 illustrates a geometric interpretation of the fourth-order Runge-Kutta method.

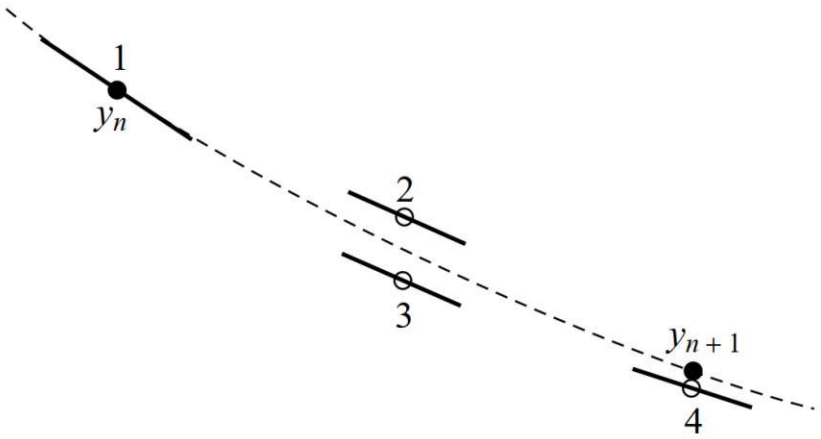


Figure 2.12: For the fourth-order Runge-Kutta method, four different evaluations are necessary. The first evaluation is at the initial point, the second and third evaluations are at the midpoint of the interval and lastly the fourth evaluation is at the endpoint. The filled circles are the final function values, while the non-filled circles represent the function values used for calculating the final values, before being discarded [PTFV92].

2.5 Orientation representations

In this section, I will describe ways to represent the attitude of a rigid body using mathematical constructs such as Euler angles and quaternions.

2.5.1 Euler Angles

Using a set of three Euler angles (also refered to as roll, pitch and yaw angles) is the most common method to describe the attitude of a rigid body in three-dimensional space. They are easy to understand and have thus become very popular. Unfortunately, they suffer from disadvantages that make them hard to work with:

- Certain essential Euler angles functions contain singularities.
- The integration of incremental changes in attitude over time is less precise using Euler angles compared to quaternions [D+06].

Euler angle rotations are calculated by applying three consecutive coordinate rotations, which are rotations about a single coordinate axis. The rotation matrices for the x-,y- and z-axes coordinate rotations with $R_i : \mathbb{R} \to SO(3), i \in [x, y, z]$ are

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{2.13}$$

$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \tag{2.14}$$

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.15}$$

The Euler angle vector representing the attitude of a rigid body is defined by

$$u := [\phi, \theta, \psi]^T, \tag{2.16}$$

where the coordinate rotation about the x-axis is by angle ϕ, the coordinate rotation about the y-axis is by angle θ and the coordinate rotation about the z-axis is by angle ψ. In order to map the Euler angle vector to its corresponding rotation matrix, one can use the function, $R_{xyz} : R^3 \to SO(3)$,

$$R_{xyz}(\phi, \theta, \psi) := R_z(\psi) R_y(\theta) R_x(\phi) \text{ [D+06]}. \tag{2.17}$$

Following regular matrix multiplication rules, R_{xyz} can be defined as

$$R_{xyz}(\phi,\theta,\psi) = \begin{bmatrix} \cos(\theta)\cos(\psi) & -\cos(\theta)\sin(\psi) & -\sin(\theta) \\ \cos(\phi)\sin(\psi)+\sin(\phi)\sin(\theta)\cos(\psi) & \cos(\phi)\cos(\psi)-\sin(\phi)\sin(\theta)\sin(\psi) & -\sin(\phi)\cos(\theta) \\ \sin(\phi)\sin(\psi)-\cos(\phi)\sin(\theta)\cos(\psi) & \sin(\phi)\cos(\psi)+\cos(\phi)\sin(\theta)\sin(\psi) & -\cos(\phi)\cos(\theta) \end{bmatrix} \tag{2.18}$$

.

Out of 12 possible rotation sequences [Hen77], I assume in this case that the rotation sequence is (x,y,z), where the rigid body is first rotated about the x-axis (roll), then about the y-axis (pitch) and finally about the z-axis (yaw). When used in this sequence these Euler angles are also referred to as Cardan angles or Tait-Bryan angles and they are often used in computer graphics and aerospace engineering [D+06].
A major drawback of Euler angle representations are singularities that emerge from the so called gimbal lock. Gimbal lock is triggered when the second Euler angle (in this case the pitch angle) reaches a critical value, e.g. 90 degrees, causing changes in the first and third Euler angle to be indistinguishable. Quaternions are immune to such singularities and are therefore the preferred method for representing attitude in three-dimensional space [D+06].

2.5.2 Quaternions

Mathematician William R. Hamilton introduced in 1843 the concept of quaternions, which are a generalization of complex numbers, as a way of describing rotations in three-dimensional space [BA12].

Advantages and disadvantages of quaternions over Euler angles

Compared to Euler angles, important functions of quaternions do not suffer from singularities and are convenient for applying angular velocity integration over time since it is less computationally expensive. However, quaternions must have unit length, i.e. unit quaternions, in order to be considered a pure rotation. Unlike Euler angles, which have three independent parameters, they are not easy to understand, since the four parameters needed to describe a quaternion are depended on each other and lack intuitive physical meanings [D+06].

Definition

A quaternion consists of a scalar part, which represents the real value, and a vector part, which represents the three imaginary values. The sum of the scalar part q_0 and the vector part $\vec{q} = (q_1, q_2, q_3)$ describe a quaternion q [Gra08]. This can be written as

$$q = q_0 + \vec{q} = q_0 + q_1 i + q_2 j + q_3 k, \tag{2.19}$$

where i, j and k satisfy the following conditions [BA12]:

$$\begin{aligned} i^2 + j^2 + k^2 &= -1 \\ ij = k, ji &= -k \\ jk = i, kj &= -i \\ ki = j, ik &= -j \end{aligned} \tag{2.20}$$

Addition and Multiplication

Adding two quaternions together is simply a component wise addition of the four elements of each quaternion. For quaternions q and p, this can be written as

$$p + q = (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k \text{ [Jia22].} \tag{2.21}$$

Quaternion multiplication is not commutative, as can be shown by 2.20, but satisfies the associativity rule [Boy17]. The product of quaternions q and p can be written as

$$\begin{aligned} q \cdot p = (q_0 + q_1 + q_2 + q_3) \cdot (p_0 + p_1 + p_2 + p_3) = \\ p_0q_0 - q_1p_1 - q_2p_2 - q_3p_3 \\ + (q_1p_0 + q_0p_1 + q_2p_3 - q_3p_2)i \\ + (q_2p_0 + q_0p_2 + q_3p_1 - q_1p_3)j \\ + (q_3p_0 + q_0p_3 + q_1p_2 - q_2p_1)k \end{aligned} \tag{2.22}$$

or more concisely as

$$q \circ p = (p_0q_0 - \vec{p} \cdot \vec{q}, q_0\vec{p} + p_0\vec{q} + \vec{q} \times \vec{p}) \text{ [Gra08].} \tag{2.23}$$

Norm, conjugate and inverse of a quaternion

The norm of a quaternion is defined as

$$|q| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}. \tag{2.24}$$

A quaternion that has a unit length is called a unit quaternion, which is a possible representation of a rigid body attitude.
The conjugate of a quaternion q is

$$q^* = q_0 - q_1i - q_2j - q_3k \tag{2.25}$$

and its inverse is

$$q^{-1} = \frac{q^*}{|q|^2} \text{ [BA12].} \tag{2.26}$$

As can be seen by the equation above, for unit quaternions, its inverse equals its conjugate. However, due to the possibility of numerical errors, this rule could be violated [Boy17].

Vector rotation

In order to rotate a vector, which lives in $\mathbb{R}^3$, using a quaternion, which lives in $\mathbb{R}^4$, one has to transform a vector into a pure quaternion, which is defined as a quaternion with the real part equal to zero.

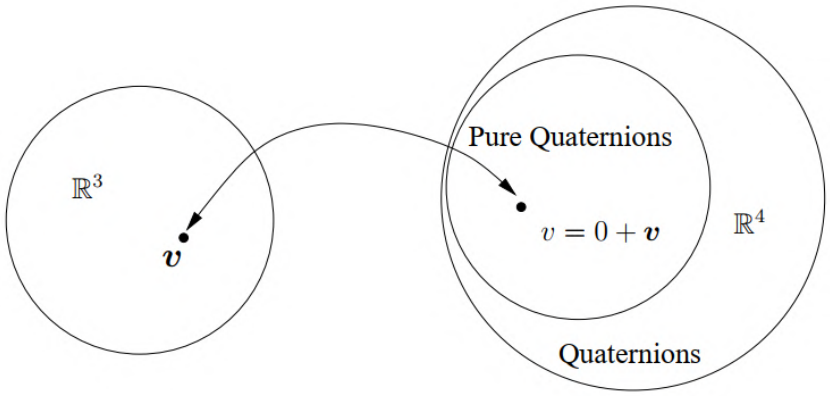


Figure 2.13: Mapping a three-dimensional vector to a pure quaternion [Jia22].

The rotation of a vector, mapped to a pure quaternion, from coordinate frame A to coordinate frame B using a unit quaternion q_R can be performed using the conjugation operation, which can be written as

$$v_B = q_R \circ v_A \circ q_R^*. \tag{2.27}$$

v_A represents the pure quaternion in coordinate frame A and v_B represents the pure quaternion in coordinate frame B after the rotation [BA12]. A unit quaternion can be converted into a three-dimensional rotation matrix, making converting a vector into a pure quaternion unnecessary:

$$\begin{aligned} &\vec{v_B} = R(q)\vec{v_A}, \ \textit{where} \\ &R(q) = \begin{bmatrix} 2(q_0^2 + q_1^2) - 1 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & 2(q_0^2 + q_2^2) - 1 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & 2(q_0^2 + q_3^2) - 1 \end{bmatrix} \text{ [Kui99].} \end{aligned} \tag{2.28}$$

Differentiation and Integration

Here, I will consider the quaternions to be unit quaternions, since their differentiation formula becomes different and more complicated when their magnitude is not one [Jia22] and because unit quaternions are the ones I will be using throughout my work.
I will consider a quaternion q to be a function of time t. $q(t)$ describes the relative change of a moving object's orientation in its body frame with respect to its world frame. Furthermore, $w(t)$ describes the angular velocity of the body frame relative to the world frame [Jia22]. The derivative of q(t) can be written as

$$\dot{q} = \frac{1}{2}\Omega(w)q. \tag{2.29}$$

$\Omega(w)$ is the real matrix representation of the angular velocity vector:

$$\Omega(w) = \begin{bmatrix} 0 & -w_1 & -w_2 & -w_3 \\ w_1 & 0 & w_3 & -w_2 \\ w_2 & -w_3 & 0 & w_1 \\ w_3 & w_2 & -w_1 & 0 \end{bmatrix} \text{ [AC13].} \tag{2.30}$$

For quaternion integration, I will consider Euler's method as an approximation of a quaternion at a certain time step. With k being the time step and h the step size, a quaternion integration using Euler's method can be written as

$$q_{k+1} = q_k + \frac{1}{2} h\Omega(w_k)q_k \text{ [Jia22].} \tag{2.31}$$

2.6 Levenberg–Marquardt algorithm

The problem of fitting a parameterized mathematical model to a collection of data values by minimizing an objective, which is represented as a sum of squares and errors of the model function relative to the collection of data values, is called a least squares problem. If the least squares objective's parameters are quadratic, the model is linear in its parameters. Is the model non-linear in its parameters, then the least squares problem is solved by utilizing iterative solution algorithms, which minimize the sum of the squares of the errors of the model function with respect to the data values. Usually, a sequence of carefully selected updates to values of the model parameters are utilized. The Levenberg-Marquand algorithm is such an algorithm that was developed to solve non-linear least squares problems [Gav19].
A function to be minimized by this algorithm has the form:

$$f(x) = \frac{1}{2} \sum_{j=1}^{m} r_j^2(x). \tag{2.32}$$

$x = (x_1, x_2, ..., x_n)$ represents a vector and r_j are called residuals. Each r_j is a function from $\mathbb{R}^n$ to $\mathbb{R}$. Furthermore, there is the assumption that $m \geq n$ [Ran04].
The Levenberg-Marquand algorithm can be considered as a combination of the gradient decent method and the Gauss-Newton method, both of which are numerical minimization algorithms [Gav19].

The Gradient Descent Method

The gradient decent method is considered a general minimization algorithm, where at each step the parameter values are updated by adding the negative of the scaled gradient. This can be written as

$$x_{i+1} = x_i - \lambda \nabla f \text{ [Ran04].} \tag{2.33}$$

Problems with simple objective functions are good applications for the gradient descent method, where good convergence is possible [Gav19].

The Gauss-Newton Method

With the Gauss-Newton method there is the assumption that the objective function near the optimal solution is approximately quadratic. This method, compared to the gradient descent method, is well-suited for moderately-sized problems where it generally has a much faster convergence [Gav19]. The update rule can be written as

$$x_{i+1} = x_i - \frac{\nabla f(x_i)}{\nabla^2 f(x_i)} \text{ [Ran04].} \tag{2.34}$$

The Levenberg–Marquardt Method

The Levenberg-Marquant algorithms uses the two methods explained above, in order to combine the advantages provided by the Gauss-Newton method and gradient descent method. Thus, the update rule is a combination of the two methods and can be written as

$$x_{i+1} = x_i - \frac{\nabla f(x_i)}{H + \lambda \operatorname{diag}[H]} \text{ [Ran04].} \tag{2.35}$$

In the equation above, H represents the Hessian matrix evaluated at x_i. The Levenberg-Marquant method behaves similar to the Gauss-Newton method if the parameters are near the optimal value, and it behaves similar to the gradient descent method if the parameters are far from the optimal value [Gav19].

The steps of the Levenberg–Marquardt algorithm can be simply described as

1. Update using the equation 2.35.
2. Calculate the error using the new parameter vector.
3. Has the error increased, then the last step is reversed and λ is increased by a significant factor. After that, go back to (1).
4. Has the error decreased, then the update step is accepted and λ is decreased by a significant factor.
5. Go back to (1) [Ran04].

CHAPTER 3

Related Work

In this chapter, I will describe existing solutions for inertial motion capture. In the first section, I will present solutions that are already on the market. After that, I will summarize several state-of-the-art research papers describing the usage of IMUs for extracting motion data.

3.1 Commercially Available Inertial Motion Capture Solutions

I this section I will list and describe three commercially available inertial motion capture systems: Xsens MVN, Sony mocopi and Perception Neuron.

3.1.1 Xsens MVN

Xsens MVN is presented as a full-body human motion capture system that is cheap and easy to use. It consists of 17 inertial and magnetic motion trackers that capture the subject's motion. That motion data is then transmitted via a wireless connection to a PC which is then processed using advanced algorithms and biomechanical models. Different versions of the Xsens motion capture engine are offered which target different customer and market needs. There is MVN Animate for the animation market and MVN Analyze for the human motion measurement market. In addition to the different motion capture engines, two different hardware systems are offered: MVN Awinda and MVN Link [SGB+18].

Hardware

Each of the 17 motion trackers contain both inertial and magnetic sensors which are 3D gyroscopes, 3D accelerometers and 3D magnetometers. Each tracker offers high sampling

rates that could exceed 1 kHz and transmits the data at a lower rate, e.g. 60 Hz. The data outputted by theses sensors are run through an advanced signal processing pipeline. The 17 tackers are placed on the user's head, sternum, shoulders, upper arms, fore arms, hands, pelvis, upper legs, lower legs and feet [SGB+18].
Next, I want to explain the differences between MVN Link and MVN Awinda.

- **MVN Link**:
  MVN Link is the body-wired solution where the motion trackers are wired and connected to an on-body data hub. This data hub is not only responsible for powering the trackers but also for gathering the data and sending the data to a computer via Wi-Fi. The data hub also makes it possible to record the motion data and store it in its internal memory without transmitting the data to the computer in real time. The MVN Link system has a battery life of 10 hours during normal use [SGB+18].
  The provided custom lycra suit allows for simplified mounting of the motion trackers on specific body locations [SGB+18].

- **MVN Awinda**:
  MVN Awinda is a completely wireless solution where the motion trackers are attached to the body using straps. Each motion tracker has its own battery and transceiver for wirelessly transmitting the data. The battery life is 6 hours and thus lower than the battery life of MVN Link [SGB+18].

Figure 3.1 shows MVN Link and MVN Awinda [SGB+18].

Software

As previously mentioned, the Xsens software engine has customized versions for the 3D Character Animation market (MVN Animate), which includes games and movies, and for the Human Motion Measurement market (MVN Analyze), which includes research, sports and ergonomics [SGB+18].
The user can choose between two processing modes that are offered by the engines. In the first mode, the data outputted by all motion trackers is combined with advanced biomechanical models in real time. The second mode adds an additional feature where the data is processed over a larger time window for achieving more consistent and optimal estimations of position and orientation of each body segment [SGB+18].

Motion tracking

The biomechanical model is a key component in the Xsens engine for estimating the final position and orientation of a tracked body part. The model consists of 23 segments which are the head, neck, shoulders, upper arms, lower arms, hands, pelvis, L5, L3, T12, T8, upper legs, lower legs, feet and toes. However, L5, L3, T12, neck and toes do not

Figure 3.1: Xsens MVN Awinda on the left and Xsens MVN Link on the right [SGB+18].

have inertial or magnetic sensors attached and their movements are estimated using a combination of the biomechanical model and the information from the connected tracked segments [SGB+18].
Additionally, the software engines provide different ways for handling interactions with the environment, specifically how to deal with different floor levels caused by inclines, stairs or soft floor [SGB+18]. Figure 3.2 shows the main components needed for the Xsens software engine to perform motion capture [SGB+18].

Calibration

Before performing motion capture using the Xsens MVN, a calibration process is required to estimate the size and proportions of the tracked subject and the orientation of the sensors with respect to the body parts they are attached to [SGB+18].
Adjusting the scale can be performed by several input parameters provided by the user. Body height and foot length are the minimum parameters required to find the most optimal scale. Providing additional input parameters can help the engine better estimate the scale of the virtual counterpart of the tracked subject.
To find the alignment between the body parts and their attached sensors, the tracked subject has to stand still in either the N-pose or the T-pose (see Figure 3.3) and then

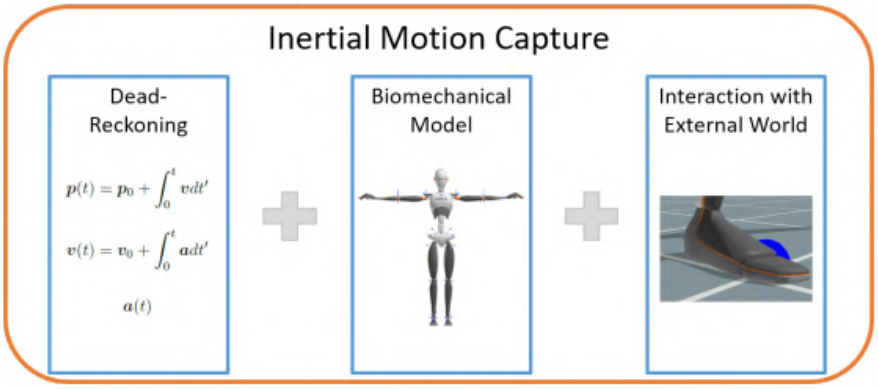


Figure 3.2: An overview of how inertial motion capture is performed by the Xsens MVN [SGB+18].

walk a few meters back and forth for a short time period [SGB+18].
In order to find the origin and the forward X-direction of the local coordinate system,

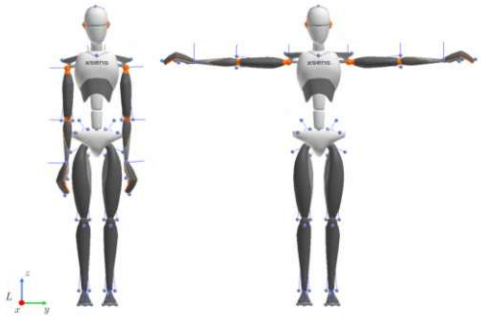


Figure 3.3: The N-pose on the left and the T-pose on the right. [SGB+18].

the last calibration step requires the subject to stand in either the N-pose or T-pose while facing towards the forward direction of the measurement environment [SGB+18].

3.1.2 Sony mocopi

Sony mocopi is a 3D inertial motion capture solution that consists of six lightweight sensors (see Figure 3.4). The manufacturer calls this solution *Mobile Motion Capture*, because this technology allows the user to record his or her own movements and transfer that information to digital characters all while wearing everyday clothes in various indoor and outdoor locations without the need for expensive and time-consuming hardware setups [moc23a]. A compatible smartphone is enough to use this motion capture system [moc23b].

Figure 3.4: The six light-weight sensors where each one is assigned to a specific body segment [moc23b].

Hardware

Each sensor module consists of a 3D accelerometer and a 3D gyroscope. A magnetometer sensor is not present in this tracking solution. Each sensor weighs 8 grams and has a diameter of 3.2 cm [moc23b].
Each sensor, which has a battery life of up to 10 hours, is paired with a compatible smartphone using Bluetooth. After successful pairing, the mocopi smartphone application can then transfer the motion data to a computer via Wi-Fi and the UDP protocol. The motion data is transmitted at a rate of 50 Hz [moc23b].

Software

The smartphone application support two modes, which are either saving videos of the motion data applied to an avatar in a chosen backdrop or saving the recorded raw motion data on the smartphone device. Additionally, it makes it possible to stream the motion data using Wi-Fi to a computer with a running Unity or Unreal Engine application [moc23b].
Since the data processed from the accelerometers and gyroscopes can drift over time, the provided software allows the user to reset the origin and orientation of the targeted avatar [moc23b].

Motion tracking

From the six sensors attached to head, waist, hands and legs, the 3D position and posture of every joint angle is estimated. This process is performed in two steps (see Figure 3.5):

1. **Estimation of the joint positions where the sensors are attached:** First the data from the accelerometers and gyroscopes are captured. Using integral calculation, the 3D position of the joints can be estimated which, however, are prone to errors that accumulate over time. Using an AI model, these errors can be corrected to get better estimations of each joint position [moc23a].

2. **Estimation of the intermediate joint positions where the sensors are not attached**: The posture and position of an intermediate joint, that connects two IMU-tracked body segments, cannot be uniquely calculated using simple geometric operations because of the complicated structure of the human body and the high degree of freedom of its joints. Therefore, an interpolation of the joint positions is performed assisted by an AI model which was trained on various human movements [moc23a].

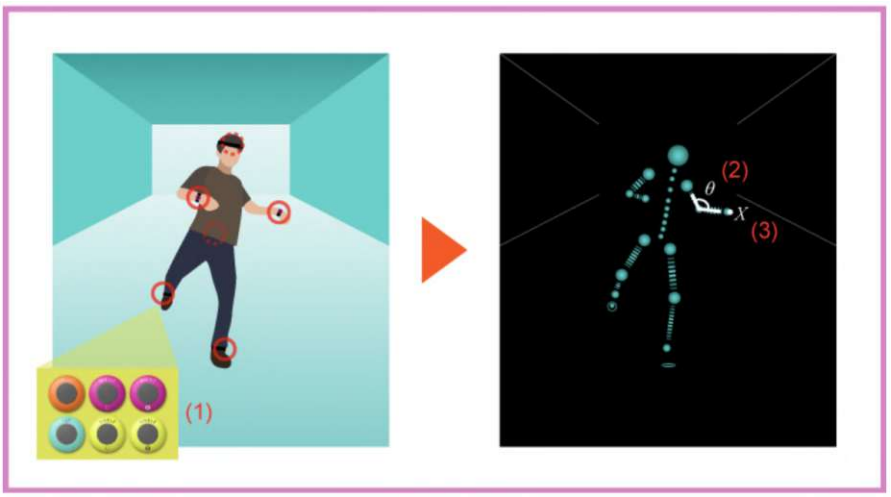


Figure 3.5: The data from the accelerometers and gyroscopes (1) are used to calculate joint posture (2) and joint position (3) [moc23b].

Calibration

Calibrating the sensors is done in four consecutive steps:

1. Selecting the height of the tracked subject in the smartphone application.

2. Standing still in the N-pose.

3. Walking one step forward.

4. Standing still in the N-pose [moc23b].

3.1.3 Perception Neuron

Perception Neuron is another inertial motion capture solution which is offered in two different variants: Perception Neuron Studio and Perception Neuron 3.

Hardware

Perception Neuron Studio is considered to be the manufacturer's flagship solution which is more advanced in both accuracy and performance. The battery life of each of the included 18 sensor modules is up to 10 hours and the system supports frame rates up to 240 FPS with a wireless range of 7 meters. Each sensor module has an embedded 9 degrees of freedom (DoF) IMU sensor. This solution can also be paired with the Perception Neuron Studio Gloves, which offer five-finger tracking using six IMUs strategically placed on each glove [pns23a].
Perception Neuron 3 is a less expensive and more portable solution with a battery life of up to 5 hours for all 18 sensors, 60 FPS sampling frequency and a 4 meter wireless range. All motion data is transferred to the computer using a provided USB Transceiver to ensure seamless communication. This solution can be paired with three-finger capture gloves. By only tracking the user's thumb, index and middle fingers using three separate IMUs, the orientation and position of the remaining fingers are approximated [pns23b]. Figure 3.6 shows the Perception Neuron 3 inertial sensor and Figure 3.7 shows all 18 of those trackers attached to a capture subject.

Figure 3.6: The Perception Neuron 3 body sensor with the dimensions 27.9 x 16.2 x 11.6 mm and 4.1 grams weight. [pns23b].

Software

With each of the hardware solution explained above, the capture and recording software solution Axis Studio is also provided. It not only supports real-time capture and recording but also streaming into third party 3D applications. Features of this software include

Figure 3.7: All 18 sensors attached to a capture subject using body straps [pns23c].

advanced data processing, for cleaning up recorded motion capture data before export, and a *magnetic field sweeper*, which visualizes magnetic field data in the environment to assist the user in finding the most optimal location for performing inertial motion capture with minimal magnetic interference [pns23d].

Calibration

The manufacturer describes four calibration steps necessary to optimally estimate the capture subject's posture [SFM+19]:

1. Sitting in a chair while remaining as still as possible.
2. Standing up, placing hands down on the side of the thighs while keeping feet parallel.
3. Standing still in a T-pose.
4. Bending knees to approximately 45 degrees and placing arms forward while palms face the floor [SFM+19].

3.2 Other Solutions for Extracting Motion Data using IMUs

I found several state-of-the-art research papers describing the usage of IMUs to extract motion data.

3.2.1 Inertial motion capture for gait analysis

[CGD+19] developed a modular inertial motion cature system using low-cost IMUs that is able to estimate body segment orientation, posture angle trends as well as performing gait recognition for working activities in industrial environments. Each sensor module consists of an IMU (MPU-6050), that is composed of a 3D accelerometer and a 3D gyroscope, and a 3D magnetometer sensor (HMC5883L). For their microcontroller they used a Raspberry Pi. The sensors are attached to the following body segments: pelvis, trunk, left/right arm, left/right forearm, left/right upper leg and left/right lower leg. The orientation extracted from each sensor module is parameterized using quaternions in order to avoid singularities. In order to achieve reliable attitude estimation, they performed sensor fusion using an Extended Kalman Filter algorithm, which takes the accelerometer, gyroscope and magnetometer readings as input. For each body segment the Kalman Filter outputs the Tait-Bryan Euler angles (roll, pitch and yaw). Figure 3.8 shows the motion tracking system in an upper-body configuration [CGD+19].

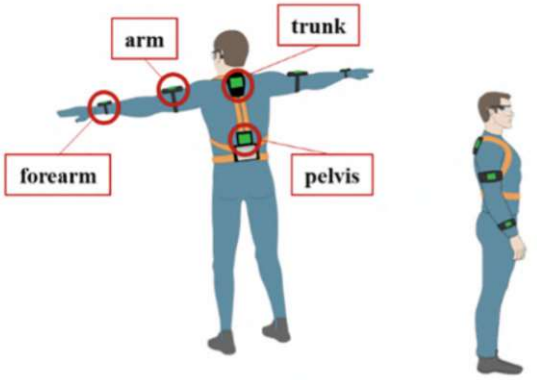


Figure 3.8: The body motion tracking system as proposed by [CGD+19].

3.2.2 Inertial motion capture for underwater gait analysis

[MFPC+21] studied underwater gait kinematics for patients in rehabilitation using an underwater inertial measurement system. They conducted a study where the goal was to evaluate the proposed underwater IMU system for calculating the subject's knee angles during gait in both land and underwater settings while comparing the results to an optical tracking solution, which they consider to be the ground truth.
The wearable inertial trackers are placed on the right upper and lower leg of the capture subject. For the IMU sensor the BNO055 was selected which has a 3D accelerometer, 3D gyroscope and also includes a 3D magnetometer. Figure 3.9 shows the basic hardware components of each of these waterproof trackers [MFPC+21].
The proposed calibration procedure for each subject consists of first standing in an upright position, then lifting the right leg up in hip flexion for at least 5 seconds. Finally, the subject walks along a straight line. They concluded that their proposed system

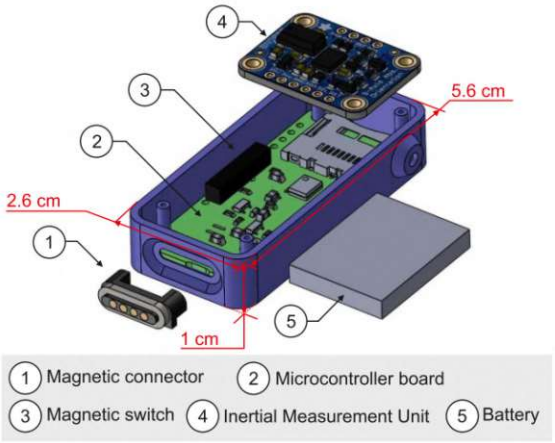


Figure 3.9: The waterproof motion trackers worn by the subjects [MFPC+21].

provides reliable and repeatable results when calculating the knee angles in an underwater setting [MFPC+21].

3.2.3 Inertial motion capture for teleoperation

[ZPW+22] suggest a solution for robot teleoperation using an IMU-based wearable motion-capture suit. Their project focuses not only on providing methods for remote controlling a robot in a more intuitive way but also providing the user with the capability of triggering more complex movements, which would be otherwise difficult to perform using e.g a joystick.
Their hardware for the robot consists of a quadruped robot, a modified 5-IMU robotic arm and RGB camera, which is responsible for providing the user with visual feedback via a VR headset, thus enabling telepresence. For their motion capture suit, they used the commercially available solution Perception Neuron. In addition to the 16 IMU sensors that are attached to the different body segments of the captured subject, the user also wears Studio Motion Capture gloves for accurate finger tracking. The HTC Vive Pro was used as a VR system. Figure 3.10 shows the teleoperation system developed by [ZPW+22].

3.2.4 Angular kinematic analysis in sports using IMUs

In the research paper published by [DGVRM+21], the level of agreement for angular velocity between an IMU gyroscope and a 3D optical motion capture system are evaluated in the context of playing tennis. They state that optical capture methods are considered to be the *gold standard* for kinematic evaluation due to their high accuracy. However, they describe the limitations that optical motion sensors have, including mandatory line of sight, specific lighting conditions as well as movement restrictions and discomfort of the user caused by the high number of markers placed on the capture subject. They

Figure 3.10: The user controlling the robot while wearing the inertial motion capture suit and a VR headset [ZPW+22].

further emphasize the importance of a high sampling rate for achieving good results whether the motion is recorded using optical or inertial sensors [DGVRM+21].
For their gyroscope sensors, they used the IDG-650 (for x- and y-axes) and ISZ-650 (for z-axes) and they recorded the angular velocity at a frequency of 128 Hz. The tracker were placed on the participant's body segments using 2 cm wide elastic belts and they ensured that they did not cause any discomfort that would affect the participant's movements while playing tennis. For optical tracking they placed passive marker on the IMU trackers and the cameras recorded the subject's movement also with a sampling frequency of 128 Hz. Each participant wore four trackers on the head, trunk, arm and forearm as can be seen in Figure 3.11 [DGVRM+21].
After conducting the experiments described in the paper, the authors concluded that the IMU gyroscope data and the data extracted from optical sensors show almost perfect linear relationship and concordance in both relative and absolute values [DGVRM+21].

3.2.5 Inertial motion capture with an improved Extended Kalman Filter algorithm

[LLW+22] propose a full-body inertial motion capture system with an improved sensor fusion method to not only estimate orientation but position of the capture subject in the virtual environment. For sensor fusion, they use the Extended Kalman Filter algorithm in combination with the foot-mounted zero-velocity-update (ZUPT) technique, which is used to estimate the position. Similar to [DGVRM+21], they compare their results to an optical motion capture system.
Their trackers consist of the IMU sensor MPU-9280, which has a built-in 3D accelerometer, 3D gyroscope and 3D magnetometer, the microprocessor STM32F103C8T6, a Wi-Fi module and a 450-mAh lithium polymer rechargeable battery. The entire hardware system can be seen in Figure 3.12. An internet router provides a Wi-Fi hotspot for both

Figure 3.11: A participant wearing four IMU trackers with reflective markers attached to them [DGVRM+21].

the sensor modules and the computer which allows for wireless data transmission between computer and each sensor module via the UDP protocol. This proposed system has a sampling frequency of 150 Hz [LLW+22].

[LLW+22] propose a calibration procedure for the inertial sensors. For the accelerometer

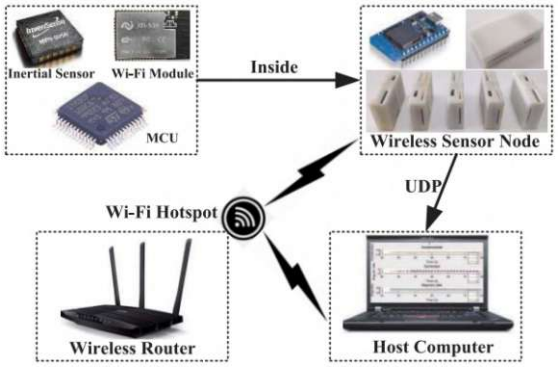


Figure 3.12: The basic hardware setup for the inertial motion capture system as proposed by [LLW+22].

calibration, they rely on the fact that when the sensor is in a static state, the sensor readings are only affected by gravity and thus in the absence of any errors the condition $g = \sqrt{a_x^2 + a_y^2 + a_z^2}$ must be satisfied in any direction. This can also be visualized as a sphere which has the radius g. However, in the real world, accelerometer sensors always suffer from orthogonal or bias errors between the axes which distort the output of the sensor and therefore make the sphere appear more like an ellipsoid. [LLW+22] suggest using the ellipsoid fitting method for reducing these errors. To collect the data for the

calibration, the sensor is placed in multiple orientations as can be seen in Figure 3.13. For the gyroscope calibration the sensor is also placed in a stationary position, in order to retrieve the bias error for each axis, which is considered to be the main error source. Ideally, when placed in a static state the angular velocities read from the gyroscope should have a normal distribution with zero mean [LLW+22].

For sensor fusion, [LLW+22] proposed the ZUPT-Aided Extended Kalman Filter, which

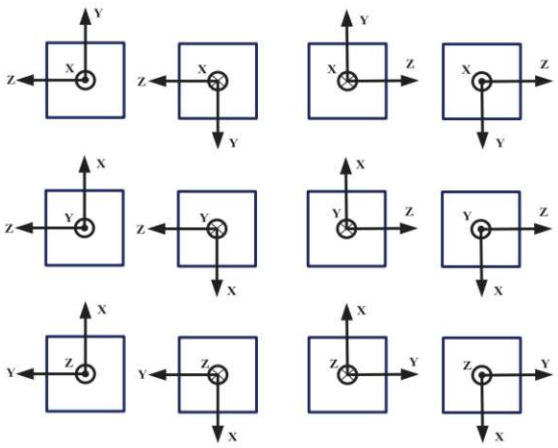


Figure 3.13: For the accelerometer calibration the IMU is placed in 12 different orientations [LLW+22].

uses the ZUPT method to update the state estimation whenever the foot is on the floor. The ZUPT method takes advantage of the fact that velocity should be exactly zero when the foot is on the floor and uses the difference between the velocity output and the value zero as the observation for the Kalman Filter.

After analyzing their results, [LLW+22] claim that their proposed method outperforms existing attitude estimation algorithms for motion tracking using inertial sensors and that their solution is feasible for 3D human motion capture.

CHAPTER 4

Methodology

In this chapter, I will describe the methodological approach for implementing my proposed inertial motion capture system. First, I will give a general overview of the implemented system and then describe my design for its hardware and software components.

4.1 System Overview

My motion capture system consists of 11 motion trackers, which are attached to the user's head, left upper arm, left lower arm, right upper arm, right lower arm, chest, tailbone, left upper leg, left lower leg, right upper leg and right lower leg using body straps. It is a completely wireless solution, where each motion tracker is powered by its own rechargeable battery and sends the motion data via Wi-Fi using UDP to a receiving computer at 60Hz. The trackers are also partially modular, which allows the user to swap the battery or IMU in case of hardware damage.
I also introduce a calibration procedure to my system, which uses a self-build robot which I will refer to as *Mini-Robot* and a calibration algorithm which I implemented to retrieve the most optimal calibration parameters for each IMU. The *Mini-Robot* has an arm to which the IMU can be attached and then rotated in different orientations.
Each motion tracker is configurable using a configuration software I developed. The configuration software allows the user to change the targeted body part as well as the kind of data that is sent from the motion tracker. The user can program the tracker to send either raw IMU data or the already estimated orientation in a quaternion or Euler angle format. Additionally, the user can send the calibration parameters of the IMU using this application and then choose whether to send calibrated IMU data or not. Another important configuration which is possible by using this software, is setting the UDP port, IP address of the targeted computer as well was the name and password of the Wi-Fi network.
The receiving computer has an application running with a fully rigged 3D humanoid

character that receives the motion data from all trackers and retargets that motion data to the specified joints of the humanoid 3D character.
I have implemented a sensor fusion algorithm for calculating the tracker's orientation in 3D space. The algorithm runs on both the motion tracker (if it is not configured to send the raw IMU data) and the motion capture application on the receiving computer.
Each motion tracker is also designed to work for purposes other than full-body motion capture, for example, objects the user wishes to track in the real world. By configuring the motion tracker to send orientation data in quaternion or Euler angle format, the user only needs to retrieve the precomputed orientation using my developed sensor fusion algorithm in a custom application. When configured to sending raw data, the IMU output is sampled and send at a rate of 60Hz. When configured to sending orientation data, the IMU output is sampled at 1000Hz and sent at a rate of 60Hz.
An overview of my described system is further explained and can be seen in Figure 4.1.

4.2 Hardware Design

In this section, I will describe my design for the two hardware devices which I developed for this project: The motion tracker and the *Mini-Robot*

4.2.1 Motion Tracker

My intent with this work, is to develop a low-cost alternative to available inertial motion captures solutions. Therefore, I started with assembling and designing a motion tracker which consists of the following four primary components:

- An IMU sensor for detecting the movements of the tracker using acceleration and angular velocity
- A microcontroller which reads the IMU data and output the sensor's orientation in three-dimensional space
- A Wi-Fi module for transmitting the data to a receiving computer
- A rechargeable battery that powers the motion tracker

My final design for the motion tracker can be seen in Figure 4.2 and here is a description of the hardware components that I used for each individual tracker:

- **The ESP32** low-cost SoC (System on Chip) microcontroller, which was developed by the Espressif System company, is considered a successor to the ESP8266. It includes an integrated Wi-Fi 802.11 b/g/n, dual mode Bluetooth, I2C interface and 4MB flash memory. In addition to the over 30 available GPIO (General-Purpose Input/Output) pins, it comes with two CPU cores adjustable to up to 240 MHz [BFS19]. The microcontroller's operating voltage is between 2.3 to

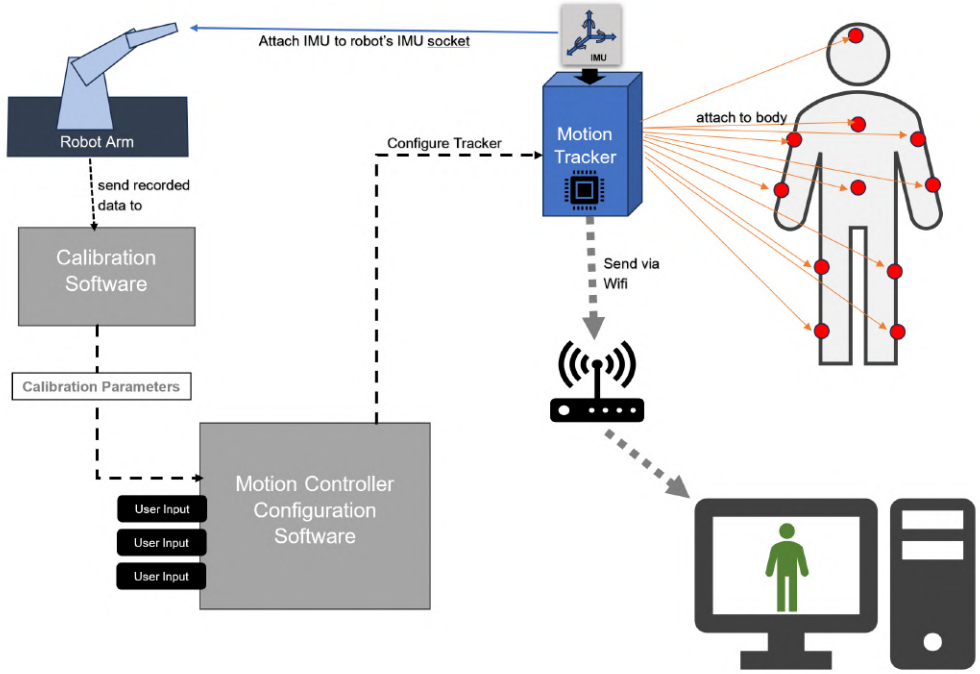


Figure 4.1: A high-level overview of my proposed inertial motion capture system: The IMU can be detached from the motion tracker and attached to the robot arm in order to perform the calibration. After the calibration parameters are computed, the IMU can be reattached to the motion tracker. The tracker can then be configured by the configuration software by first connecting the tracker to the computer. The user's selected configuration and the calibration parameters are sent to the motion tracker using the configuration software. The configured 11 motion trackers are then attached to the capture subject's body segments which are marked by the red circles. The motion data is then sent from the trackers to the receiving computer via Wi-Fi and then processed and targeted to the rigged 3D character.

3.6V [Esp23]. Using the Micro USB port of the ESP32, I can establish a serial connection to the computer.

- **The MPU-6050** is a 6 DoF IMU device which contains a 3D gyroscope and a 3D accelerometer. With the three onboard 16-bit analog-to-digital converters (ADCs) the gyroscope and accelerometer outputs are digitized. With the user-programmable full-screen ranges, precision tracking of both fast and slow motions is possible. Full-screen ranges available for the gyroscope are $\pm 250, \pm 500, \pm 1000 and \pm 2000^o/sec$ and the full-screen ranges available for the accelerometer are $\pm 2g, \pm 4g, \pm 8g and \pm 16g$. Communication with this device can be performed using I2C.
- **A 3.7V 1100mAh rechargeable lithium battery** with a cut-off voltage of 4.2V [Spe].

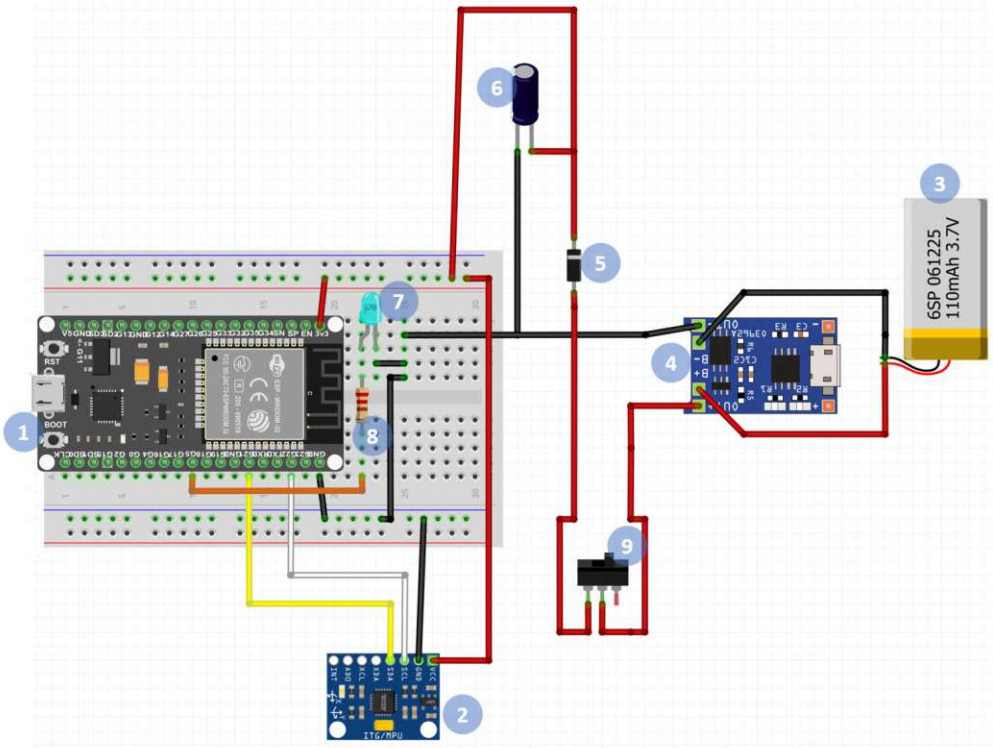


Figure 4.2: The circuit diagram for my proposed motion tracker: **(1)** ESP32, **(2)** MPU-6050, **(3)** 3.7V 1100mAh lithium battery, **(4)** TP4056, **(5)** 20V/1A Schottky diode, **(6)** 100μf electrolytic capacitor, **(7)** LED, **(8)** 220Ω resistor.

- **The TP4056 Type-C USB module** is a constant-current and constant-voltage linear charger for single cell lithium-ion batteries with a fixed 4.2 charge voltage [Nan]. Using the TP4056, the lithium battery can be recharged via USB-C.
- **A 20V/1A Schottky diode** for low forward voltage drops [Tai]. The Schottky diode serves to protect the ESP32 from the high voltage of an overcharged lithium battery. When overcharged the output voltage of the battery used in my system becomes close to 4.2V which exceeds the maximum operating voltage of the ESP32 which is 3.6V.
- **A 100μf electrolytic capacitor** which is used for energy storage, filtering and output voltage regulation [TJS23].
- **A 3-pin toggle switch** serving as an on/off switch for the microcontroller.
- A **220Ω resistor** to limit the current flowing through the **light-emitting diode (LED)**. The LED serves as a signal to the user, that the motion tracker is turned on. Also, when dim, it signals the user that the battery needs to be recharged.

I have encompassed all these components in my self-designed chassis which I have 3D printed. In order to assemble these components as compactly as possible, I designed a PCB. PCBs are non-conductive substrate plates which are utilized in the assembly and interconnection between electronic components. These interconnections are facilitated through routes of a conductive material recorded on the substrate [SSM19].
I would like to further note that this proposed hardware design can be easily modified, for instance, by choosing a different microcontroller or a different IMU sensor. The MPU-6050 has a dedicated digital motion processor (also referred to as DMP) which can fuse the accelerometer and gyroscope data and output orientation data as quaternions. However, this is very specific to the MPU-6050 and possibly not transferable to other IMU sensors. My introduced sensor fusion method should be compatible across different commercially available inertial sensors.

4.2.2 Mini-Robot for Calibration

This *Mini-Robot* is responsible for automatically placing an IMU in several random orientations, which are recorded and then used during my calibration process. Using three different servo motors connected by a serf-designed 3D printed rig, this *Mini-Robot* is capable for rotating around the roll, pitch and yaw axes.
For each servo motor, I use the **SG90 9g micro servo motor**, which is a light-weight motor with a maximum rotation of 180^o [lux]. All servo motors are controlled by an **ESP32 microcontroller**. On the top-most surface of the robot's rig (arm), which should be parallel to the ground when in idle position, I placed a socket for easy attachment and detachment of the IMU. Unlike the motion tracker, this robot is not powered by a battery but through the ESP32's Micro USB port.
My final design for the *Mini-Robot* can be seen in Figure 4.3

4.3 Software Design

In this section, I will describe the design of my developed software responsible for the IMU calibration, attitude estimation from the IMU data and applying that information to a 3D humanoid virtual character.

4.3.1 Calibration

Ideally, an IMU's tri-axial accelerometer and gyroscope share the same 3D orthogonal sensitivity axes with a scaling factor that converts the digital measurements by each sensor into real physical quantities. However, due to sensor axis misalignments, cross-axis sensitivities, inaccurate scaling and non-zero biases of low-cost MEMS based IMUs, as the ones used in this project, these measured quantities are distorted. These errors could be the result of inaccuracies in the assembly of these products. Therefore, a calibration procedure is needed to identify the values of these distortions [TPM14].
My approach for calibrating the IMU sensors is based on the work by [TPM14]. I will give a summary of their work and then describe my approach and how it differs.

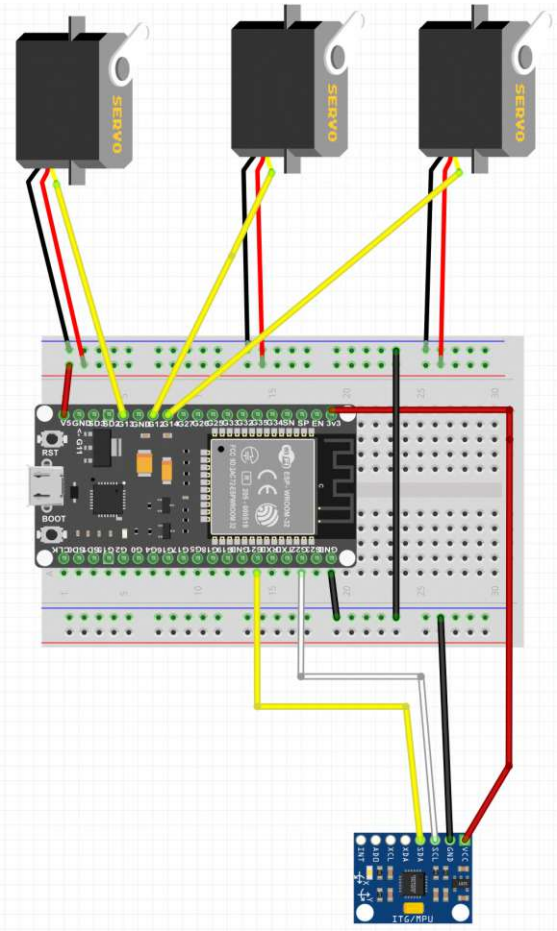


Figure 4.3: The circuit diagram for the *Mini-Robot*.

[TPM14] suggest that during the data collection phase, the IMU should first be put in a static position with no motion for a certain time period and then moved into different positions to generate unique temporarily stable rotations. This collected data is subsequently used to estimate the sensor errors. They do note, however, that cross-axis sensitivity errors are not addressed in their work since they are not easily distinguishable from minor misalignment errors. [TPM14] use the idea, that when an accelerometer is put in a static state, the magnitude of the outputted acceleration vector equals the magnitude of the gravity added by noise, biases and misalignment errors. By applying a minimization algorithm over the set of static positions, these errors can be estimated and thereby calibrate the accelerometer. The accuracy of the accelerometer relies on precisely identifying the static positions in the collected data. This is done with a variance-based static detector where the magnitude of the variance of the accelerometer values in a specific time interval is checked against a predefined threshold.
Using the gravity vector positions measured by the calibrated accelerometer as a reference, the gyroscope can be calibrated. By applying angular velocity integration using the fourth-

order Runge-Kutta method between two consecutive static attitudes, the gravity vector position of the new orientation can be estimated. By minimizing the errors between the estimated gravity vectors and the ones given by the accelerometer as reference, the gyroscope can be successfully calibrated. However, the accuracy of the gyroscope calibration is strongly reliant on the preciseness of the accelerometer calibration [TPM14]. The key difference in my approach is the absence of a static detector as all the static positions are clearly identified by my self-build *Mini-Robot*. Additionally, I have omitted the accelerometer bias error parameters as they have caused problems in my attitude estimation approach.

Data Collection

As previously mentioned, performing data collection using the *Mini-Robot* adds the advantage of unmistakably labeling the periods in which the IMU is in motion or in a static position. This should increase the accuracy of the calibration output, since the approach described in [TPM14] relies on statistical methods. Additionally, the IMU sensors are securely placed in the robot's IMU socket which should prevent the sensor from slipping or tilting when moved in a static position, which is a possibility when the sensor is instead moved by hand as described in [TPM14]. Also, [TPM14] recommend rotating the IMU in 35 to 50 static positions which is a tedious process when manually done by hand. The *Mini-Robot*, however, adds a level of automation to that process. The downside of using the robot for data collection is the need for an additional device to perform the calibration task.
Figure 4.4 shows a high-level diagram of the data collection process using the *Mini-Robot*. First the robot arm is reset to its idle position. While the raw IMU data is read during every frame, the data collection process starts by calculating the gyro bias after the robot leaves its idle state after $T_{gyroinit}$ seconds. After that, the robot arm with its attached IMU is set in a random orientation, while the IMU data recorded during that event is marked as *data collected while in motion*. After $T_{motionwait}$ seconds, the robot arm should be in a static position free from any jittering caused by the fast movements of the robot arm. The IMU data recorded during $T_{staticwait}$ is marked as *data collected while in a static position*. As suggested by [TPM14], the accelerometer data collected during the static states should be averaged to filter out possible noise. Therefore, after the robot arm enters the static state, the program waits $T_{staticwait}$ seconds before it repositions the robot arm in the next iteration. During this entire process the collected data is recorded by an external software and saved in a file which will then be fed to the actual calibration algorithm.

Accelerometer calibration

As explained above, the goal of this calibration is to convert the non-orthogonal accelerometer frame into an orthogonal one while also estimating the scaling errors. Similar to [TPM14], I will refer to the non-orthogonal accelerometer frame as AF and the orthogonal accelerometer frame as AOF. They also define the condition that the AOF's x-axis must

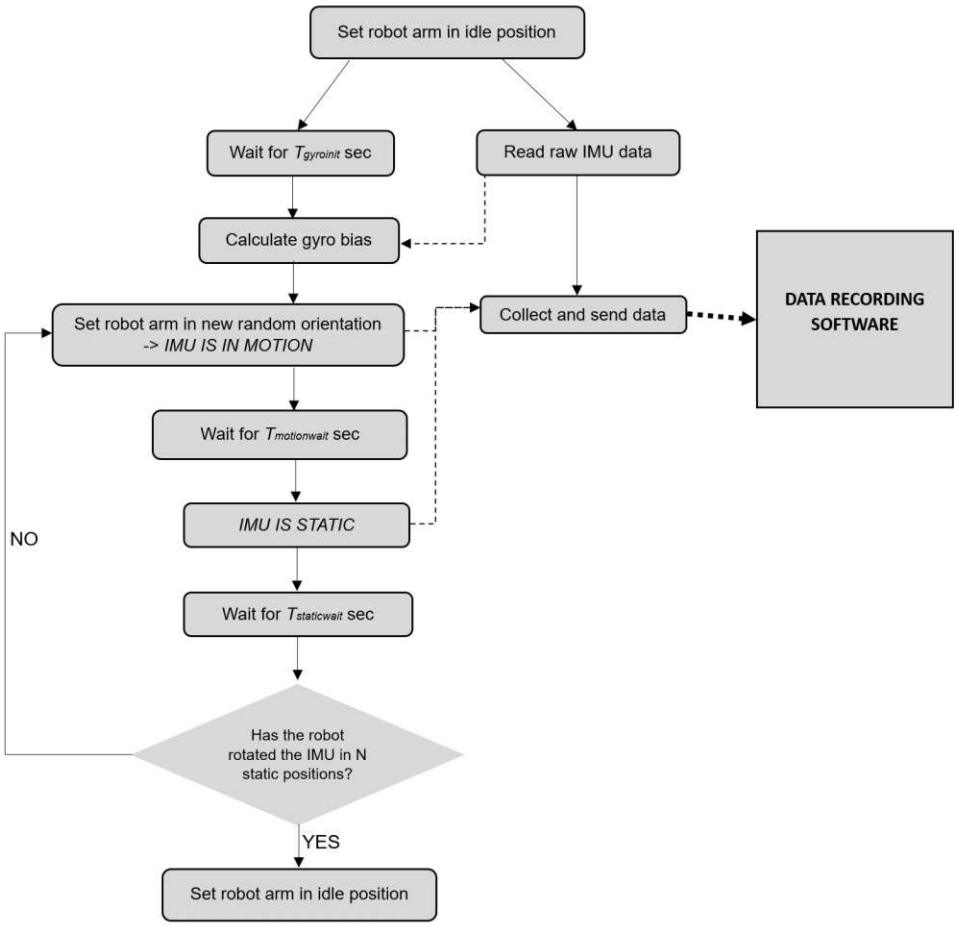


Figure 4.4: Data collection as performed by the *Mini Robot*.

coincide with the AF's x-axis and that the AOF's y-axis must lie in the plan spanned by the AF's x and y axis. Furthermore, the body frame (BF) is defined which in our case corresponds to the coordinate frame of the IMU chassis and is also orthogonal. Generally, there is no connection between the body frame and the accelerometer and gyroscope frames but they might deviate by small angles. By following the assumption that the AOF coincides with the BF, the AF can be transformed into the AOF using the equation

$$a^O = T^a a^S, \qquad T^\alpha = \begin{bmatrix} 1 & -\alpha_{yz} & \alpha_{zy} \\ 0 & 1 & -\alpha_{zx} \\ 0 & 0 & 1 \end{bmatrix} \text{ [TPM14].} \tag{4.1}$$

a^S is the skewed acceleration vector (in the AF) outputted by the accelerometer which is transformed into the orthogonal acceleration vector a^O (in the AOF) using the rotation matrix T^α, where α_{ij} refers to the rotation of the *i-th* AF axis around the *j-th* AOF or BF axis. Because the AOF is equal to the BF, the lower triangular matrix of T^α equals

to zero [TPM14]. Figure 4.5 shows the relationship between a non-orthogonal sensor frame and the body frame.

In addition to the misalignment errors, the scaling errors need to be taken into account

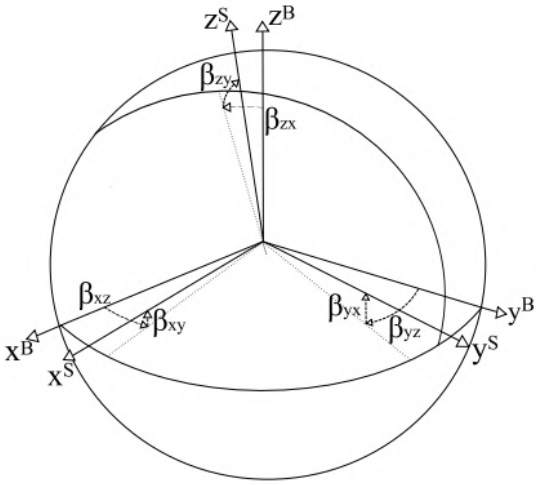


Figure 4.5: Transforming the non-orthogonal sensor frame, defined by the axes x^S, y^S and z^S, into the orthogonal body fame, defined by the axes x^B, y^B and z^B. βij refers to the rotation of the *i-th* non-orthogonal sensor frame axis around the *j-th* BF axis [TPM14].

as well. This can be done by incorporating a scaling matrix K^a into equation 4.1. The final accelerometer sensor error model equation thus becomes

$$a^O = T^a K^a a^S, \qquad K^a = \begin{bmatrix} s_x^a & 0 & 0 \\ 0 & s_y^a & 0 \\ 0 & 0 & s_z^a \end{bmatrix} \text{ [TPM14].} \tag{4.2}$$

Similar to [TPM14], I am also neglecting the measurement noise errors, since during the data collection approach the accelerometer outputs during the static positions of the IMU are gathered over a period of time with the intent of averaging them before using them in the calibration algorithm. However, as previously mentioned, I am neglecting the bias errors, which are incorporated into the accelerometer sensor error model proposed by [TPM14], due to problems that occurred during my implementation when performing attitude estimation.

From equation 4.2, a vector containing the unknown parameters that need to be estimated for the accelerometer calibration can be derived and be written as

$$\theta^{acc} = [\alpha_{yz}, \alpha_{zy}, \alpha_{zx}, s_x^a, s_y^a, s_z^a]. \tag{4.3}$$

Using that parameter vector, 4.2 can be rewritten as the function

$$a^O = f(a^S, \theta^{acc}) \text{ [TPM14].} \tag{4.4}$$

In our case, a^S corresponds to the averaged acceleration vector gathered during a period in which the IMU sensor was in a static state. [TPM14] defines the cost function used to estimate the parameter vector θ^{acc} as

$$L(\theta^{acc}) = \sum_{k=1}^{N}(||g||^2 - ||f(a_k^S, \theta^{acc})||^2)^2 \tag{4.5}$$

where N is the number of static positions assumed by the *Mini-Robot* during data collection, a_k^S is the averaged accelerometer vector taken during the *k-th* static position and g is the gravity vector. As described by [TPM14], I use the cost function 4.5 to estimate and minimize the parameter vector θ^{acc} with the help of the Levenberg-Marquardt algorithm, which is the final goal of the accelerometer calibration.

Gyroscope calibration

Similar to the accelerometer, the non-orthogonal gyroscope frame is referred to as GF and the orthogonal gyroscope frame is referred to as GOF. The condition that the GOF's x-axis must coincide with the GF's x-axis and that the GOF's y-axis must lie in the plan spanned by the GF's x and y axis also applies here. However in this case the GOF does not coincide with the BF.
The sensor error model equation for the gyroscope is very similar to that of the accelerometer and can be written as

$$\omega^O = T^g K^g \omega^S, \quad T^g = \begin{bmatrix} 1 & -\gamma_{yz} & \gamma_{zy} \\ \gamma_{xz} & 1 & -\gamma_{zx} \\ -\gamma_{xy} & \gamma_{yx} & 1 \end{bmatrix} \quad K^g = \begin{bmatrix} s_x^g & 0 & 0 \\ 0 & s_y^g & 0 \\ 0 & 0 & s_z^g \end{bmatrix} \text{ [TPM14]}. \tag{4.6}$$

ω^S is the skewed angular velocity vector (in the GF) retrieved from the gyroscope and is transformed into the orthogonal angular velocity vector ω^O (in the GOF) using the transformation matrix T^g, where γ_{ij} refers to the rotation of the *i-th* GF axis around the *j-th* AOF or BF axis. Because the GOF does not coincide with the BF, the lower triangular matrix of T^g does not equal to zero. Similar to the accelerometer measurements, the scaling errors also need to be taken into account and therefore the scaling matrix K^g is added to the equation. The bias errors can be neglected as they were already calculated during the data collection process. I also neglect measurement noise as described by [TPM14].
From equation 4.6, a vector containing the unknown parameters that need to be estimated for the gyroscope calibration can be derived and written as

$$\theta^{gyro} = [\gamma_{yz}, \gamma_{zy}, \gamma_{zx}, \gamma_{xz}, \gamma_{xy}, \gamma_{yx}, s_x^g, s_y^g, s_z^g]. \tag{4.7}$$

[TPM14] describe the cost function for minimizing the parameter vector θ^{gyro} as

$$L(\theta^{gyro}) = \sum_{k=2}^{N}(||u_{a,k} - u_{g,k}||^2) \tag{4.8}$$

where

$$u_{g,k} = \Psi[\omega_i^S, u_{a,k-1}]. \tag{4.9}$$

Ψ is an operator that takes a series of gyroscope readings ω_i^S and an initial accelerometer vector $u_{a,k-1}$, which corresponds to an accelerometer vector measured and averaged in the *k-1-th* static state and was subsequently corrected using θ^{acc} and equation 4.4, and then computes a final accelerometer vector using the gyroscope readings taken between the *k-1-th* and the *k-th* static state. The cost function 4.8 compares the actual calibrated acceleration vector measured at the *k-th* static state with the one calculated using the gyroscope readings [TPM14]. In my case and like the approach proposed by [TPM14], Ψ is an angular velocity integration algorithm that uses the fourth-order Runge-Kutta method.
In order to rotate $u_{a,k-1}$ into $u_{g,k}$, a rotation matrix must be calculated using the gyroscope readings. I calculate the matrix by first applying angular velocity integration on a quaternion where the scalar part is *one* and the vector part equals to the zero vector. After that I convert the resulting quaternion into a three-dimensional rotation matrix (as described in 2.5.2).
As described by [TPM14] and similarly to the accelerometer calibration, I use the cost function 4.8 to estimate and minimize the parameter vector θ^{gyro} with the help of the Levenberg-Marquardt algorithm. After calculating the estimates, I append the bias vector $[b_x^g, b_y^g, b_z^g]$ to the estimated parameter vector θ^{gyro}. Thus, the final parameter vector outputted by the algorithm becomes

$$\theta^{gyro} = [\gamma_{yz}, \gamma_{zy}, \gamma_{zx}, \gamma_{xz}, \gamma_{xy}, \gamma_{yx}, s_x^g, s_y^g, s_z^g, b_x^g, b_y^g, b_z^g]. \tag{4.10}$$

The results from both the accelerometer and gyroscope calibrations are stored in a separate file via a software I developed, which implements these described algorithms. The file containing the calibration parameters can be used by the configuration software explained in the next section.

4.3.2 Motion tracker configuration software

In this section, I will describe the configuration software I developed for reprogramming the motion tracker according to the user's needs. I designed a user interface for the software to make it easier to use. In order to configure the motion tracker, it has to be connected to the computer, on which the configuration software is running, via USB. The software has the following features:

- Upon launch the active COM ports are detected and if only one is active and a motion tracker is connected to it, all the previously stored configuration data is read from the motion tracker's memory and fed to the configuration software and visualized. That way the user can see the current configuration of the motion tracker.
- If more than one motion tracker is connected, the user can select between them from a dropdown menu.

- In order to connect the motion controller to an available network, the user can enter the Wi-Fi name and associated password in the appropriate input fields and send that information to the motion tracker.
- If the user wants to target a different computer or if the receiving computer's IP address has changed, that new IP address can be also entered. I am also providing the option of detecting the computer's IP address automatically by the software, which can be triggered by clicking on a single button.
- The UDP port, to which the motion data is sent by the motion trackers, can be changed by the user.
- An important feature is the possibility of choosing the targeted body part of the motion tracker via the user interface.
- The user can choose via the configuration software how the data is being sent by the motion tracker. Either the raw IMU data is sent or the orientation data computed by the motion tracker's microcontroller is sent as Euler angles or as a quaternion using my proposed sensor fusion algorithm (described in the following section). In either scenario, the data will be transmitted via Wi-Fi at a rate of 60Hz. However, if the orientation data is transmitted, the sampling rate of the IMU output will be set to 1000Hz. If the raw IMU data is sent, the sampling rate of the IMU output is set to 60Hz.
- The calibration parameters, which were stored in a file by the calibration software, can be sent through the configuration software.
- The user can also choose via the interface, whether the motion tracker should use the calibration parameters or not.

In order to control the integrity of the data strings that are assembled and sent by the configuration software to the motion tracker, I implemented a simple handshake mechanism between the two systems. When the user tries to send the newest configuration to the tracker, a data string containing the new configuration information is assembled and sent via serial communication to the motion tracker. The tracker receives that information parses it and sends the data string back to the configuration software. Once it is received, the configuration software compares the received data string with the data string that was sent out by the software itself. If the string matches, a positive feedback message is displayed in the interface of the software. This handshake mechanism is illustrated in 4.6.

4.3.3 Attitude estimation

In this section, I will explain my approach for estimating the 3D orientation of the motion tracker.

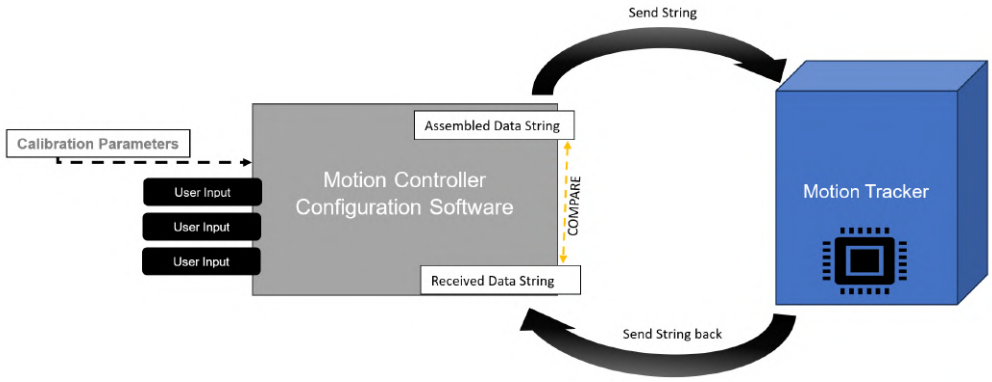


Figure 4.6: My implemented handshake mechanism when there is a communication between the configuration software and the motion tracker.

Data correction and reorientation

The first step is to correct the accelerometer and gyroscope data using the calibration parameters θ^{acc} and θ^{gyro} calculated by the calibration software and passed to the motion tracker through the configuration application. This is done on the microcontroller directly after retrieving the new input from the IMU and using the equations 4.2 and 4.6. Even if the user decides not to send the calibrated values from the microcontroller, the gyroscope bias will always be removed using the gyroscope bias error vector stored as the last three elements of θ^{gyro} since this is the biggest source of error for gyroscope sensors.
The next step is to perform sensor fusion using a quaternion-based Extended Kalman filter, which I have implemented on both the microcontroller and the motion capture application on the receiving computer. However, before processing the IMU data using the Kalman filter, I first reorient the IMU output. The reason for that is the skewed position in which the IMU sensor lies in my assembled motion tracker since, ideally, I want the IMU to lie horizontal to the ground of the motion tracker chassis. Currently, the IMU is strongly rotated around its y-axis in order to fit inside the motion tracker chassis. By employing the rotating matrix for the y-axis $R_y(\alpha)$, as given by 2.14, I reorient both the accelerometer input a and gyroscope input g using

$$a^{new} = R_y(\alpha)^T \cdot a, \quad g^{new} = R_y(\alpha)^T \cdot g. \tag{4.11}$$

After this reorientation, the IMU data can finally be processed by the subsequently explained sensor fusion method.

Quaternion-based Extended Kalman filter

Due to the advantages quaternions provide over Euler angles (as described in 2.5.2), I have decided to implement a quaternion-based Extended Kalman filter as a sensor fusion method. As part of my work, I designed two variations of that Kalman filter. The first one is based on the work by [Sab11], [FLZ+17] and [WZS15] while the second variation

was a result of my own experimentation with Kalman filters. Henceforth, I will refer to the former as *KF1* and the latter as *KF2*.
After covering the theoretical aspects of the Kalman filter in 2.3.2, in this section, I will describe how I designed the Kalman filter for processing IMU data and outputting quaternions as orientation estimates.

KF1 Design

I will explain the design of this Kalman filter using its basic components which are the state vector, dynamic model and observation model (as explained in 2.3.2):

- **The state vector**: Since this Kalman filter is quaternion-based, the state vector is represented by a 4D vector containing a quaternion which is expressed by $x_k = [q0, q1, q2, q3]$ [FLZ+17] and is always normalized after each frame.
- **The dynamic model**: Angular velocities outputted by 3D gyroscopes are measured in the body frame which allows the kinematic equations of a rigid body to be used in calculating the orientation state. In this case, angular velocities are treated as external inputs to the Kalman filter as opposed to measurements and the gyroscope noise is considered to be process noise and not measurement noise. The advantage of this approach is the reduced dimension of the state vector which results in a more efficient Kalman filter algorithm [Sab11].
  The state transition matrix A can be derived from the quaternion derivative equation $\dot{q} = \frac{1}{2}\Omega(w)q$ using the angular velocity vector ω (see Chapter 2.5.2). In order to project the state x_k forward in time, the equation

$$x_{k+1} = f(x_k, w_k) = x_k + .\frac{\Delta t}{2}\Omega(w_k)x_k \tag{4.12}$$

  can be used (Δt is the time interval between x_k and x_{k+1}) [Sab11], which can be rewritten as (and as result derive the state transition matrix A)

$$\begin{aligned} x_{k+1} &= f(x_k, w_k) = A_k \cdot x_k, \\ A_k &= \begin{bmatrix} 1 & -\frac{\Delta t}{2}w_{k,1} & -\frac{\Delta t}{2}w_{k,2} & -\frac{\Delta t}{2}w_{k,3} \\ \frac{\Delta t}{2}w_{k,1} & 1 & \frac{\Delta t}{2}w_{k,3} & -\frac{\Delta t}{2}w_{k,2} \\ \frac{\Delta t}{2}w_{k,2} & -\frac{\Delta t}{2}w_{k,3} & 1 & \frac{\Delta t}{2}w_{k,1} \\ \frac{\Delta t}{2}w_{k,3} & \frac{\Delta t}{2}w_{k,2} & -\frac{\Delta t}{2}w_{k,1} & 1 \end{bmatrix}. \end{aligned} \tag{4.13}$$

  One difference in my approach in creating the dynamic model for the Kalman filter, is that I use the second-order Runge-Kutta method (or midpoint method) for numerical integration as opposed to the Euler method in order to increase the accuracy of the estimated state (see 2.4.2). This method takes as input two consecutive angular velocity vectors which are then averaged. Thus, 4.13 becomes

$$x_{k+1} = f(x_k, \frac{w_k + w_{k+1}}{2}). \tag{4.14}$$

Finally, the process noise covariance matrix Q representing the gyroscope noise is a 4x4 identity matrix multiplied by a selected predefined value.

- **The observation model**: Here, I will establish an observation model by using the 3D accelerometer data as measurements z. In the work by [Sab11], [FLZ+17] and [WZS15], an observation model is described using the data from the magnetometer sensor as well which is not part of my proposed system.
  First, I will define the normalized gravity vector in the navigation frame n which is $g^n = [0\ 0\ 1]^T$ and the accelerometer vector in the body frame b which is $a^b = [a_x\ a_y\ a_z]^T$. The predicted state vector x expressed as a quaternion q describes the orientation of the body frame with respect to the navigation frame and as described in chapter 2.5.2, a vector can be rotated by a matrix representation of a quaternion $R(q)$. Utilizing that information, a^b can be expressed as

$$\begin{aligned} a^b &= R(q)^T \cdot g^n \\ &= \begin{bmatrix} g_x^n(0.5 - q_y^2 - q_z^2) + g_y^n(q_w q_z + q_x q_y) + g_z^n(q_x q_z + q_w q_y) \\ g_x^n(q_x q_y + q_w q_z) + g_y^n(0.5 - q_x^2 - q_z^2) + g_z^n(q_w q_x + q_y q_z) \\ g_x^n(q_w q_y + q_x q_z) + g_y^n(q_y q_z + q_w q_x) + g_z^n(0.5 - q_x^2 - q_y^2) \end{bmatrix}. \end{aligned} \tag{4.15}$$

  Since $g^n = [0\ 0\ 1]^T$, I can simplify the equation above and subsequently define the observation model

$$h(x_k) = \begin{bmatrix} (x_{k,1}x_{k,3} + x_{k,0}x_{k,2}) \\ (x_{k,0}x_{k,1} + x_{k,2}x_{k,3}) \\ (0.5 - x_{k,1}^2 - x_{k,2}^2) \end{bmatrix} \text{ [WZS15].} \tag{4.16}$$

  When calculating the measurement innovation, both z_k and $h(x_k)$ need to be normalized.
  Due to the non-linearity of the measurement equation, the Jacobian matrix H needs to be computed as well which can be written as

$$H(x_k) = \frac{\delta h(x_k)}{\delta x} = \begin{bmatrix} \frac{\delta h(x_k)}{\delta x_0} & \frac{\delta h(x_k)}{\delta x_1} & \frac{\delta h(x_k)}{\delta x_2} & \frac{\delta h(x_k)}{\delta x_3} \end{bmatrix}. \tag{4.17}$$

  and be further simplified to

$$H(x_k) = \begin{bmatrix} -x_{k,2} & x_{k,3} & -x_{k,0} & x_{k,1} \\ x_{k,1} & x_{k,0} & x_{k,3} & x_{k,2} \\ x_{k,0} & -x_{k,1} & -x_{k,2} & x_{k,3} \end{bmatrix} \text{ [WZS15].} \tag{4.18}$$

KF2 Design

The static vector and dynamic model of this variation of the Kalman filter are identical to *KF1*. The main difference lies in the observation model where the measurement vector is another quaternion. First, I calculate the roll and pitch angles from the accelerometer vector using

$$roll_{acc} = \arctan_2(a_y, a_z), \quad pitch_{acc} = \arctan_2(-a_x, \sqrt{a_y^2 + a_z^2}) \text{ [dfr23].} \tag{4.19}$$

Since I cannot infer the yaw angle from the accelerometer, I use the state vector quaternion to read the existing yaw angle. This can be done by converting the quaternion q to Euler angles (roll, pitch and yaw) [D+06] using

$$\begin{bmatrix} roll_x \\ pitch_x \\ yaw_x \end{bmatrix} = \begin{bmatrix} \arctan_2(2 * (q_0 * q_1 + q_2 * q_3), 1 - 2 * (q_1^2 + q_2^2)) \\ \arcsin(2 * (q_0 * q_2 - q_3 * q_1)) \\ \arctan_2(2 * (q_0 * q_3 + q_1 * q_2), 1 - 2 * (q_2^2 + q_3^2)) \end{bmatrix}. \tag{4.20}$$

After getting the yaw angle from the state vector quaternion x, the measurement quaternion vector z can be formed by converting the Euler angles $roll_{acc}$, $pitch_{acc}$ and yaw_x to a quaternion [D+06] using

$$\begin{bmatrix} z_w \\ z_x \\ z_y \\ z_z \end{bmatrix} = \begin{bmatrix} \cos(\frac{roll_{acc}}{2})\cos(\frac{pitch_{acc}}{2})\cos(\frac{yaw_x}{2}) + \sin(\frac{roll_{acc}}{2})\sin(\frac{pitch_{acc}}{2})\sin(\frac{yaw_x}{2}) \\ \sin(\frac{roll_{acc}}{2})\cos(\frac{pitch_{acc}}{2})\cos(\frac{yaw_x}{2}) - \cos(\frac{roll_{acc}}{2})\sin(\frac{pitch_{acc}}{2})\sin(\frac{yaw_x}{2}) \\ \cos(\frac{roll_{acc}}{2})\sin(\frac{pitch_{acc}}{2})\cos(\frac{yaw_x}{2}) + \sin(\frac{roll_{acc}}{2})\cos(\frac{pitch_{acc}}{2})\sin(\frac{yaw_x}{2}) \\ \cos(\frac{roll_{acc}}{2})\cos(\frac{pitch_{acc}}{2})\sin(\frac{yaw_x}{2}) - \sin(\frac{roll_{acc}}{2})\sin(\frac{pitch_{acc}}{2})\cos(\frac{yaw_x}{2}) \end{bmatrix}. \tag{4.21}$$

Here, the observation model is simply $h(x_k) = x_k$ and the observation matrix H is simply a 4x4 identity matrix.
By using the accelerometer to calculate the Euler angles, I am introducing a major flaw to this design because the calculated angles are only stable in certain ranges. The pitch angle can only be reliably calculated in the range +/-90 degrees. If the calculated angle exceeds that range, then the sensor fusion algorithm should only rely on the gyroscope for estimating the attitude. In *KF1*, the Kalman gain is automatically adjusted to such scenarios but this is not the case in this Kalman filter variation. Therefore, I must manipulate the Kalman gain by external inputs at runtime. I check if acc_z is higher that a certain threshold (level of tilt), where the pitch angle can still be reliably calculated. If that is not the case, I set the Kalman gain matrix values to zero, thus forcing the Kalman gain to only trust the gyroscope values. If acc_z is again higher that the predefined threshold, then I replace the Kalman gain zero-matrix with last Kalman gain matrix before the manipulation.
However, this proposed Kalman filter is still not yet fully stable. The reason is that a rotation can be represented by two different unit quaternions, where one is the negative of the other. Therefore, before I calculate the measurement innovation using z, I check whether the scalar parts of z and x match. If not, I negate z which solves this instability problem.

4.3.4 Data preparation and transmission

After the orientation is calculated using the sensor fusion algorithm described above, the data is prepared to be send either as a quaternion or as Euler angles. If the user wishes to receive the orientation as a quaternion, no further operation is needed when preparing the data for transmission, since the state vector of the Kalman filter is a quaternion. If Euler angles are desired, then the quaternion is converted to Euler angles using the equation 4.20.

Body part	Index
Head	0
Chest	1
Tailbone	2
Left Upper Arm	3
Left Lower Arm	4
Right Upper Arm	5
Right Lower Arm	6
Left Upper Leg	7
Left Lower Leg	8
Right Upper Leg	9
Right Lower Leg	10

Table 4.1: A lookup table which shows the index assigned to each body part.

If the user desires to receive only the raw IMU data from the motion tracker, then the data correction and sensor fusion processes are skipped to reduce the number unnecessary calculations on the microcontroller. The raw data consists of the 3D acceleration vector, the 3D angular velocity vector and the value *deltaTime*, which corresponds to the time interval between two consecutive samplings of the IMU data. The reason the raw IMU data is sampled at 60Hz, is to ensure that the *deltaTime* value is consistent with the sending rate which is also 60Hz. Since the step size *deltaTime* is important in the numerical integration part of the sensor fusion algorithm, it needs to be as accurate as possible when the motion capture application on the receiving computer is performing sensor fusion instead of the microcontroller.
No matter in what format the data is sent, it includes the body part index which is derived from the assigned body part in the configuration software using the lookup table 4.1.
The entire process of preparing and transmitting the data is further illustrated in Figure 4.7.

4.3.5 Motion capture application

I developed the motion capture application which receives the motion data from the motion tracker and applies it to a 3D fully rigged humanoid character [3dc23] using the 3D gaming engine *Unity*.
By listening to a specified UDP port in a dedicated *thread*, the transmitted data from the motion trackers can be received in the *Unity* application. The received data is parsed and according to the structure of the data different actions are triggered. If the structure is recognized as data containing the raw IMU information, the sensor fusion algorithm, previously described in 4.3.3, is performed on that data. If the structure is recognized as data containing quaternions or Euler angles, then the computed orientation by the motion tracker is directly fed to the tracked virtual object.

Unity uses a left-handed Y-up coordinate system, which is different from the coordinate system of the IMU chosen in my system (MPU-6050), which has a right-handed Z-up coordinate system. Therefore, every orientation calculated from the sensor data needs to be mapped to *Unity*'s coordinate system using the following mapping:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{IMU} \Rightarrow \begin{bmatrix} x \\ -z \\ y \end{bmatrix}_{Unity} . \tag{4.22}$$

After getting the new orientation, I use the body part index to map it to the targeted body segment of the virtual character. However, since the motion tracker can face different directions depending on where it is placed on the subject's body, there needs to be another axis manipulation as can be seen and explained in detail in Figure 4.8.
Finally, after each calculated rotation (a quaternion) a final correction is performed. Even after reorienting the sensor data as described in 4.3.3, the first calculated orientation might still not be parallel to the ground (rotation about z-axis is zero) even if the tracker is placed parallel to the ground as well. To fix that, the calculated quaternion is rotated using the initial quaternion saved after the first orientation calculation for a particular body segment. This can be done by

$$q_{new} = q_{new} * q^*_{initial}. \tag{4.23}$$

In order for the motion capture process to work correctly, the capture subject must first stand in the N-pose similar to Figure 3.3.

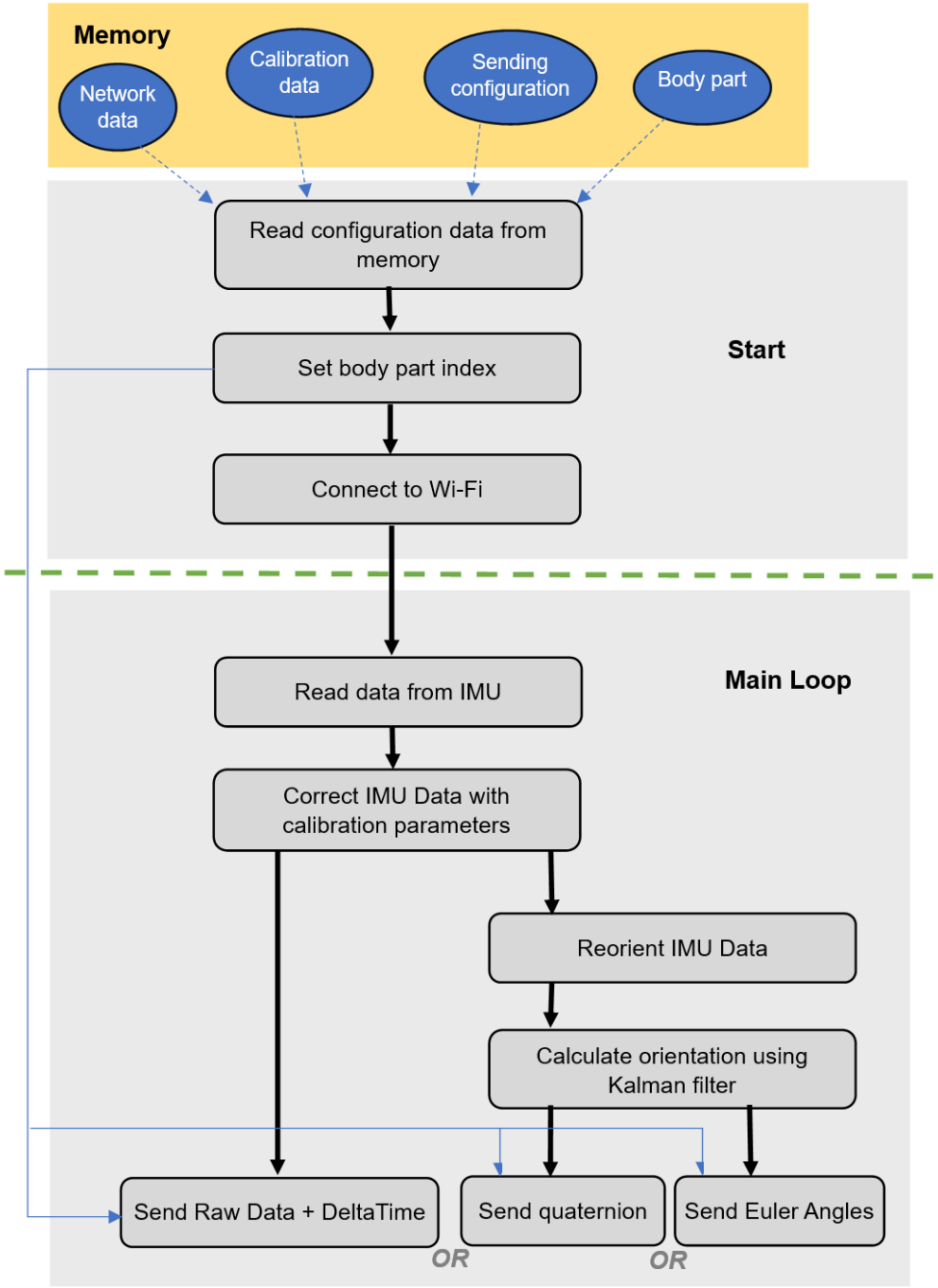
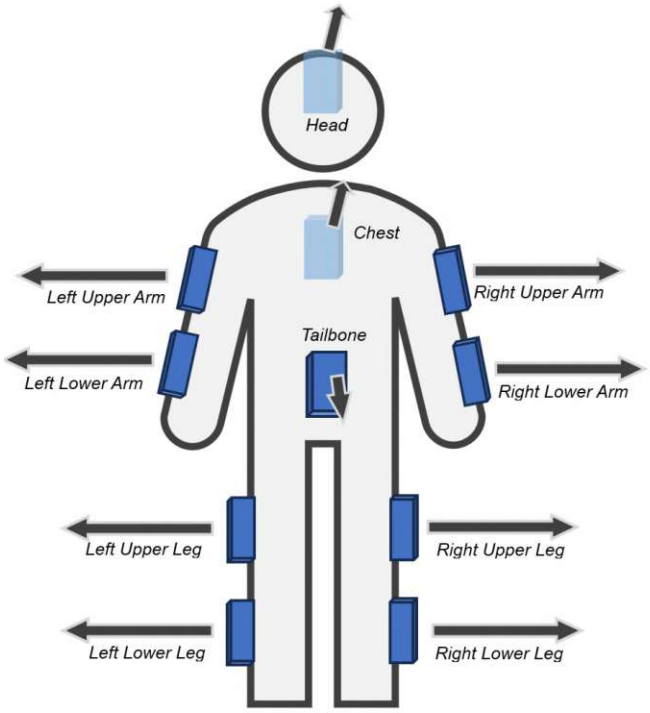


Figure 4.7: Shows the data preparation and transmission process performed by each motion tracker.



Body part	Axes direction flip
Head	$(x, y, -z)$
Chest	$(x, y, -z)$
Tailbone	$(0, y, 0)$
Left Upper Arm	(x, y, z)
Left Lower Arm	(x, y, z)
Right Upper Arm	$(-x, y, -z)$
Right Lower Arm	$(-x, y, -z)$
Left Upper Leg	(x, y, z)
Left Lower Leg	(x, y, z)
Right Upper Leg	$(-x, y, -z)$
Right Lower Leg	$(-x, y, -z)$

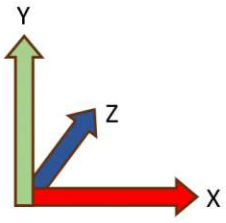


Figure 4.8: Shows motion trackers attached to the capture subject (from behind). The coordinate axes in the bottom-right corner of the figure represent the coordinate axes in *Unity* whose coordinate system I assume corresponds to the real-world coordinate system (navigation frame). Each motion tracker's IMU's z-axis (body frame) faces a different direction depending on where it is placed on the body. The directions are visualized by the arrows (trackers attached to the head and chest face towards the navigation frame's positive z-axis and the tracker attached to the tailbone faces towards the navigation frame's negative z-axis). The table in the top-right corner of the image shows how for each targeted body part the axes of the orientation in *Unity* has to be changed. Since in my application, the tailbone is only responsible for the yaw-movements of the capture subject, the x and z values of its rotation in *Unity* are set to zero.

CHAPTER 5

Hardware Implementation

I this chapter, I will detail my approach on how I assembled the motion trackers and the *Mini-Robot*.

5.1 3D printing

I have 3D printed the chassis of all 11 motion trackers as well as the rig for the *Mini-Robot* using the *Anycubic i3 Mega S* 3D printer. For the filament, I chose PLA and I 3D printed with the following parameters (using the 3D printing software *Ultimaker Cura*):

- **Layer Height**: 0.2
- **Printing temperature**: 200°C
- **Build Plate Temperature**: 60°C

First, I 3D-modeled the chassis and the robot rig in the 3D-software *Maya* which I then exported to **stl**-files. The 3D model for the chassis can be seen in Figure 5.1 and the model for the rig can be seen in Figure 5.2. The dimensions of each motion tracker chassis is 6.3 x 4.2 x 2.7 cm. The base of the *Mini-Robot* rig has the dimensions 10 x 7 x 1 cm while the maximum height of the assembled robot is 15 cm.
Since I want the side of the motion tracker chassis to be see-through (to be able to see the LEDs illuminated when the trackers are powered on), I decided not to print the side cover for the chassis. Instead, I manually cut 2 mm acrylic glass (6.3 x 4.2 cm) which I then mounted onto the open side of the chassis using all-purpose waterproof glue. Before gluing the glass, however, I had to cut out the corner of each glass piece (2 x 1 cm) in order to access the toggle switch used for turning the trackers on and off.
Due to occasional irregularities of the surfaces of the final prints, especially around and

inside the screw holes, I had to manually embellish the printed models using a variety of file tools.

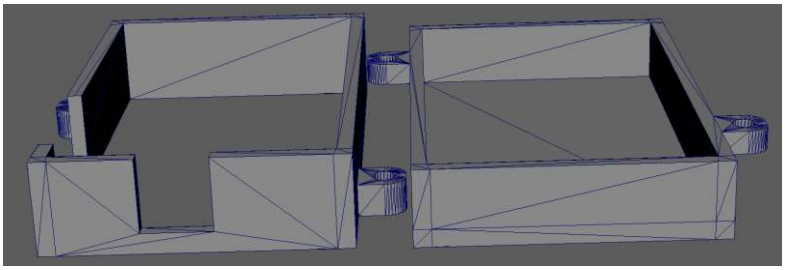


Figure 5.1: Shows the 3D model of the motion tracker chassis.

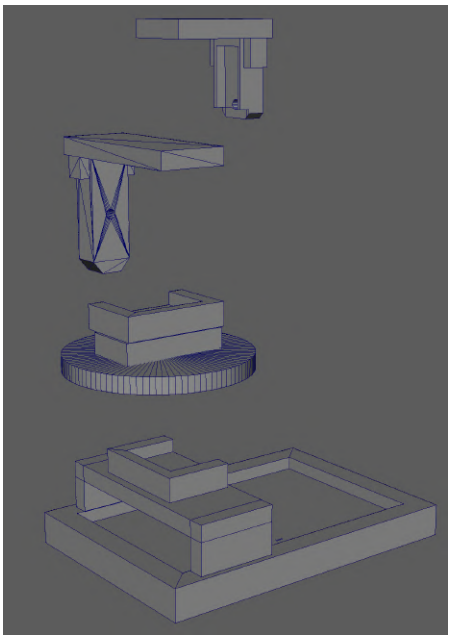


Figure 5.2: Shows the 3D model of the *Mini-Robot* rig.

5.2 Motion trackers

In this section, will give an overview of the wiring, PCB design and describe how I assembled the motion trackers while adding some degree of modularity to them.

5.2.1 Wiring

Here, I want to detail the connections between the individual hardware components I used for assembling each motion tracker:

- Connect the lithium battery's power wire to the TP4056's positive **B** pin and the battery's ground wire to the negative **B** pin.
- The TP4056's negative **OUT** pin is connected to the ESP32's **GND** pin. The TP4056's positive **OUT** pin is connected to the middle pin of the 3-pin toggle switch.
- The left or right pin of the toggle switch is connected to the **anode** of the Schottky diode. The Schottky diode's **cathode** is connected to the electrolytic capacitor's **positive** pin.
- The electrolytic capacitor's **positive** pin is connected to the ESP32's **3v3** pin. The electrolytic capacitor's **negative** pin is connected to the ESP32's **GND** pin.
- The MPU-6050's **VCC** pin is connected to the ESP32's **3v3** pin and its **GND** pin is connected to the ESP32's **GND** pin. The MPU-6050's **SCL** pin is connected to the ESP32's **GPIO22** pin. The MPU-6050's **SDA** pin is connected to the ESP32's **GPIO21** pin.
- The ESP32's **GPIO5** pin is connected to the resistor and the resistor is connected to the LED's **cathode**. The LED's **anode** is connected to the ESP32's **GND** pin.

5.2.2 PCB

In order to build a compact motion tracker, I designed a PCB to minimize the number of wires needed to connect the hardware components. I designed the PCB using the open-source software *Fritzing* [fri]. To reduce the size and simultaneously avoid the intersections of the individual routes, I decided to use a two-layer PCB. When designing the PCB, as suggested by [SSM19], I avoided bends with 90 degree angles when placing the routes and replaced them by 45 degree bends instead. In my final design, the PCB has the dimensions of 53 x 35 x 1 mm.
Figure 5.3 show both the PCB design in *Fritzing* as well as the manufactured PCB which I have ordered from [jlc].

5.2.3 Modularity

As previously mentioned, I tried to introduce a certain degree of modularity to the motion trackers in case of faulty or damaged hardware components. In my proposed motion tracker, it is possible to remove and exchange both the IMU and the battery. Also, hard-soldering the IMU to the PCB, would make it difficult to rerun the calibration process as the IMU would need to be inserted into the IMU socket of the *Mini-Robot* for data collection.
Instead of soldering the IMU itself, I soldered a 4-pin female header where the IMU can be inserted. Similarly for the battery which has a *Micro JST 1.25 connector*, I soldered a *Micro JST 2-pin female connector* into the two pins on the PCB intended for the battery (see Figure 5.4).

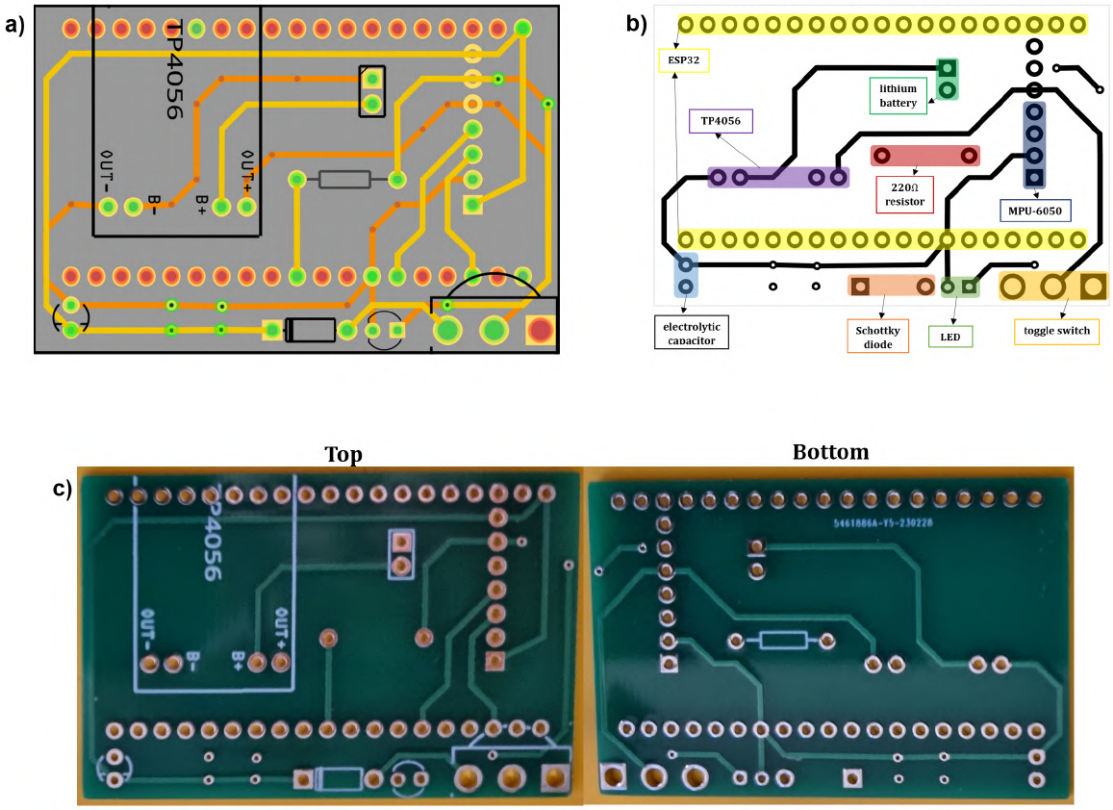


Figure 5.3: **a)** Shows my two-layer PCB design in *Fritzing*. The light-orange routes are the traces on the top layer and the dark-orange routes are the traces on the bottom layer. **b)** Shows where each hardware component is placed on the PCB. **c)** Shows the real manufactured PCB I received after placing the order.

5.2.4 Assembly

Apart from the IMU and the battery, all hardware components are placed on the PCB's top layer and soldered. The ESP32 is the last component whose pins are soldered to the PCB because several hardware elements such as the resistor and the TP4056 lie directly under the microcontroller which is possible due to the elevation provided by the ESP32's male pin headers. Additionally, I completely removed several of the ESP32's male pin headers to ensure accessibility and enough space for the TP4056's USB-C port. Since the battery is placed directly under the PCB, I had to make sure that all the ESP32's metal pins sticking out of the PCB bottom layer are cut off.
I soldered 90 degree bend male 4-pin headers to the MPU-6050 which allowed me to insert it into the 4-pin female header (see Figure 5.4). However, the IMU will not fit unless the male pins are bent to an obtuse angle, which eventually results in the IMU output correction step explained in 4.3.3.
The assembled motion tracker, before it is inserted into the chassis, can be seen in Figure

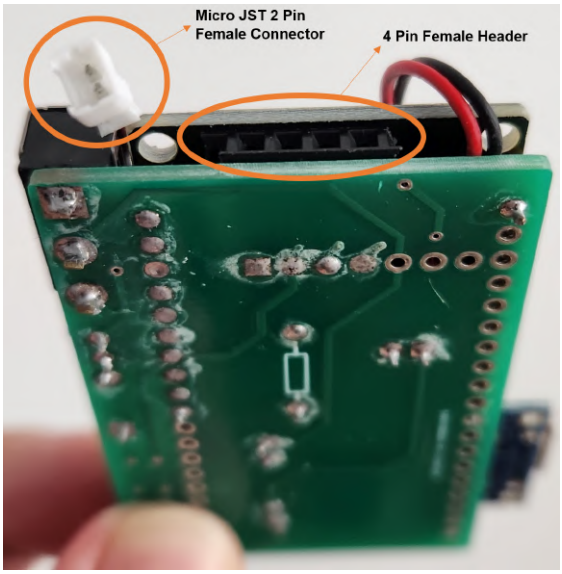


Figure 5.4: Shows the 4-pin female header for the IMU and the Micro JST 2-pin female connector for the battery.

5.5. The motion tracker inside the 3D printed chassis with the glued-on strap can be seen in Figure 5.6. The weight of each motion tracker is around 75 grams.
I have repeated this process for all 11 motion trackers.

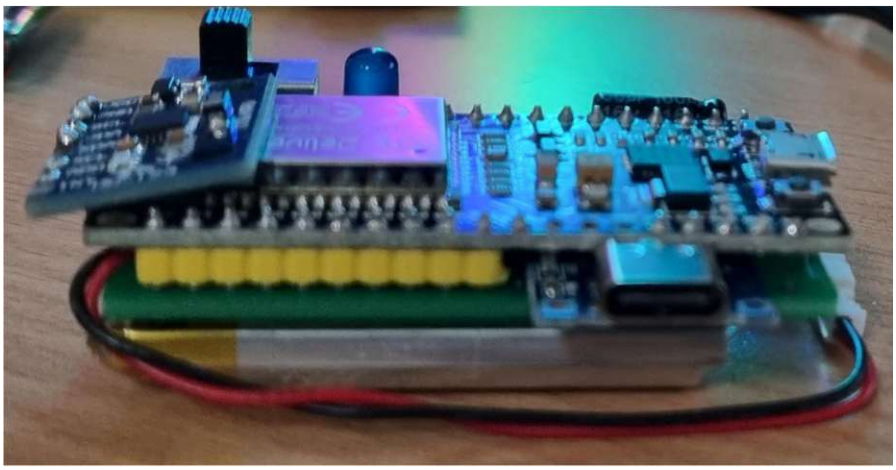


Figure 5.5: Shows the assembled motion tracker without its chassis.

5.3 Mini-Robot for Calibration

In this section, will give an overview of the wiring and assembly of the *Mini-Robot*.

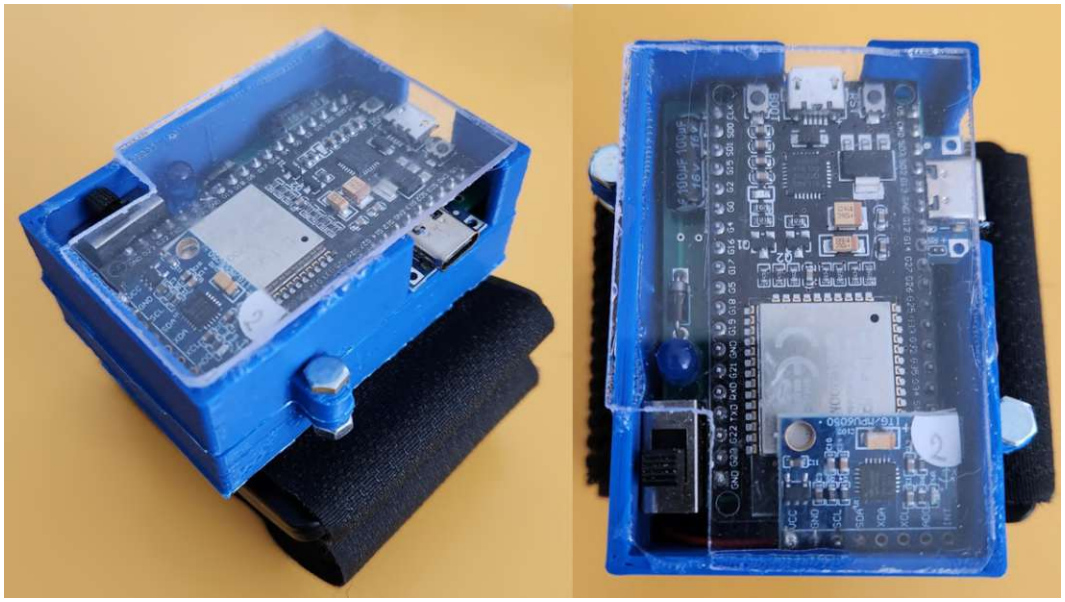


Figure 5.6: Shows the assembled motion tracker with its chassis and glued on body strap.

5.3.1 Wiring

Similarly to the motion tracker, I will detail the connections between the individual hardware components I used to assemble the *Mini-Robot*:

- Each SG90 9g micro servo motor's **VCC** pin is connected to the ESP32's **V5** pin and each servo motor's **GND** pin is connected to the ESP32's **GND** pin. All three servo motor's signal pins are connected to the ESP32's **GPIO12**, **GPIO13** and **GPIO14** pins, respectively.
- The MPU-6050's **VCC** pin is connected to the ESP32's **3v3** pin and its **GND** pin is connected to the ESP32's **GND** pin. The MPU-6050's **SCL** pin is connected to the ESP32's **GPIO22** pin. The MPU-6050's **SDA** pin is connected to the ESP32's **GPIO21** pin.

5.3.2 Assembly

I decided against designing a PCB for the *Mini-Robot* since I needed the weight of the breadboard which in addition to the 3D printed base provides good enough stability when the robot arm is in movement. All individual components of the 3D printed rig are either glued together or attached to the servo horns using screws. Similar to the motion tracker, a 4-pin female header is glued to the top surface of the robot arm so that the IMU can be easily attached or detached during the calibration process. The assembled *Mini-Robot*, which has a weight of 170 grams, can be seen in Figure 5.7.

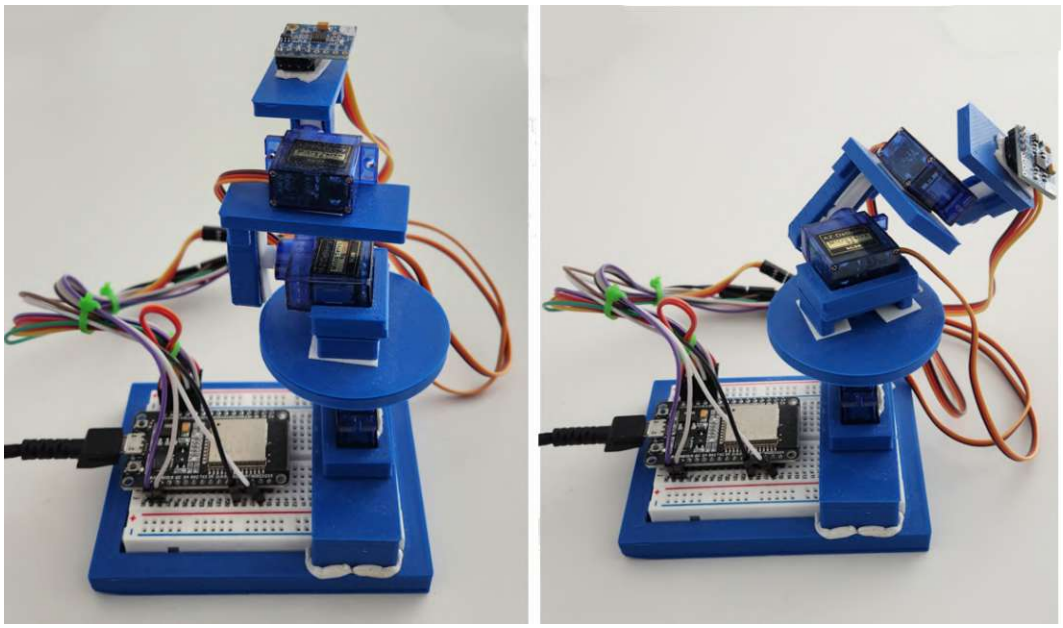


Figure 5.7: Shows the *Mini-Robot* in the idle position (left) and with the robot arm rotated in a random orientation (right).

CHAPTER 6

Software Implementation

I this chapter, I will detail my implementation approach for the software which I developed to process the motion capture data. First, I will discuss my implementation for the calibration software and move into the implementation of the attitude estimation approach using the Kalman filter on both the *ESP32* and in the *Unity* game engine.

6.1 Calibration

The calibration procedure can be divided into two key steps:

1. Data collection using the *Mini-Robot* and recording
2. Accelerometer and gyroscope calibration

In this section, I will focus only on the software implementation for the data collection and calibration process.

6.1.1 Data collection and recording

The script containing the instructions (see Algorithm 6.1) for controlling the robot and collecting the data was written by me in the *Arduino IDE* environment. Here is a breakdown of the important parts of the algorithm:

- **Line 1-7**: These instructions are executed in the *start()* function of the *Arduino* script. In the first instruction, I set the three servo motors to their start positions so that the top-most surface of the *Mini-Robot* lies parallel to the ground. After that I set the full-screen range of the IMU. I set the accelerometer sensitivity to $2g$, which means that the raw accelerometer data needs to be divided by 16384,

and the gyroscope sensitivity to $250°/sec$, which means that the raw gyroscope data needs to be divided by 131. This ensures that both sensors have the highest possible sensitivity.

- **Line 8**: Here we enter the *loop()* function of the *Arduino* script. N_s is the number of static positions, which the robot arm has to rotate to. I have set $N_s = 60$.
- **Line 10-11**: Here is where I read the data from the IMU.
- **Line 12-14**: Before setting the different static positions, the gyroscope bias needs to be calculated. For that, the robot arm stands still in its idle position until N_g angular velocity samples are read from the gyroscope sensor. Afterwards, the bias is calculated by averaging all collected samples across each axis and the result is stored in a 3D vector. For N_g, I chose 10000 to achieve a high accuracy for the resulting bias vector.
- **Line 15-21**: This step is reached after the gyroscope bias was calculated and if enough time has passed since the last time a new static position was set. If that condition is satisfied, the robot arm is set in a random orientation. This is done by passing three values to the three servo motors which are responsible for setting the roll, pitch and yaw rotations. These values are in the range between 0 and 180. Immediately after setting the robot arm's new orientation, the *boolean* flag $IMUinMotion$ is set to true to indicate that from this time on all the data that is gathered is stored while the IMU is moving. Additionally, the $attitudeNum$ counter variable is increased to mark the new static position. T_{static} is the time span the robot arm should remain in a static position, which I have set to 5.5 *seconds*. T_{motion} is the amount of time the robot arm should wait (after moving to a new position) before marking the collected data with $IMUinMotion = False$. In my implementation T_{motion} is set to 4.5 *seconds*. Due to the uneven structure of the robot arm, quick movements of the servo motors lead to jittering which affect the IMU readings. Therefore, T_{motion} makes sure to wait until the jittering of the robot arm has diminished before entering the static state.
- **Line 24**: During each iteration and only after the gyroscope bias was calculated, all the collected data is printed to the serial port.

While the data is printed to the serial port, I record it using the application *SerialPlot*. This is a Qt-based software for reading data from the serial port while being able to plot and record that data in real-time (see Figure 6.1) [ser23]. For that, the *Mini-Robot* needs to be connected to the computer, on which this software is running, via USB. After connecting it and choosing the data format, the recording in *SerialPlot* must be initialized before the data starts to get printed to the serial port in order to not lose any information.
After the *Mini-Robot* has reached its final static position and has finished printing to the serial port, the data gathered by *SerialPlot* is saved in a file, which I then convert to a *CSV*-file. In Figure 6.2 I explain the structure of the stored file.

Algorithm 6.1: Mini-Robot IMU data collector

```
1  Set Robot Arm in idle position
2  Set IMU sensitivity
3  gyroBias ← {0, 0, 0}                          ▷ Vector containing gyroscope bias
4  attitudeNum ← −1                              ▷ Index of current static position
5  gyrocalibrated ← False   ▷ After gyroscope bias is calculated this is set to true
6  IMUinMotion ← False      ▷ If robot arm is moving then this variable is true
7  deltaTime ← 0            ▷ Time passed between this frame and last frame
8  while attitudeNum < N_s do
9      deltaTIme = CalculateDeltaTime()
10     accData = ReadFromAccelerometer()
11     gyroData = ReadFromGyro()
12     if gyrocalibrated = False and N_g samples were collected then
13         gyroBias ← AverageGyroData()
14     end
15     if gyrocalibrated = True and IMUinMotion = False and
        (T_static + T_motion) time has passed then
16         Set Robot Arm in random position
17         attitudeNum ← attitudeNum + 1
18         IMUinMotion ← True
19     end
20     if gyrocalibrated = True and IMUinMotion = True and
        T_motion time has passed then
21         IMUinMotion ← False
22     end
23     if gyrocalibrated = True then
24         SendData(accData, gyroData, gyroBias, deltaTime, IMUinMotion, attitudeNum)
25     end
26 end
```

6.1.2 Accelerometer and gyroscope calibration

I implemented the calibration algorithm for the accelerometer (see Algorithm 6.2) and for the gyroscope (see Algorithm 6.3 and 6.4) in *Python 3.8.16*.
Here, I will describe the important parts of the Algorithm 6.2:

- **Line 1**: The first step is to read the *CSV*-file and store it in a *Pandas DataFrame*.
- **Line 3-6**: I prepare the data using *numpy* so that in the end I have a matrix where the *i-th* row contains the averaged accelerometer data of the *i-th* static position.
- **Line 7**: Here, I need to set my initial guess for the unknown parameter vector θ^{acc} (see Equation 4.3). After trying the initial guess proposed by [TPM14], which is

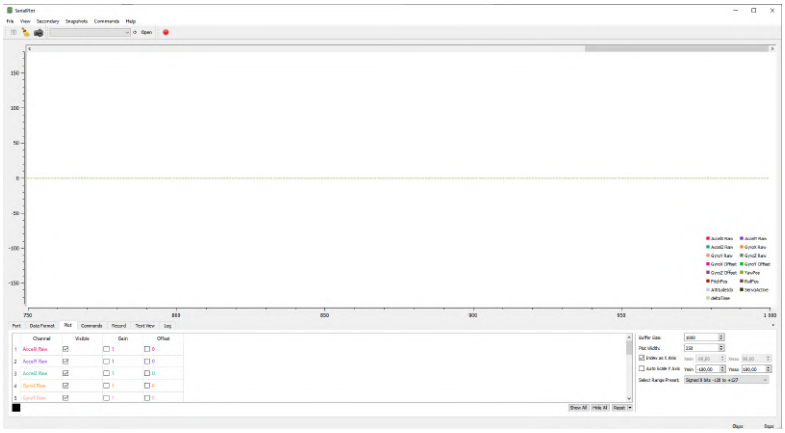


Figure 6.1: *SerialPlot* for reading data from the serial port and plotting it in real-time.

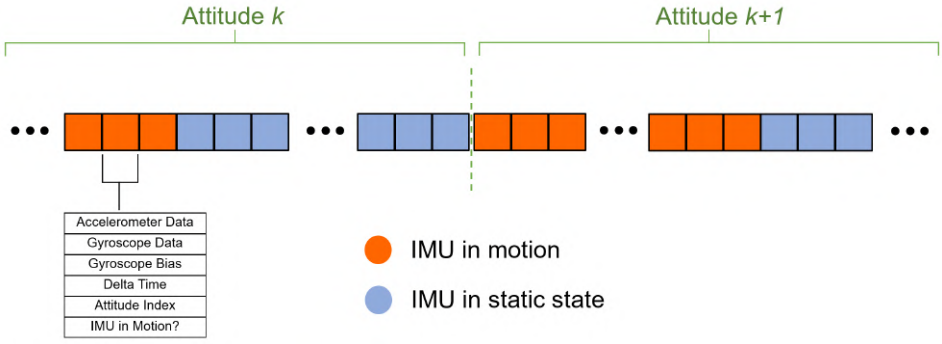


Figure 6.2: This figure illustrates how the recorded data is stored. Each square can be considered a row in the *CSV*-file, that contains all the data (listed in the bottom-left corner) collected during an iteration of the main loop in the *Arduino* script. This diagram shows two consecutive static positions which are separated by the dashed green line. Each static position contains two sets of differently marked data. It starts with a series of data that is collected while the robot arm is moving and ends with a series of data that is collected while the robot arm is in a static position.

$\theta^{acc} = [1, 0, 0, 1, 0, 0]$, I found that the minimization algorithm did not provide good solutions. Afterwards, I changed it to $[0, 0, 0, 1, 1, 1]$, which means that ideally all the axes of the AF coincide with the AOF or BF and that there are no scaling errors. Therefore, these initial values should be closest to the ideal solution if there are not any severe manufacturing errors with the targeted IMU sensor. With this initial guess, I was able to receive good estimates for the parameter vector.

- **Line 8**: In this step, I employ the *Levenberg–Marquardt* algorithm to provide the best possible estimates for θ^{acc}. For that, I use the already existing implementation *scipy.optimize.least_squares* with the parameter *method='lm'* [lm23]. Additionally, I pass the cost function $AccelerometerCostFunc(\theta^{acc}, SP)$ (see Equation 4.5) which

computes the vector of residuals with respect to θ^{acc}.

- **Line 9**: Get the final result for θ^{acc} from the *Python* function described in **line 8**.

Algorithm 6.2: Accelerometer Calibration

```
1  Function CalibrateAccelerometer ()
2      accData, IMUinMotion, attitudeNum ← ReadDataFile()
3      Filter accData where IMUinMotion = False
4      Group accData by same attitudeNum
5      SP ← new [attitudeNum × 3] matrix ▷ Each row contains a static position
6      SP ← Average accData in the same group
7      θ^acc ← Init vector with chosen values
8      results ← LevenbergMarquardt(AccelerometerCostFunc, θ^acc, SP)
9      θ^acc ← results.solution
10 End Function

11 Function AccelerometerCostFunc (θ^acc, SP)
12     T^a, K^a ← Init [3x3] matrices with chosen values θ^acc
13     gravityMag ← 9.8
14     residuals ← Init with empty list
15     for k ← 0 to SP.rows do
16         f ← T · K · SP[i]
17         residual ← (||gravityMag||^2 − ||f||^2)^2
18         residuals[i] ← residual
19     end
20     return residuals
21 End Function
```

Next, I will describe important aspects of the Algorithm 6.3:

- Since the gyroscope calibration uses the acceleromter calibration as a reference, the function *CalibrateGyroscope* takes as input the averaged static positions and the estimated θ^{acc} from Algorithm 6.2.
- **Line 1**: I read the *CSV*-file and store it in a *Pandas DataFrame*.
- **Line 3-11**: I prepare the data using *numpy* so that in the end I have one array AVV where the *i-th* element has a list of 3D vectors containing the angular velocities that transition the *i-1-th* static position into the *i-th* static position and a second array dtV that contains the corresponding $deltaTime$ values.
- **Line 12**: I set my initial guess for the unknown parameter vector θ^{gyro} (see Equation 4.10) to $\theta^{gyro} = [0, 0, 0, 0, 0, 0, 1, 1, 1]$. Similar to my initial guess for θ^{acc},

this means that ideally all the axes of the GF coincide with the AOF or BF and that there are no scaling errors.

- **Line 13**: I employ the *Levenberg–Marquardt* algorithm to provide estimates for θ^{gyro}. I pass the cost function *GyroscopeCostFunc* (see Equation 4.8) which computes the vector of residuals with respect to θ^{gyro}. The function requires additional inputs such as the averaged static positions and the estimated θ^{acc}.

- **Line 14**: Get the final result for θ^{gyro} from the *Python* function described in **line 13**.

- **Line 19-37**: In the cost function, I iterate through all the static positions and, in each iteration, I fetch two consecutive static positions and correct them using the parameter vector θ^{acc} (see **lines 22-23** and Equation 4.2). In **line 24**, I fetch all the angular velocities that should transform the corrected *Attitude* into *NextAttitude*. Between **lines 26-33**, I initialize a quaternion with $[1, 0, 0, 0]$, where the first element is the scalar part, and perform angular velocity integration using that quaternion as the starting point with the angular velocities previously collected. I perform the numerical integration using the fourth-order Runge-Kutta method which is described in Algorithm 6.4 (the operator Ω returns a matrix representation of the angular velocity vector, see Equation 2.30). It is important to note, that the angular velocities must be transformed from degrees to radians by multiplying them with $\frac{PI}{180}$ because the calculated rotation matrix from the quaternion (in **line 33**) will be otherwise incorrect. To calculate the 3x3 rotation matrix from the resulting quaternion see Equation 2.28. Using this rotation matrix, *Attitude* can be transformed and compared with the actual value *NextAttitude* in **line 35**.

The solutions calculated by *CalibrateGyroscope* and *CalibrateAcceleromter* are fed to the **motion tracker calibration software**. I have implemented that application using *C++* and *Qt* for the user interface. For communicating with the ESP32, I used the library *SerialPort* provided by [Man16].

6.2 Attitude estimation

Here, I will describe the implementation details for my attitude estimation approach in both the ESP32 and the Unity motion capture application.
On the ESP32, I set the full-screen range of the accelerometer to $2g$ and that of the gyroscope to $250°/sec$ to ensure that both sensors have the highest possible sensitivity.

6.2.1 Data correction and reorientation

In order to reduce the number of mathematical operations on the ESP32 when correcting the accelerometer and gyroscope output using the calibration parameters θ^{acc} and θ^{gyro},

Algorithm 6.3: Gyroscope Calibration

Function *CalibrateGyroscope* (θ^{acc}, SP)
  $gyroData, gyroBias, deltaTimes, IMUinMotion, attitudeNum \leftarrow$ *ReadDataFile*()
  Filter $gyroData$ where $IMUinMotion = True$ *And* $attitudeNum > 0$
  Filter $deltaTimes$ where $IMUinMotion = True$ *And* $attitudeNum > 0$
  Remove bias from $gyroData$ using $gyroBias$ vector
  $AVV \leftarrow$ *array with size* $(attitudeNum-1)$, *each entry is list of 3D vectors*
  $dtV \leftarrow$ *array with size* $(attitudeNum-1)$, *each entry is list of floats*
  for $i \leftarrow 1$ **to** $attitudeNum$ **do**
    Add $gyroData$ vectors with $attitudeNum = i$ to list $AVV[i-1]$
    Add $deltaTimes$ values with $attitudeNum = i$ to list $dtV[i-1]$
  end
  $\theta^{gyro} \leftarrow$ Init vector with chosen values
  $results \leftarrow$ $LevenbergMarquardt(GyroscopeCostFunc, \theta^{gyro}, \theta^{acc}, SP, AVV, dtV)$
  $\theta^{gyro} \leftarrow results.solution$ and $gyroBias$
End Function

Function *GyroscopeCostFunc* $(\theta^{gyro}, \theta^{acc}, SP, AVV, dtV)$
  $T^g, K^g \leftarrow$ Init [3x3] matrices with chosen values θ^{gyro}
  $residuals \leftarrow$ Init with empty list
  for $k \leftarrow 0$ **to** $SP.rows - 1$ **do**
    $idx \leftarrow k$
    $nextIdx \leftarrow k + 1$
    $Attitude \leftarrow$ CorrectAttitude$(SP[idx], \theta^{acc})$
    $NextAttitude \leftarrow$ CorrectAttitude$(SP[nextIdx], \theta^{acc})$
    $angularVelocities \leftarrow AVV[nextIdx]$
    $deltaTimes \leftarrow dtV[nextIdx]$
    $q \leftarrow \{1, 0, 0, 0\}$ ▷ quaternion with scalar part=1 and vector part zero
    for $i \leftarrow 0$ **to** $angularVelocities.Size - 1$ **do**
      $dt \leftarrow deltaTimes[i]$
      $\omega 0 \leftarrow angularVelocities[i] * \frac{PI}{180}$
      $\omega 1 \leftarrow angularVelocities[i+1] * \frac{PI}{180}$
      $q \leftarrow$ IntegrateRungeKutta4$(q, \omega 0, \omega 1, dt)$
    end
    $R \leftarrow$ GetRotationMatrixFromQuaternion(q)
    $NextAttitudeEst \leftarrow R \cdot Attitude$
    $residual \leftarrow ||NextAttitude - NextAttitudeEst||^2$
    $residuals[i] \leftarrow residual$
  end
  return $residuals$
End Function

Algorithm 6.4: Fourth-order Runge-Kutta integration for quaternions

Input: A quaternion q, angular velocity $\omega 0$ at timestep t, angular velocity $\omega 1$ at timestep $t+1$, step size $deltaTime$ which is time passed between t and $t+1$

Output: quaternion q_{next} at next time step

Function *IntegrateRungeKutta4* $(q, \omega 0, \omega 1, \Delta t)$

/*Ω returns a real matrix representation of the angular velocity vector*/

$q_1 \leftarrow q$

$k_1 \leftarrow 0.5 * \Omega(\omega 0) \cdot q1$

$q_2 \leftarrow q + 0.5 * \Delta t * k_1$

$k_2 \leftarrow 0.5 * \Omega(\frac{\omega 0+\omega 1}{2}) \cdot q2$

$q_3 \leftarrow q + 0.5 * \Delta t * k_2$

$k_3 \leftarrow 0.5 * \Omega(\frac{\omega 0+\omega 1}{2}) \cdot q3$

$q_4 \leftarrow q + \Delta t * k_3$

$k_4 \leftarrow 0.5 * \Omega(\omega 1) \cdot q4$

$q_{next} \leftarrow q + \Delta t * (\frac{k1}{6} + \frac{k2}{3} + \frac{k3}{3} + \frac{k4}{6})$

$q_{next} \leftarrow normalize(q_{next})$

return q_{next}

End Function

I simplified the equations 4.2 and 4.6 to

$$\begin{bmatrix} a_x^O \\ a_y^O \\ a_z^O \end{bmatrix} = \begin{bmatrix} s_x^a \; a_x^S - \alpha_{yz} \; s_y^a \; a_y^S + \alpha_{zy} \; s_z^a \; a_z^S \\ s_y^a \; a_y^S - \alpha_{zx} \; s_z^a \; a_z^S \\ s_z^a \; a_z^S \end{bmatrix} \tag{6.1}$$

and

$$\begin{bmatrix} \omega_x^O \\ \omega_y^O \\ \omega_z^O \end{bmatrix} = \begin{bmatrix} \omega_x^S \; s_x^g - \omega_y^S \; \gamma_{yz} \; s_y^g + \omega_z^S \; \gamma_{zy} \; s_z^g \\ \omega_x^S \; \gamma_{zx} \; s_x^g + \omega_y^S \; s_y^g - \omega_z^S \; \gamma_{xz} \; s_z^g \\ -\omega_x^S \; \gamma_{xy} \; s_x^g + \omega_y^S \; \gamma_{yx} \; s_y^g + \omega_z^S \; s_z^g \end{bmatrix}. \tag{6.2}$$

Next, is the IMU data reorientation step, where the IMU output is rotated based on the skewed angle of the IMU connected to the ESP32 (which is necessary to make the IMU fit inside the chassis). I found that the angle $\alpha = 75$ for the $R_y(\alpha)$ rotation matrix (see 4.11), is a good estimate to ensure that the initial orientation is close to $q = [1, 0, 0, 0]^T$.

6.2.2 Quaternion-based Kalman Filter

I implemented my previously described Kalman filter in both the *Arduino IDE*, which is to be uploaded to the ESP32, and the *Unity* environment.
For the implementation of the *Arduino* script, I created a C++ library containing all the

Kalman filter functions. I utilized the single-header public domain linear algebra library *linalg.h* developed by [Ors] to perform the necessary vector and matrix operations for the Kalman filter algorithm.
In *Unity*, I implemented the Kalman filter in C#. Even though the *Unity* engine provides implementations for matrix data structures and operations, it is limited to only square matrices. Therefore, I used the C# library *Math.NET Numerics* which provides methods and algorithms for numerical computations [Mat]. I imported this library into *Unity* using the *.Net* package manger tool *NuGet*. In order to use *Math.NET Numerics* in a C# script, it is necessary to add the line **"using MathNet.Numerics.LinearAlgebra;"** at the top of the script.

The algorithm used to implement the Kalman filter (specifically *KF1*) in both the above mentioned environments can be seen in 6.5 and here I will describe important aspects of the algorithm:

- **line 11-17**: Here is where the Kalman filter is initialized when the first set of data is received by the IMU. I provide an initial estimate $measurement_{initEst}$ of the state vector by calculating the roll and pitch Euler angles from the accelerometer data while presuming that the yaw angle is zero. Since this Kalman filter is quaternion based, the calculated Euler angles are converted into a quaternion using the equation 4.21. This quaternion is then used as the initial estimate for the state vector x. For $QVal$ and $RVal$, I have chosen the values 0.001 and 10, respectively.
- **line 18**: Since I am using the second-order Runge-Kutta method (midpoint method) in my Kalman filter, I need to average the two angular velocity vectors $\omega 0$ and $\omega 1$ and convert them from degrees to radians.
- **line 19**: Here, I perform the Kalman filter prediction step.
- **line 20**: Here, I perform the Kalman filter correction step.
- **line 22**: I calculate the state transition matrix A using Equation 4.13.
- **line 28**: I calculate the observation model $h(x)$ using Equation 4.16.
- **line 29**: I calculate the Jacobian matrix H using Equation 4.18.
- **line 33**: The state vector x which represents the rotation as a quaternion must always be normalized for it to be considered a pure rotation.

For the implementation of *KF2*, the algorithm differs in a few aspects. The Kalman gain matrix is a 4x4 matrix and the observation matrix H is a 4x4 identity matrix. The measurement noise covariance matrix R is a 4x4 identity matrix multiplied with the same $RVal = 10$ value. The measurement vector z is a 4D vector and is a quaternion representation of the rotation calculated using the accelerometer data as described in **lines 12-14**.

Algorithm 6.5: Quaternion-based Extended Kalman Filter

Function *InitializeKalmanFilter ()*
  $x \leftarrow [1,0,0,0]$ ▷ state vector (quaternion with scalar part=1)
  $A \leftarrow$ Init [4x4] identity matrix ▷ state-transition matrix
  $H \leftarrow$ Init [3x4] matrix ▷ observation matrix
  $P \leftarrow$ Init [4x4] identity matrix ▷ process covariance matrix
  $Q \leftarrow$ Init [4x4] identity matrix * $QVal$ ▷ process noise covariance matrix
  $R \leftarrow$ Init [3x3] identity matrix * $RVal$ ▷ measurement noise covariance matrix
  $K \leftarrow$ Init [4x3] matrix ▷ Kalman gain matrix
End Function

Input: Receive following data from IMU every frame and pass them as parameters to the Kalman Filter: The 3D accelerometer vector a, 3D angular velocity vector $\omega 0$ at timestep t, 3D angular velocity vector $\omega 1$ at timestep $t+1$, step size dt which is time passed between t and $t+1$

Function *ExtendedKalmanFilter*$(a, \omega 0, \omega 1, dt)$
  if $KalmanFilterNotIntialized$ **then**
    $roll \leftarrow \arctan_2(a_y, a_z)$
    $pitch \leftarrow \arctan_2(-a_x, \sqrt{a_y^2 + a_z^2})$
    $measurement_{initEst} \leftarrow EulerAngleToQuaternion([roll, pitch, 0])$
    InitializeKalmanFilter()
    $x \leftarrow measurement_{initEst}$
  end
  $\omega_{avg} = \frac{\omega 0 + \omega 1}{2} * \frac{PI}{180}$
  predict(ω_{avg}, dt)
  correct(a)
End Function

Function *predict* (ω, dt)
  $A \leftarrow GetStateTransitionMatrix(\omega, dt)$
  $x \leftarrow A \cdot x$
  $P \leftarrow A \cdot P \cdot A^T + Q$
End Function

Function *correct* (z)
  $h \leftarrow ObservationModel(x)$
  $H \leftarrow GetObservationMatrix(x)$
  $K \leftarrow P \cdot H^T \cdot (H \cdot P \cdot H^T + R)^{-1}$
  $x \leftarrow x + K \cdot (normalized(z) - normalized(h))$
  $P \leftarrow (I - K \cdot H) \cdot P$
  $x \leftarrow normalized(x)$
End Function

6.2.3 Receiving and processing motion capture data in Unity

In order to receive the IMU data from the motion trackers, I implemented a UDP listener as a Unity component. For that, I used the already existing **UdpClient** class under **namespace System.Net.Sockets**. For the port, I used the same one that was set in the configuration software and the buffer size of the received data was set 120000. Using **System.AsyncCallback**, I defined a function to be called every time IMU data is received.
The incoming data is separated by whitespaces and stored in an array. By checking against the length of the array, I can deduct what type of data was sent. If the data is detected as the already calculated orientation in Euler angels or quaternion, then the information is passed directly to the assigned body segment of the virtual avatar depending on the body part index. If the data is the raw IMU data, then the orientation is first calculated using the Algorithm 6.5 before passed to the assigned body segment.

CHAPTER 7

Technical Evaluation

In this chapter, I will perform technical evaluations for my implemented calibration procedure as well as the sensor fusion algorithms I implemented in order to correctly estimate the orientations of the motion trackers.

7.1 Calibration

There are two aspects of my calibration procedure which I will evaluate in this section. First, I will present my evaluation results for how many static positions are necessary to achieve a good estimate for the calibration parameters θ^{acc} and θ^{gyro}. Then, I will discuss the effect of the calibration parameters on the collected IMU data.

7.1.1 Parameter estimation

During my implementation, I collected 60 static positions for every IMU in order to estimate the calibration parameters. For the evaluation, I took a subset of these static positions and calculated the parameter vectors θ^{acc} and θ^{gyro}. I started with 10 static positions and incrementally increased that number by 5 until 60 static positions were reached. I calculated the relative error between every solution for θ^{acc} and θ^{gyro} and the final solution for θ^{acc} and θ^{gyro} where 60 static positions were used.
The relative error for vectors is defined as

$$e_{rel} = \frac{||\hat{x} - x||}{||x||} \text{[Bin16]} \tag{7.1}$$

where $\hat{x}$ is an approximation of the true value x. In my case, x represents the calibration parameters calculated using 60 static positions. To show these relative errors as percentages every e_{rel} is multiplied by 100. I performed this evaluation process for three

IMU 1		IMU 2		IMU 3	
	MAE		MAE		MAE
Before Correction	0.358	Before Correction	0.207	Before Correction	0.378
After Correction	0.046	After Correction	0.0516	After Correction	0.062

Table 7.1: Shows the mean absolute error between the magnitude of the vectors obtained from the accelerometer while in a static position (before and after they were corrected using θ^{acc}) and the real gravity vector magnitude. Each table represents the results for one IMU.

different IMUs (which I will refer to as IMU 1, IMU 2 and IMU 3) and the results are presented in Figure 7.1.

Discussion of results

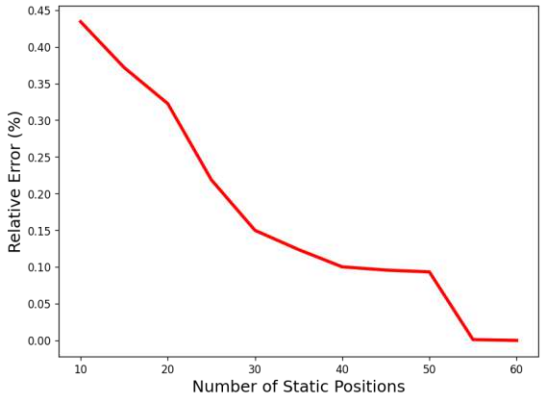
As can be seen in Figure 7.1, it takes a minimum of 30 static position to get a relative error equal or below 0.25% for the accelerometer calibration parameter vector θ^{acc} and a minimum of 45 static position to get a relative error equal or below 5% for the gyroscope calibration parameter vector θ^{gyro}. Therefore, I conclude that a minimum of 45 static positions are required to get a good estimate for θ^{acc} and θ^{gyro}. However, it is clear from the results that the more static positions are used for the parameter estimation, the more accurate results will be.

7.1.2 Effectiveness of estimated calibration parameter

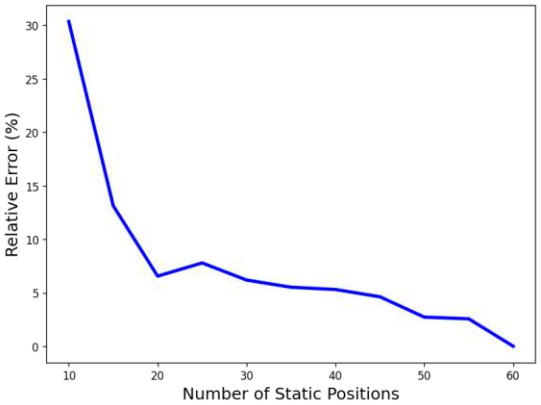
Next, I will evaluate the effectiveness of the calibration parameters on the recorded IMU data. As mentioned in 4.3.1, when an accelerometer is put in a static state, the magnitude of the outputted acceleration vector should be ideally equal to the magnitude of the gravity vector which is 9.81. Therefore, in order to evaluate the effectiveness of the calibration parameters for the accelerometer, I corrected all the acceleration vectors that were recorded while the IMU was in a static state using θ^{acc} and compared their magnitude to the magnitude of the gravity vector. For the comparison, I calculated the mean absolute error (MAE) between all the corrected results and 9.81. The MAE can be calculated by

$$e_{mae} = \frac{\sum_{i=1}^{N} ||\hat{x}_i - x_i||}{N} \text{[WL18]}. \tag{7.2}$$

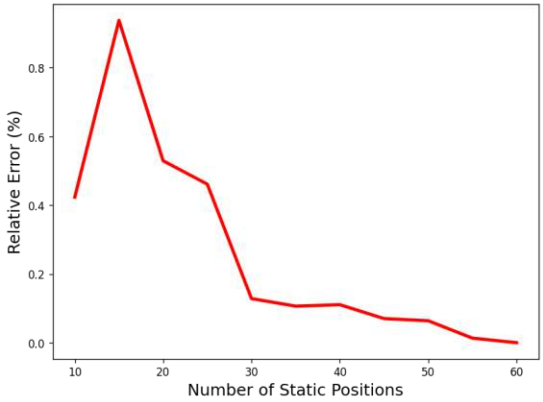
In my case, $\hat{x}$ corresponds to a corrected gravity vector in a static position while x is equal to the vector [0 0 9.81]. N is the number of gravity vectors that were corrected using θ^{acc}. I again performed this evaluation process for three different IMUs and the results are presented in Table 7.1, where the MAE is calculated for the original gravity vectors and the corrected ones using the estimated θ^{acc}. The results are further illustrated in Figure 7.2.



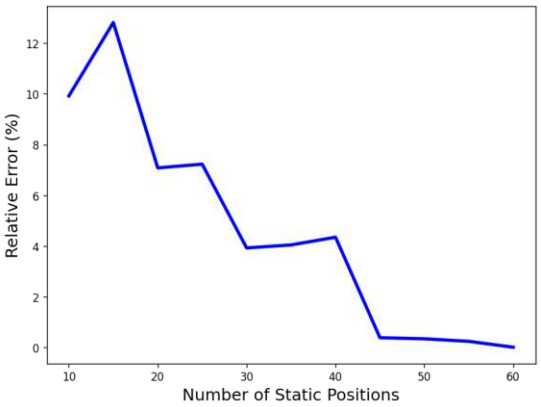
(a) **IMU 1**: Results for accelerometer calibration parameters θ^{acc}



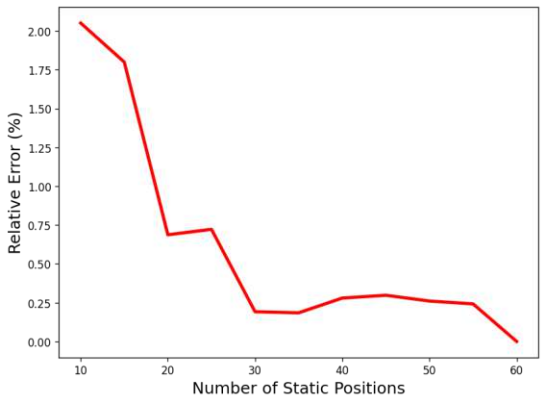
(b) **IMU 1**: Results for gyroscope calibration parameters θ^{gyro}



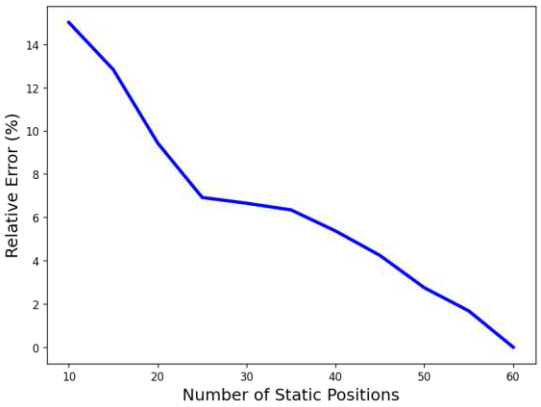
(c) **IMU 2**: Results for accelerometer calibration parameters θ^{acc}



(d) **IMU 2**: Results for gyroscope calibration parameters θ^{gyro}

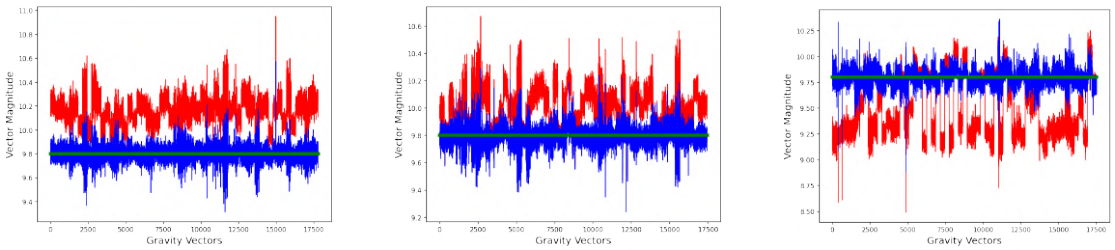


(e) **IMU 3**: Results for accelerometer calibration parameters θ^{acc}



(f) **IMU 3**: Results for gyroscope calibration parameters θ^{gyro}

Figure 7.1: Shows the number of static positions required to achieve a good estimate for the calibration parameters θ^{acc} (left) and θ^{gyro} (right). When computing the relative error, the calibration vector solution calculated from 60 static positions is used as a ground truth.



(a) IMU 1 (b) IMU 2 (c) IMU 3

Figure 7.2: Accelerometer calibration results for three different IMUs. The green line is the magnitude of the real gravity vector which is 9.81. The red line represents the magnitude of the vectors obtained from the accelerometer while in a static position. The blue line represents the magnitude of the vectors obtained from the accelerometer while in a static position that were corrected using the calibration parameters θ^{acc}.

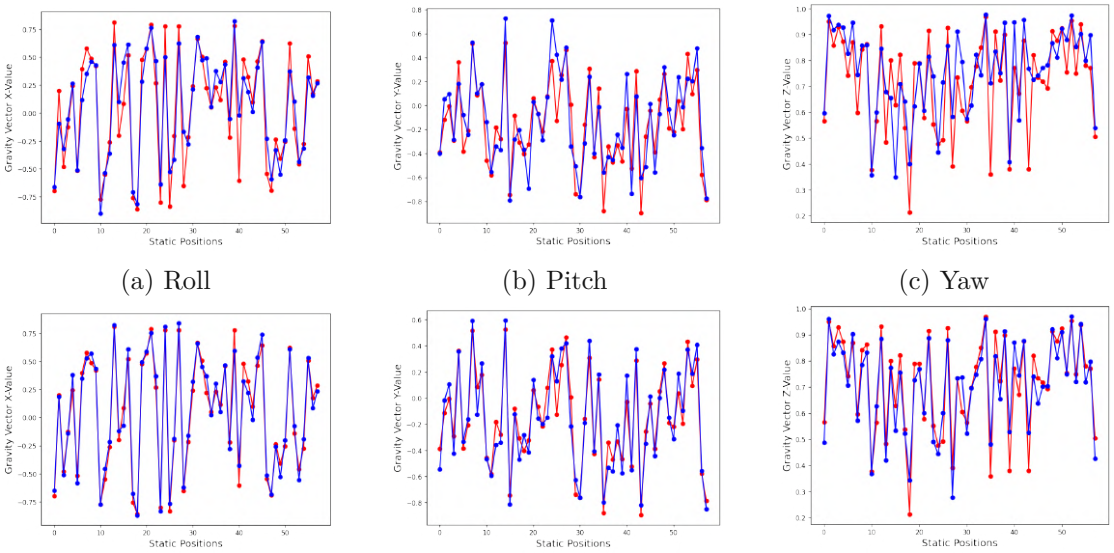
IMU 1	
	MAE
Roll	0.163
Pitch	0.121
Yaw	0.106
Roll (corr.)	0.062
Pitch (corr.)	0.056
Yaw (corr.)	0.053

IMU 2	
	MAE
Roll	0.136
Pitch	0.139
Yaw	0.093
Roll (corr.)	0.062
Pitch (corr.)	0.091
Yaw (corr.)	0.050

IMU 3	
	MAE
Roll	0.189
Pitch	0.150
Yaw	0.122
Roll (corr.)	0.071
Pitch (corr.)	0.087
Yaw (corr.)	0.070

Table 7.2: Shows the mean absolute error between the estimated static positions calculated using the gyroscope values and real static positions for each individual axis. The top half of each table shows the error calculated without the calibration parameters θ^{gyro} while the bottom half of each table shows the error calculated using gyroscope data corrected with the calibration parameters θ^{gyro}. Each table represents the results for one IMU.

In order to evaluate the effectiveness of the calibration parameters for the gyroscope, I utilized all the averaged static positions that were corrected using θ^{acc} and all the angular velocity vectors that were collected between each static position. For each corrected gravity vector, I performed angular velocity integration using the fourth-order Runge-Kutta method and calculated the subsequent static position. This provides me with an array of values that contains the estimated static positions $\hat{x}$ which I can then compare with the original array of values containing the actual static positions x. I again calculated the MAE using these two arrays and the results are presented in Table 7.2 and Figure 7.3.



(a) Roll (b) Pitch (c) Yaw

(d) Roll (corr.) (e) Pitch (corr.) (f) Yaw (corr.)

Figure 7.3: Shows the difference between the estimated static positions calculated using the gyroscope values (blue filled circles) and real static positions (red filled circles). The values are presented for each individual axes where (a) - (c) show the estimated static positions for the x, y and z angles before employing the calibration parameters θ^{gyro}. (d) - (f) show the estimated static positions for x, y and z angles using the calibration parameters θ^{gyro}.

Discussion of results

As can be seen by the results, applying the estimated θ^{acc} and θ^{gyro} to the gravity vectors and angular velocity vectors, respectively, clearly reduced the error rates. For the accelerometer, it can be visually seen in Figure 7.2 where the magnitudes of the corrected gravity vectors (blue curve) are much closer to the actual magnitude of the gravity vector which is 9.81 (green line) than the original gravity vectors (red curve). For the gyroscope, it can be seen in Figure 7.3 that the estimated static positions after the correction using θ^{gyro} (blue filled circles) are much closer to the real static positions (red filled circles) than before the correction. Therefore, I conclude that the estimated calibration parameters θ^{acc} and θ^{gyro} using 60 static positions were effective in reducing the errors of the distorted measurements of an IMU.

7.2 Attitude Estimation

For the attitude estimation evaluation, I will first focus on the yaw angle accuracy before comparing my motion tracker to the commercially available HTC Vive tracker. Then, I will give a qualitative evaluation of my proposed full-body motion capture solution in Unity.

7.2.1 Yaw angle accuracy

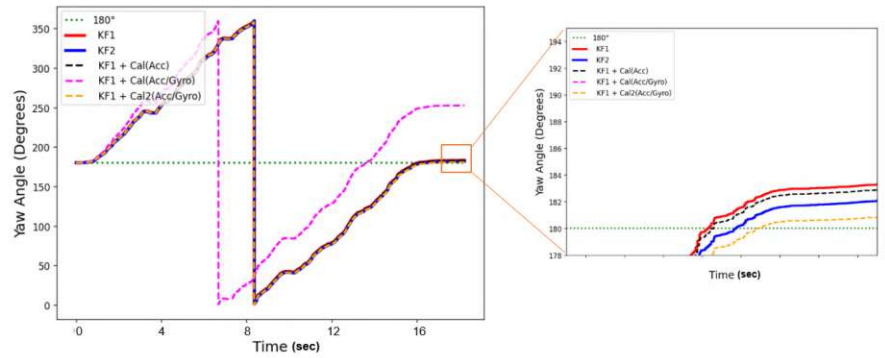
Since I am not using a magnetometer sensor in my motion capture system, the estimated yaw angle will suffer the most from sensor errors and drifts. Therefore, during my evaluation process, I focused first on determining the accuracy of the estimated yaw angle using my sensor fusion algorithms.
In order to perform the evaluation for the yaw angle estimation, I placed the motion tracker on an even surface and with its acrylic glass side against a wall and then rotated it by 360 degrees before placing the tracker on the even surface and against the wall again. This ensures that the final and end orientation of the motion tracker are identical. Using these two reference points, I can compare the calculated yaw angle after the full rotation with its initial yaw angle which ideally should be identical. Since I am only comparing the last recorded value, I used the absolute error (AE) which is simply $e_{ae} = |\hat{x} - x|$ [Bin16]. $\hat{x}$ is the estimated yaw angle and x is the true yaw angle value.
The results can greatly vary for slow and fast rotations and I have, therefore, performed two sets of experiments:

1. Very slow full rotations about the z-axis that took 17-21 seconds
2. Very fast full rotations about the z-axis that took 2.5-3.5 seconds

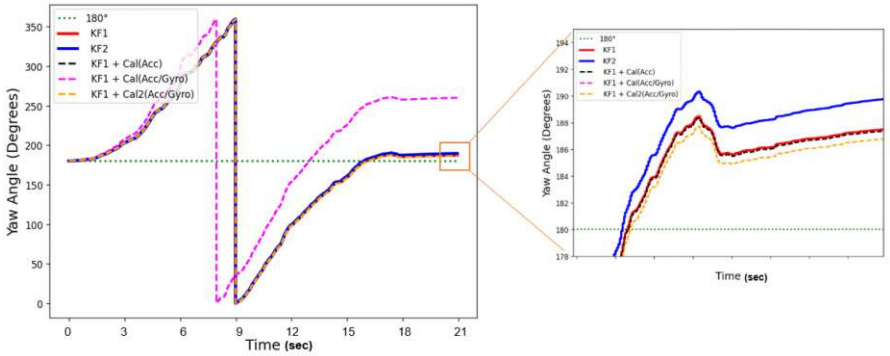
I performed these experiments for three different IMUs and, for each IMU, I conducted three trails for each slow and fast rotation. For each IMU, I have averaged the calculated absolute errors for all the trails and for each sensor fusion algorithm. The following sensor fusion methods were used during my experiments:

- **KF1** without correcting the IMU data using the calibration parameters
- **KF2** without correcting the IMU data using the calibration parameters
- KF1 where only the accelerometer data is corrected using θ^{acc} (I will refer to it as **KF1 + Cal(Acc)**)
- KF1 where both the accelerometer and gyroscope data are corrected using θ^{acc} and θ^{gyro} (I will refer to it as **KF1 + Cal(Acc/Gyro)**)
- KF1 where both the accelerometer and gyroscope data are corrected using θ^{acc} and a modified θ^{gyro} with its scaling parameters for all axes set to one (I will refer to it as **KF1 + Cal2(Acc/Gyro)**)

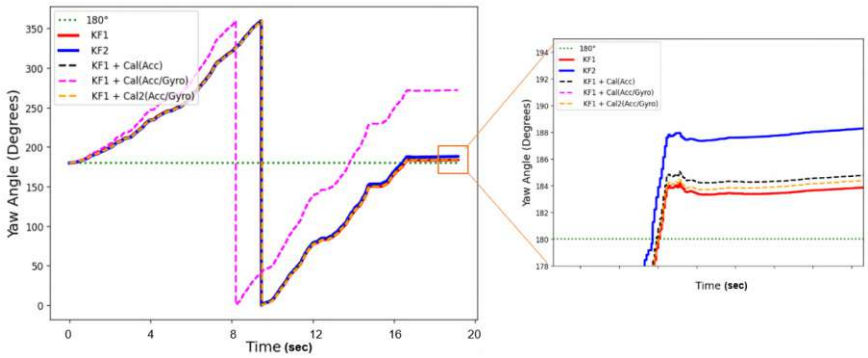
All of these described sensor fusion methods were applied simultaneously on the IMU data using the motion capture application running on the receiving computer.
For the slow motion tracker rotations, the results are presented in Table 7.3 and Figure 7.4. For the fast motion tracker rotations, the results are presented in Table 7.4 and Figure 7.5.



(a) IMU 1

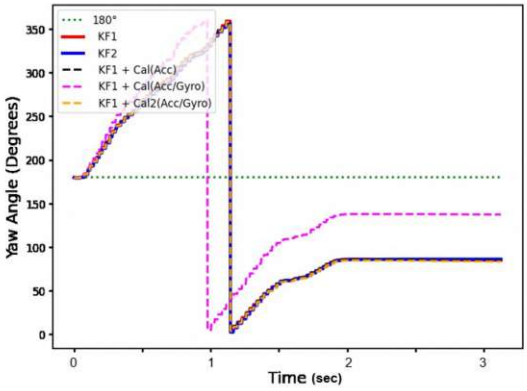


(b) IMU 2

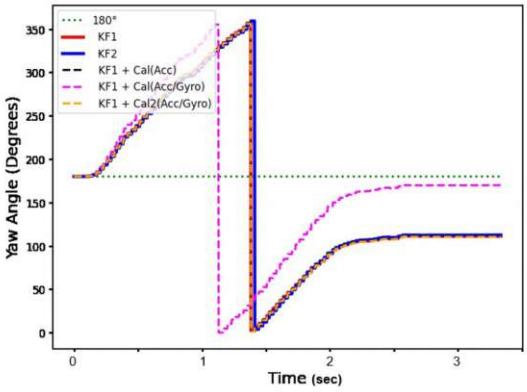


(c) IMU 3

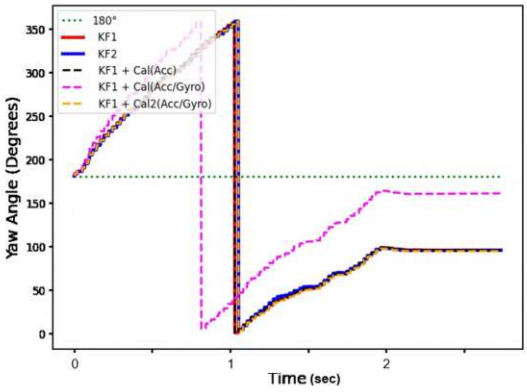
Figure 7.4: Shows the yaw angle changes of a motion tracker while it is being rotated about its z-axis by 360°. I rotated the motion tracker by hand very slowly, so that one full rotation took between 17-21 seconds. Normally, the motion tracker orientation has a yaw angle of 0° when it is in its initial position. However, for better visualization, I recentered the yaw angels around 180°. Each plot shows the result for a different IMU sensor.



(a) IMU 1



(b) IMU 2



(c) IMU 3

Figure 7.5: Shows the yaw angle changes of a motion tracker while it is being rotated about its z-axis by 360°. I rotated the motion tracker by hand very fast, so that one full rotation took between 2.5-3.5 seconds. Normally, the motion tracker orientation has a yaw angle of 0° when it is in its initial position. However, for better visualization, I recentered the yaw angles around 180°. Each plot shows the result for a different IMU sensor.

IMU 1	
Sensor Fusion Method	AE
KF1	4.434
KF2	3.546
KF1 + Cal(Acc)	2.900
KF1 + Cal(Acc/Gyro)	74.226
KF1 + Cal2(Acc/Gyro)	0.8292

IMU 2	
Sensor Fusion Method	AE
KF1	7.907
KF2	9.068
KF1 + Cal(Acc)	7.776
KF1 + Cal(Acc/Gyro)	80.073
KF1 + Cal2(Acc/Gyro)	6.760

IMU 3	
Sensor Fusion Method	AE
KF1	2.827
KF2	8.638
KF1 + Cal(Acc)	3.760
KF1 + Cal(Acc/Gyro)	91.497
KF1 + Cal2(Acc/Gyro)	3.720

Table 7.3: Shows the absolute error (measured in degrees) between the final yaw angle that was recorded after the completion of one full rotation of the motion tracker about its z-axis and the actual yaw angle value which is equal to the initial yaw angle value. Each rotation was conducted very slowly (between 17-21 seconds for the full rotation). Each table shows the results for one IMU where multiple sensor fusion methods are compared.

Discussion of results

When looking at the results for the slow motion tracker rotations, it is clear that after one full rotation there is a noticeable drifting error when looking at the final yaw angle. In two out of three cases, KF1 performs better than KF2 and also KF1 + Cal2(Acc/Gyro) performs better than KF1. The difference between KF1, KF1 + Cal(Acc) and KF1 + Cal2(Acc/Gyro) are, however, minimal as they range between ±2 degrees. A clear outlier are the results for KF1 + Cal(Acc/Gyro) where the AE can range between 74 and 92 degrees. This means that the estimated scaling parameters for the gyroscope are decreasing the accuracy of the estimated yaw angle. My theory is that during the data collection process of the calibration procedure, the *Mini-Robot* changes the orientation of its arm in fast and abrupt movements which is reflected in the collected angular velocities.

IMU 1	
Sensor Fusion Method	AE
KF1	99.377
KF2	100.892
KF1 + Cal(Acc)	99.642
KF1 + Cal(Acc/Gyro)	47.435
KF1 + Cal2(Acc/Gyro)	100.508

IMU 2	
Sensor Fusion Method	AE
KF1	77.309
KF2	76.707
KF1 + Cal(Acc)	77.416
KF1 + Cal(Acc/Gyro)	20.720
KF1 + Cal2(Acc/Gyro)	77.865

IMU 3	
Sensor Fusion Method	AE
KF1	99.577
KF2	95.030
KF1 + Cal(Acc)	99.481
KF1 + Cal(Acc/Gyro)	37.311
KF1 + Cal2(Acc/Gyro)	100.899

Table 7.4: Shows the absolute error (measured in degrees) between the final yaw angle that was recorded after the completion of one full rotation of the motion tracker about its z-axis and the actual yaw angle value which is equal to the initial yaw angle. Each rotation was conducted very fast (between 2.5-3.5 seconds for the full rotation). Each table shows the results for one IMU where multiple sensor fusion methods are compared.

This would make the estimated scaling parameters for the gyroscope only suitable for similarly fast rotations. This theory is confirmed when looking at the results for the fast motion tracker rotations. Without exceptions, KF1 + Cal(Acc/Gyro) performs better than the all the other methods. However, the AE ranges between 20 and 48 degrees and is, therefore, still a noticeable drifting error.

7.2.2 HTC Vive tracker comparison

Here, I will compare the attitude estimation results of my motion tracker to the commercially available HTC Vive tracker. The HTC Vive virtual reality system consists of a head-mounted display, controllers and laser emitters which are called lighthouses. This system uses the inside-out principle which does not rely on external cameras. Instead,

the lighthouses send out alternating vertical and horizontal laser sweeps which hit the photodiodes on the surface of the headset, controller and trackers. Based on the time difference at which the diodes are hit by the laser sweeps, the orientation and position of the tracked object can be calculated [NLL17].
I attached the HTC Vive tracker to my motion controller using tape as can be seen in Figure 7.6. In order to ensure that both the motion tracker and the Vive tracker have the first initial orientation, I save the initial quaternion gathered after the first orientation calculation for both the trackers and then apply the Equation 4.23. Similar to the previously described yaw angle evaluation, I performed two sets of experiments:

1. One very slow random rotation about the x-, y- and z-axes that took approximately 60 seconds
2. One very fast full rotation about the x-, y- and z-axes that took approximately 35 seconds

I conducted these experiments for two different IMUs and I calculated for each one the AE between the last recorded Euler angle values as well as the MAE using all the recorded Euler angle values of both trackers where the Vive tracker orientation values serve as the ground truth. I calculated these errors for multiple sensor fusion methods as described in 7.2.1 which were applied simultaneously on the IMU data using the motion capture application running on the receiving computer. For the slow motion tracker and Vive tracker rotations, the results are presented in Table 7.5 and Figure 7.7. For the fast motion tracker and Vive tracker rotations, the results are presented in Table 7.6 and Figure 7.8.

Discussion of results

As can be seen by the results and looking at the AE , the roll and pitch angles for both slow and fast rotations are very close to the estimated orientation of the Vive tracker. As observed in the evaluation for the yaw angle estimation, the results for KF2 are less accurate than KF1 and the yaw angles suffer from the same drifting issues. By looking at the calculate MAE , the convergence rate of the estimated orientation towards the true orientation (convergence rate of the Kalman filter) is conveyed. For slow rotations, the convergence rate of my implemented sensor fusion methods are acceptable which is reflected in the relatively low MAE . However, by looking at the results for the fast rotations, the MAE is higher which is expected as the convergence rate is not high enough. This means that, for fast rotations, there can be a visible delay until the estimated orientation of my motion tracker reaches its desired result.

7.2.3 Full-body motion capture quality

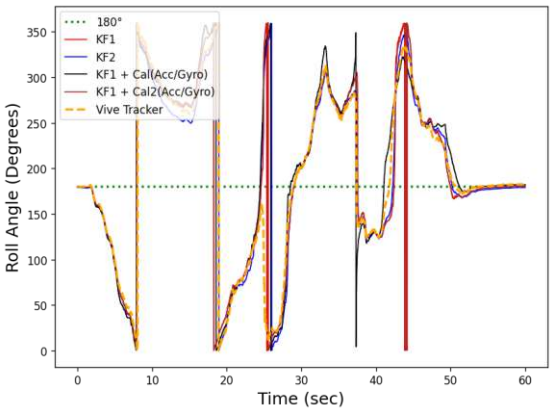
Finally, I will qualitatively evaluate the performance of my full-body motion capture suit. I conducted two experiments where, during the first one, the capture subject made slow

Figure 7.6: Shows the motion tracker attached to an HTC Vive tracker via tape.

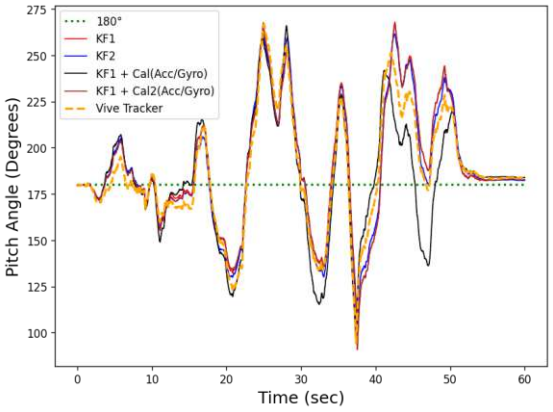
movements for approximately 60 seconds and, during the second one, the capture subject made fast movements for approximately 25 seconds. I used the sensor fusion method **KF1 + Cal2(Acc/Gyro)**. The result showing side-by-side comparisons between the capture subject and the virtual humanoid 3D character in Unity at different timestamps can be seen in Figure 7.9 and 7.10.

Discussion of results

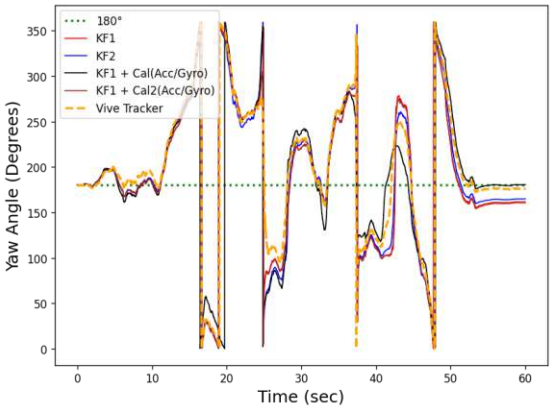
For slow body movements, my motion capture solution provides acceptable results. However, after a 360 degree full-body rotation by the capture subject (see Figure 7.9f-h), the yaw angle drifts are noticeably increased as can be seen by the slightly twisted right leg and pelvis.
For fast body movements, my motion capture solution also provides acceptable results in the beginning but again shows a decline in accuracy for the estimated yaw angles after the 360 degree full-body rotation by the capture subject (see Figure 7.10f-g). This time, the yaw angle drift is even more pronounced when looking at the legs, pelvis and head (see Figure 7.10g-h).
I also observed other weaknesses during the evaluation. If the capture subject moves into a crouching position or performs motions such as walking or jumping, these movements are not accurately reflected in the virtual environment. This is due to the lack of 3D position tracking in my implemented motion capture solution.



(a) Roll

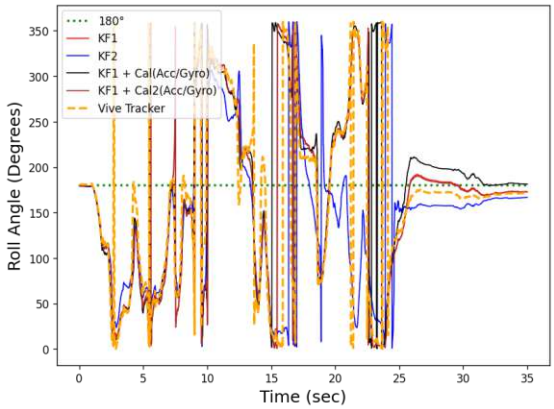


(b) Pitch

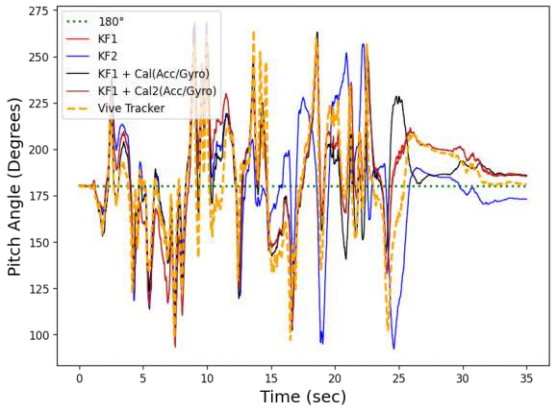


(c) Yaw

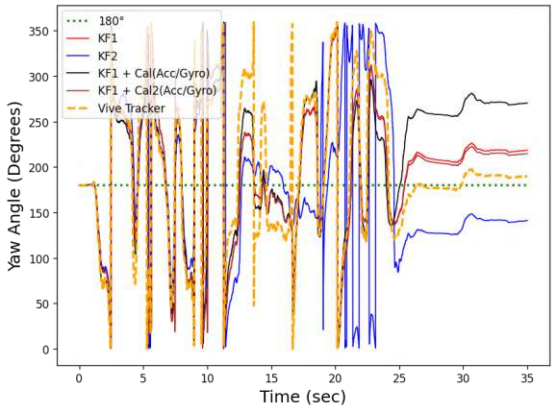
Figure 7.7: Shows the roll, pitch and yaw angle changes of a motion tracker while it is being randomly rotated about all its axes. I rotated the motion tracker by hand slowly for 60 seconds.



(a) Roll



(b) Pitch



(c) Yaw

Figure 7.8: Shows the roll, pitch and yaw angle changes of a motion tracker while it is being randomly rotated about all its axes. I rotated the motion tracker by hand very fast for 35 seconds.

(a) Initial pose

(b) Pose after 20 seconds

(c) Pose after 24 seconds

(d) Pose after 28 seconds

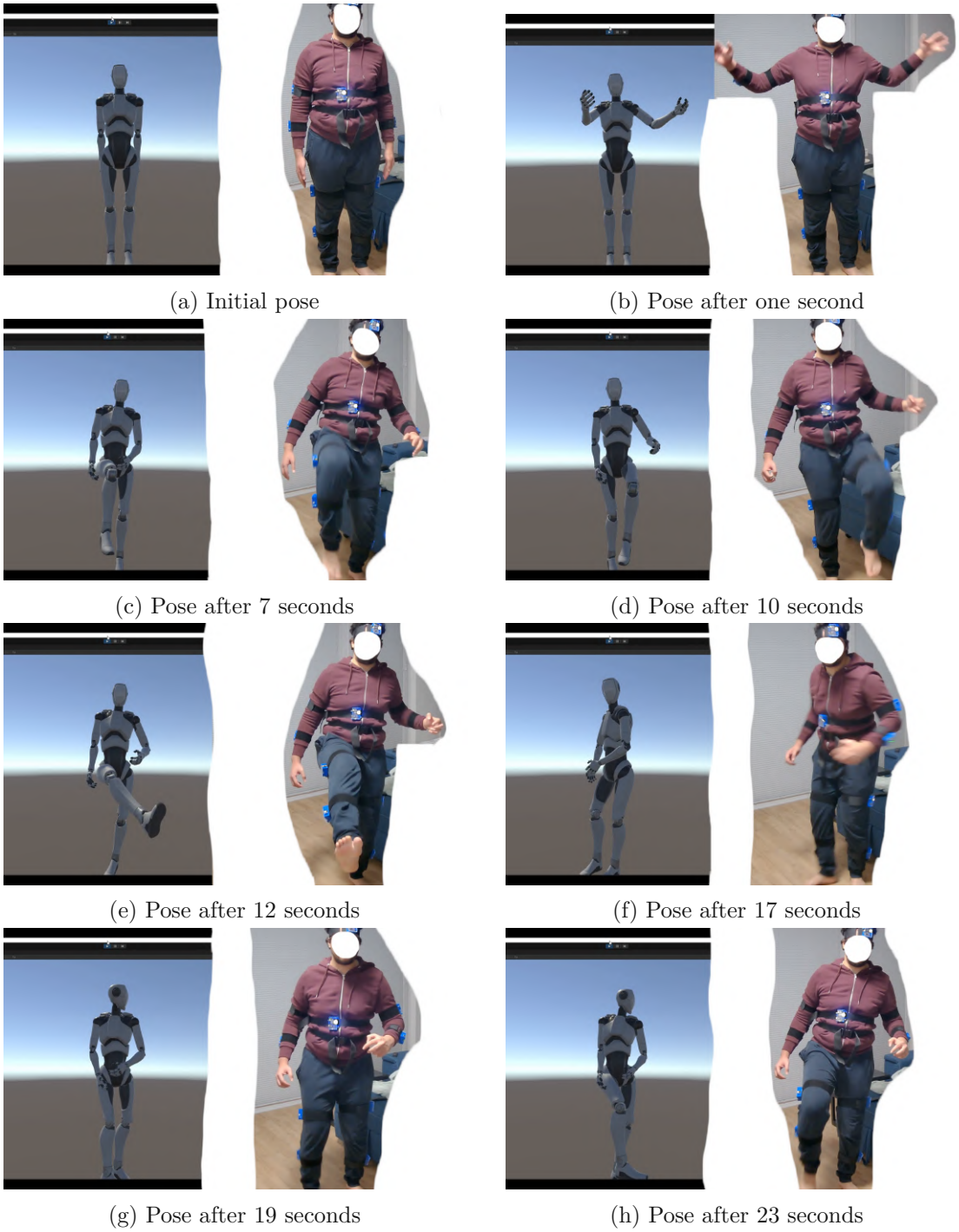
(e) Pose after 36 seconds

(f) Pose after 48 seconds

(g) Pose after 52 seconds

(h) Pose after 56 seconds

Figure 7.9: Shows a capture subject making slow body movements for 60 seconds as he is wearing my inertial motion capture suit. The real capture subject is shown on the right and the virtual humanoid 3D character in Unity is shown on the left side of the images.



(a) Initial pose

(b) Pose after one second

(c) Pose after 7 seconds

(d) Pose after 10 seconds

(e) Pose after 12 seconds

(f) Pose after 17 seconds

(g) Pose after 19 seconds

(h) Pose after 23 seconds

Figure 7.10: Shows a capture subject making fast body movements for 25 seconds as he is wearing my inertial motion capture suit. The real capture subject is shown on the right and the virtual humanoid 3D character in Unity is shown on the left side of the images.

IMU 1

Sensor Fusion Method	AE (final rot.)			MAE (all rot.)		
	Pitch	Yaw	Roll	Pitch	Yaw	Roll
KF1	1.066	22.904	3.076	7.044	14.662	16.637
KF2	1.334	18.665	4.627	5.973	12.178	14.161
KF1 + Cal(Acc/Gyro)	0.246	3.210	1.575	8.648	17.844	16.771
KF1 + Cal2(Acc/Gyro)	0.8702	21.939	3.151	6.856	14.935	15.809

IMU 2

Sensor Fusion Method	AE (final rot.)			MAE (all rot.)		
	Pitch	Yaw	Roll	Pitch	Yaw	Roll
KF1	3.160	23.508	1.012	8.212	24.627	33.1589
KF2	3.591	6.441	5.753	10.753	28.822	35.982
KF1 + Cal(Acc/Gyro)	5.202	53.732	7.206	9.070	31.211	32.590
KF1 + Cal2(Acc/Gyro)	3.870	9.765	4.379	9.158	21.344	32.497

Table 7.5: Shows the absolute error (measured in degrees) between the motion tracker's last recorded Euler angels and the Vive tracker's last recorded Euler angels after a slow 60 second random rotation. Additionally, the mean absolute error (measured in degrees) between all the recorded rotations between the two trackers for each of their axes are also shown. Each table presents the results for one IMU where multiple sensor fusion methods are compared.

IMU 1

Sensor Fusion Method	AE (final pos.)			MAE (curve)		
	Pitch	Yaw	Roll	Pitch	Yaw	Roll
KF1	5.026	37.726	7.775	13.733	31.638	27.746
KF2	7.792	39.734	14.118	21.086	66.407	43.891
KF1 + Cal(Acc/Gyro)	4.851	89.709	0.404	13.554	43.639	40.965
KF1 + Cal2(Acc/Gyro)	4.423	34.067	8.131	13.512	30.790	27.376

IMU 2

Sensor Fusion Method	AE (final pos.)			MAE (curve)		
	Pitch	Yaw	Roll	Pitch	Yaw	Roll
KF1	0.502	91.928	11.457	19.216	56.980	39.451
KF2	0.605	126.882	11.186	24.700	90.930	78.417
KF1 + Cal(Acc/Gyro)	0.2817	18.086	12.722	14.172	31.300	29.128
KF1 + Cal2(Acc/Gyro)	0.174	90.788	11.439	18.519	55.175	38.332

Table 7.6: Shows the absolute error (measured in degrees) between the motion tracker's last recorded Euler angels and the Vive tracker's last recorded Euler angels after a fast 35 second random rotation. Additionally, the mean absolute error (measured in degrees) between all the recorded rotations between the two trackers for each of their axes are also shown. Each table presents the results for one IMU where multiple sensor fusion methods are compared.

CHAPTER 8

Conclusion

In this chapter, I will give a short summary of my project and then discuss the limitations of my inertial motion capture system. Finally, I will discuss some future work with possible improvements that could increase the accuracy and usability of my system.

8.1 Summary

In this thesis, I proposed a low-cost and easy-to-use inertial motion capture system. I designed and assembled 11 motion trackers that are attached to the capture subject's head, left upper arm, left lower arm, right upper arm, right lower arm, chest, tailbone, left upper leg, left lower leg, right upper leg and right lower leg. The principal parts of each motion tracker are the ESP32 microcontroller, the 6 DoF IMU sensor MPU-6050 and a 3.7V 1100mAh rechargeable lithium battery enclosed in a 3D printed chassis. In order to assemble the motion trackers in a compact way, I designed and ordered custom PCBs. Furthermore, I introduced a certain degree of modularity to each motion tracker where the IMU and the battery can be detached and replaced if necessary.
Additionally, I designed and assembled a robot that can rotate its arm around the roll, pitch and yaw axes which is a crucial part of my implemented robot-assisted calibration procedure for each individual IMU. The goal of this calibration method is to minimize the errors of the accelerometer and gyroscope measurements in order to increase the accuracy of the estimated attitude of the motion tracker.
In order to correctly estimate the orientation of each motion tracker in 3D space, I implemented a sensor fusion method in the form of an Extended Kalman filter (*KF1*) which takes the angular velocities from the gyroscope as external input and the accelerometer data as measurements. I also implemented a second variation of the Kalman filter (*KF2*) based on my own experiments. Both Kalman filters are quaternion-based to avoid the singularities typically found in Euler angles parameterizations. In my evaluation, I found that *KF1* provides more accurate and more stable results than *KF2*.

I developed a configuration software for the motion tracker as a way to reprogram the tracker according to the user's needs. Possible configurations include the ability to change the network credentials, the assigned body part, the type of data being sent (Euler angles, quaternions or raw IMU data) or overriding the calibration parameters of the motion trackers.
The motion trackers send the IMU data via Wi-Fi using UDP to a receiving computer with a running motion capture application. I developed this motion capture application using Unity and it includes a humanoid, fully rigged 3D character mirroring the movements of the capture subject. Additionally, the motion tracker orientation data can also be retargeted to any 3D object in the virtual scene.

8.2 Limitations and future work

My proposed inertial motion capture solution has several limitations that could be improved in future iterations of this project.
As already discussed during the evaluation, my proposed calibration procedure could be improved my reprogramming the robot arm to decrease its rotation speed. Because the calibration parameters were estimated using data that was collected with a very high rotation speed of the robot arm, the corrected measurements became only suitable for fast rotation of the IMUs (reflected in the scaling values of the gyroscope calibration parameters). This led to a significant decrease in accuracy especially when estimating the yaw angles of the motion trackers during slow rotations. Therefore, I suggest improving the data collection process, by running multiple iterations of the recording process where the orientation changes are captured at multiple speed levels. For each of these iterations with a specified speed level, the calibration parameters are estimated. This way a lookup table with multiple scaling parameters assigned to different angular velocity magnitudes could be created where the scaling parameters for the gyroscope data could change at runtime based on the current magnitude of the angular velocity vector. That way the yaw angle estimations could become more accurate for both slow and fast rotations and higher convergence rate for all the angles could also be achieved.
A simpler approach for improving the estimated yaw angle accuracy is incorporating a magnetometer sensor in each motion tracker. However, this would lead to an increased cost of the tracking solution and also make it more susceptible to magnetic interferences.
A more difficult approach could be combining my inertial motion capture solution with a markerless optical motion capture solution, where a minimum of two webcams are added to the setup and record the pose of the capture subject using existing libraries such as *OpenPose* [CHS+19]. However, this would make my motion capture solution very computationally expensive which would require additional expensive hardware.
The most important improvement to the overall motion capture quality would be the addition of a kinematic model. As already explained in chapter 3, several existing inertial motion capture solutions rely on these kinematic models to improve the motion capture accuracy. Also adding rotation constrains to the rigged 3D character could prevent unnatural or unrealistic poses caused by possible sensor errors. I found that 11 motion

trackers are the minimum to achieve a good approximation of the entire body movements. Adding more motion trackers could improve the results but increase the overall cost of my proposed solution and additionally make the motion capture suit more uncomfortable due to the increased number of trackers attached to the capture subject's body.
A possible way to decrease the number of motion trackers is to use a virtual reality system with included finger tracking such as the Meta Quest [met] or VIVE Focus 3 [viv]. Not only can finger tracking be added to the motion capture application, the motion trackers attached to the lower arms can be eliminated and approximated using the optically tracked hand wrists and the motion trackers attached to the upper arms. Also, the motion tracker attached to the head would be eliminated and replaced with the virtual reality headset. The downside of this approach is that the hand must always be in view of the headset cameras which would limit certain possible movements of the capture subject. Another disadvantage is the added price for the virtual reality system.

Other limitations and possible future improvements include:

- In my implemented motion capture application, there is no correspondence between the size of the capture subject and the virtual 3D character. Introducing another calibration step before the starting the motion capture process where the user can set the height of the virtual character can be useful for adding additional realism and better correspondence between the tracked human and the animated character.
- Right now, I can reset the estimated angles if the sensor fusion algorithm is running on the receiving computer which can be helpful if the accumulated drifting errors in the estimated yaw angles become too noticeable. This is not possible when the sensor fusion algorithm is running on the microcontroller instead since I do not have a way to communicate and send messages to the ESP32. By implementing a bidirectional communication between the ESP32 and the receiving computer, I should be able to reset the motion trackers from the motion capture application.
- The chassis of the motion tracker attached to the capture subject's head can be improved. Due to its flat surface, it becomes uncomfortable to wear after a certain period of time. Designing a slightly more curved bottom surface for the chassis of the head tracker could make it more comfortable when it is strapped to the user's head.
- The design of the chassis could be improved in order to have a mechanism that would better hold the electronics inside of the chassis and prevent them from moving when they are subjected to fast movements. My current solution is to tape the battery and PCB to the ground of the chassis.

List of Figures

List of Tables

List of Algorithms

Acronyms

AE absolute error. 86, 89–91

DoF degrees of freedom. 33, 43, 99

IMU inertial measurement unit. 2, 10, 11, 13, 18, 32, 33, 35–39, 41–43, 45–50, 52–54, 57, 58, 60, 63–66, 69, 70, 72, 76, 77, 79, 81–84, 86–91, 97–100, 104–108

IMUs inertial measurement units. 1–3, 9, 27, 33–36, 45, 82, 84, 86, 91, 100, 106

LED light-emitting diode. 44, 63

LEDs light-emitting diodes. 6, 61

MAE mean absolute error. 82, 84, 91, 97, 98

MEMS Microelectromechanical systems. 10, 11, 45

PCB Printed Circuit Board. 2, 45, 62–64, 66, 101, 105

UDP User Datagram Protocol. 2, 41, 52, 57, 79, 100

Bibliography

[3dc23] Starter assets - third person character controller: Urp: Essentials. https://assetstore.unity.com/packages/essentials/starter-assets-third-person-character-controller-urp-196526, 2023. Accessed: (2023-10-10).

[AC13] Michael Andrle and John Crassidis. Geometric integration of quaternions. *Journal of Guidance Control Dynamics*, 36:1762–1767, 11 2013.

[BA12] Moti Ben-Ari. A tutorial on euler angles and quaternions. 2012.

[BFS19] Marek Babiuch, Petr Foltýnek, and Pavel Smutný. Using the esp32 microcontroller for data processing. In *2019 20th International Carpathian Control Conference (ICCC)*, pages 1–6, 2019.

[Bin16] David Bindel. Notes for matrix computations (cs 6210), 09 2016.

[Boy17] Michael Boyle. The integration of angular velocity. *Advances in Applied Clifford Algebras*, 27, 09 2017.

[BRRP97] Bobby Bodenheimer, Chuck Rose, Seth Rosenthal, and John Pella. The process of motion capture: Dealing with the data. In Daniel Thalmann and Michiel van de Panne, editors, *Computer Animation and Simulation '97*, pages 3–18, Vienna, 1997. Springer Vienna.

[CGD+19] F. Caputo, Alessandro Greco, Egidio D'Amato, Immacolata Notaro, and Stefania Spada. *IMU-Based Motion Capture Wearable System for Ergonomic Assessment in Industrial Environment*, pages 215–225. 01 2019.

[CHS+19] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[D+06] James Diebel et al. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35, 2006.

[Dad14] Majid Dadafshar. Accelerometer and gyroscopes sensors: operation, sensing, and applications. *Maxim Integrated [online]*, 2014.

[dfr23] How_to_use_a_three-axis_accelerometer_for_tilt_sensing. `https://wiki.dfrobot.com/How_to_Use_a_Three-Axis_Accelerometer_for_Tilt_Sensing`, 2023. Accessed: (2023-10-08).

[DGVRM+21] Gabriel Delgado-García, Jos Vanrenterghem, Emilio J Ruiz-Malagón, Pablo Molina-García, Javier Courel-Ibáñez, and Víctor Manuel Soto-Hermoso. Imu gyroscopes are a valid alternative to 3d optical motion capture system for angular kinematics analysis in tennis. *Proceedings of the Institution of Mechanical Engineers, Part P: Journal of Sports Engineering and Technology*, 235(1):3–12, 2021.

[Esp23] Espressif Systems. *ESP32 Series Datasheet v4.3*, 10 2023. Version 4.3.

[FLZ+17] Kaiqiang Feng, Jie Li, Xiaoming Zhang, Chong Shen, Yu Bi, Tao Zheng, and Jun Liu. A new quaternion-based kalman filter for real-time attitude estimation using the two-step geometrically-intuitive correction algorithm. *Sensors*, 17(9), 2017.

[fri] Fritzing. `https://fritzing.org/`. Accessed: (2023-10-11).

[Gav19] Henri P Gavin. The levenberg-marquardt algorithm for nonlinear least squares curve-fitting problems. *Department of civil and environmental engineering, Duke University*, 19, 2019.

[Gra08] Basile Graf. Quaternions and dynamics. *arXiv: Dynamical Systems*, 2008.

[Hen77] David M. Henderson. Euler angles, quaternions, and transformation matrices for space shuttle analysis. 1977.

[HS20] Roland Hostettler and Simo Sarkka. Lecture notes on basics of sensor fusion, September 2020.

[Jia22] Yan-Bin Jia. Quaternions*. 2022.

[jlc] Fritzing. `https://jlcpcb.com/`. Accessed: (2023-10-11).

[KHS17] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. volume 11. 2017.

[Kle04] Rachel Kleinbauer. Kalman filtering implementation with matlab. 2004.

[Kui99] Jack B Kuipers. *Quaternions and rotation sequences : a primer with applications to orbits, aerospace, and virtual reality*. Princeton Univ. Pr., Princeton, NJ [u.a.], 1999.

[KW08] Midori Kitagawa and Brian Windsor. *MoCap for artists : workflow and techniques for motion capture*. Elsevier/Focal Press, Amsterdam ; Boston, 1st edition edition, 2008.

[LLW+22] Jie Li, Xiaofeng Liu, Zhelong Wang, Hongyu Zhao, Tingting Zhang, Sen Qiu, Xu Zhou, Huili Cai, Rongrong Ni, and Angelo Cangelosi. Real-time human motion capture based on wearable inertial sensor networks. *IEEE Internet of Things Journal*, 9(11):8953–8966, 2022.

[lm23] scipy.optimize.least_squares - scipy v1.11.3 manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html, 2023. Accessed: (2023-10-07).

[lux] luxorparts. *SG90 Micro Servo*.

[Man16] Manash Kumar Mandal. Serial communication with an arduino using c++ on windows. https://blog.manash.io/serial-communication-with-an-arduino-using-c-on-windows-d08710186498.f94efw74b, 2016. Accessed: (2023-10-11).

[Mat] Math.net numerics. https://numerics.mathdotnet.com/. Accessed: (2023-10-11).

[met] Getting started with hand tracking on meta quest headsets. https://www.meta.com/en-gb/help/quest/articles/headsets-and-accessories/controllers-and-hand-tracking/hand-tracking/. Accessed: (2023-11-02).

[MFPC+21] Cecilia Monoli, Juan Francisco Fuentez-Pérez, Nicola Cau, Paolo Capodaglio, Manuela Galli, and Jeffrey A. Tuhtan. Land and underwater gait analysis using wearable imu. *IEEE Sensors Journal*, 21(9):11192–11202, 2021.

[moc23a] Sony corporation - mocopi | about mocopi. https://www.sony.net/Products/mocopi-dev/en/documents/Home/Aboutmocopi.html, 2023. Accessed: (2023-09-19).

[moc23b] Sony mocopi 3d motion capture system | qmss1/uscx. https://electronics.sony.com/more/mocopi/all-mocopi/p/qmss1-uscx, 2023. Accessed: (2023-09-19).

[Mol18] Christopher Molthrop. A comparison of euler and runge-kutta methods. *EGR 244L, Duke University*, 2018.

[Nan] NanJing Top Power ASIC Corp. *TP4056 1A Standalone Linear Li-Ion Battery Charger with Thermal Regulation in SOP-8*.

[NLL17] Diederick Niehorster, Li Li, and Markus Lappe. The accuracy and precision of position and orientation tracking in the htc vive virtual reality system for scientific research. *i-Perception*, 8:204166951770820, 06 2017.

[Nog12] Pedro Alves Nogueira. Motion capture fundamentals a critical and comparative analysis on real-world applications. 2012.

[Ors] Sterling Orsten. linalg.h. `https://github.com/sgorsten/linalg`. Accessed: (2023-10-11).

[pns23a] Neuronmocap. `https://neuronmocap.com/pages/perception-neuron-studio-system`, 2023. Accessed: (2023-09-27).

[pns23b] Neuronmocap. `https://neuronmocap.com/pages/perception-neuron-3`, 2023. Accessed: (2023-09-27).

[pns23c] Neuronmocap. `https://neuronmocap.com/products/perception-neuron-3-body-kit`, 2023. Accessed: (2023-09-27).

[pns23d] Neuronmocap. `https://neuronmocap.com/pages/axis-studio`, 2023. Accessed: (2023-09-27).

[PTFV92] W.H. Press, S.A. Teukolsky, B.P. Flannery, and W.T. Vetterling. *Numerical Recipes in FORTRAN 77: Volume 1, Volume 1 of Fortran Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1992.

[Rah18] M Rahul. Review on motion capture technology. *Global journal of computer science and technology*, 18(1):22–26, 2018.

[Ran04] Ananth Ranganathan. The levenberg-marquardt algorithm. *Tutoral on LM algorithm*, 11(1):101–110, 2004.

[Roj03] Raul Rojas. The kalman filter. *Institute for informatik*, pages 128–132, 2003.

[Sab11] Angelo Sabatini. Kalman-filter-based orientation determination using inertial/magnetic sensors: Observability analysis and performance evaluation. *Sensors (Basel, Switzerland)*, 11:9182–206, 12 2011.

[ser23] Serialplot - realtime plotting software. `https://www.citationmachine.net/apa/cite-a-website/new`, 2023. Accessed: (2023-10-07).

[SFM+19] Ryan Sers, Steph Forrester, Esther Moss, Stephen Ward, Jianjia Ma, and Massimiliano Zecca. Validity of the perception neuron inertial motion capture system for upper body motion analysis. *Measurement*, 149:107024, 09 2019.

[SGB+18] Martin Schepers, Matteo Giuberti, Giovanni Bellusci, et al. Xsens mvn: Consistent tracking of human motion using inertial sensing. *Xsens Technol*, 1(8):1–8, 2018.

[Spe] Specification of Li-ion Polymer Battery 3.7V 10000 mAh. *AA Portable Power Corp.*

[SSM19] Santiago Silvestre, Jordi Salazar, and Jordi Marzo. Printed circuit board (pcb) design process and fabrication. *Czech Technical University of Prague, Faculty of electrical engineering*, page 45, 2019.

[Tai] Taiwan Semiconductor. *SR102 – SR115*. Version: G2104.

[TJS23] Jami Torki, Charles Joubert, and Ali Sari. Electrolytic capacitor: Properties and operation. *Journal of Energy Storage*, 58:106330, 2023.

[TN18] Tomas Thalmann and Hans Neuner. Tri-axial accelerometer calibration for leveling. *GeoPreVi 2018 - Geodesy for Smart Cities*, 2018.

[TPM14] David Tedaldi, Alberto Pretto, and Emanuele Menegatti. A robust and easy to implement method for imu calibration without external equipments. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3042–3049, 2014.

[Ver21] Lucia Vera. Tracking systems in virtual reality: which is the best choice? `https://3dcoil.grupopremo.com/blog/tracking-systems-virtual-reality-the-best-choice/`, 2021. Accessed: (2023-08-17).

[viv] Hand tracking. `https://www.vive.com/au/support/focus3/category_howto/hand-tracking.html`. Accessed: (2023-11-02).

[VP20] Rachel V Vitali and Noel C Perkins. Determining anatomical frames via inertial motion capture: A survey of methods. *Journal of Biomechanics*, 106:109832, 2020.

[WB01] Greg Welch and Gary Bishop. An introduction to the kalman filter. *Proc. Siggraph Course*, 2001.

[WL18] Weijie Wang and Yanmin Lu. Analysis of the mean absolute error (mae) and the root mean square error (rmse) in assessing rounding model. *IOP Conference Series: Materials Science and Engineering*, 324(1):012049, mar 2018.

[Woo07] Oliver J Woodman. An introduction to inertial navigation. Technical report, University of Cambridge, Computer Laboratory, 2007.

[WZS15] Li Wang, Zheng Zhang, and Ping Sun. Quaternion-based kalman filter for ahrs using an adaptive-step gradient descent algorithm. *International Journal of Advanced Robotic Systems*, 12:1, 09 2015.

[ZPW+22] Chengxu Zhou, Christopher Peers, Yuhui Wan, Robert Richardson, and Dimitrios Kanoulas. Teleman: Teleoperation for legged robot loco-manipulation using wearable imu-based motion capture, 2022.

[ZS11] Eva Zupan and Miran Saje. Integrating rotation from angular velocity. *Advances in Engineering Software*, 42(9):723–733, 2011.