Implementation and Simulation-based Evaluation of Visual-Inertial Marker Integration for an Optical Motion Capture System

Andy Brinkmeyer[1]

MSc. Computer Science

Supervisor: Simon Julier

Submission date: 11 September 2020

Abstract

High performance optical tracking of body-mounted markers, as used for motion capture or virtual reality applications, relies on measurements from multiple cameras to triangulate a position. If not enough measurements are available the estimation equations are underdetermined and no solution can be found. This problem can be minimised by using a high number of cameras but not completely eliminated. One proposed solution is to equip the markers with inertial measurement units (IMU). This kind of sensor does not rely on external visibility and can therefore provide measurements even if all cameras are occluded. But it also suffers from measurement drift that needs to be compensated. This is often achieved through fusing multiple sensor measurements.

In this work a pipeline for simulating visual-inertial motion capture measurements based on the *PhaseSpace Impulse X2E* motion capture system is developed and used to produce three distinct simulation scenarios. Also, a graphical model for fusing raw optical linear detector measurements with inertial data is designed, implemented and evaluated using two methods: a real-time capable fixed-lag smoother and batch optimisation. Camera-only triangulation is used as a baseline for comparison.

It could be shown that the visual-inertial system can improve estimates in simple to moderate kinematic environments but struggles when confronted with rapid human motion. It was also demonstrated that the system is able to fill in short gaps where no camera measurements are available. Though the quality of the solution drastically varies with complexity of the underlying trajectory and the duration of the camera blackout.

Acknowledgements

I would like to thank our whole team, Simon Julier, Sebastian Friston, Wolfgang Stuerzlinger and Chong Tang for the great collaboration throughout the project. The progress would not have been possible without all the interesting discussions and all the helpful input you provided.

Contents

Chapter 1

Introduction

Optical tracking is a method used for motion capture and virtual reality applications. Common commercial high performance systems like the *PhaseSpace Impulse X2E*[1] rely on body-mounted markers that are tracked by multiple cameras, a schematic view of such a system is shown in Figure 1.1. Given the measurements of a single marker from multiple camera perspectives allows triangulating its position in three dimensional space. But this is only possible if the marker is visible by at least a minimum number of cameras, two for cameras which produce two dimensional images and three for cameras that produce a one dimensional image (also called linear detectors). Below this threshold the marker position can not be recovered. This presents a problem for motion capture and virtual reality applications where occlusion of markers can occur due to other objects in the scene, specific body poses (e.g. crossing ones arms) or faulty cameras. The best case scenario for such a situation is that once the marker is visible again it continues being properly tracked. The worst case is that the tracking fails because the obscured marker can not be identified after becoming visible again and manual intervention is required. In both cases the marker position while occluded is lost. Interpolation approaches can be used as an attempt to recover the position but become more uncertain the longer the gap in the data is. The marker position simply can not be recovered without further information.

In fields like aerospace and robotics Inertial Measurement Units (IMU) are used for computing a bodies position and rotation at all times by measuring linear acceleration and angular velocity. Though, consumer grade and small form factor IMUs suffer from significant drift of their estimate caused by accumulation of integrated errors. Therefore, IMUs are usually used as just one part of a navigation solution consisting of several additional measurement sources like GPS or odometry. The measurements are then fused to provide a single estimate that is more reliable than the individual measurements by themselves. Additionally, this often allows estimating sensor biases or even to automatically calibrate the system while in use.

In this work the approach of inertial measurement fusion is adapted to work with a motion capture system. An IMU is added to each individual marker to measure its linear acceleration and angular velocity. A graphical model for fusing the inertial data with raw linear detector

[1] `https://https://www.phasespace.com/x2e-motion-capture/`

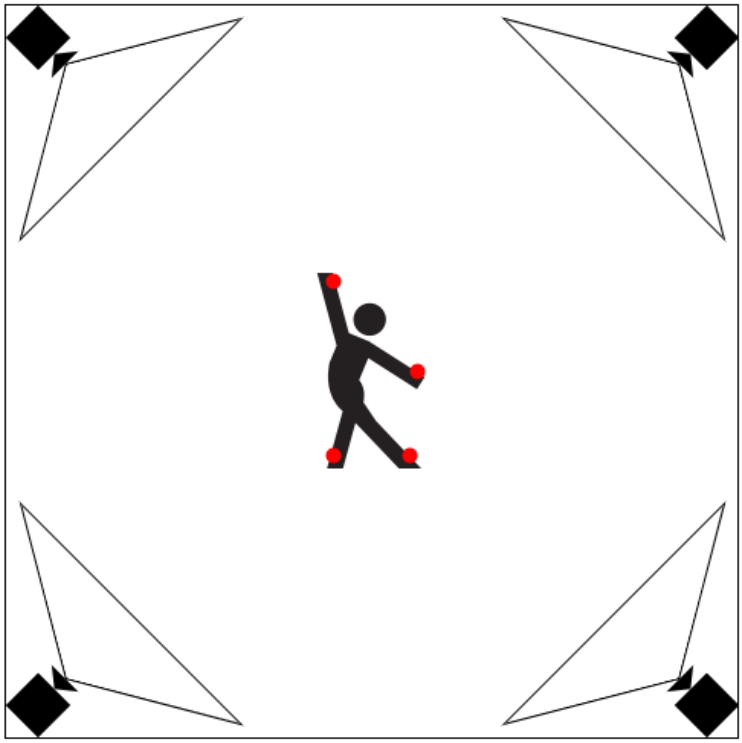

Figure 1.1: Schematic top-down view of a simple optical motion capture setup. The square outline are walls. The triangle visualise the camera field of views. The cameras are placed on the walls and are oriented to capture the interior of the room. Markers are placed on the humans body to track his movement, a small subset is shown as red dots. Only four cameras are displayed for readability but it is common to use eight or more [1].

measurements to estimate the markers position, rotation and velocity is defined. Also, a pipeline for simulating the modified motion capture system is developed for evaluating the performance of the graphical model compared to conventional camera-only triangulation.

This work starts with an introduction to motion capture technology and pose estimation in Chapter 2. The development of a simulation environment for generating artificial optical motion capture measurements is outlined in Chapter 3. Next, Chapter 4 presents a graphical model for pose estimation in optical motion capture systems. The models performance is evaluated in Chapter 5. Finally, the work is concluded and interesting directions for further research are presented in Chapter 6.

Chapter 2

Background

This chapter provides an overview of the fundamental fields and technologies this thesis builds upon. First, a short introduction to *Motion Capture Systems* is given in Section 2.1, in particular to the *PhaseSpace Impulse X2E* system which is used in this project. In Section 2.2 *Inertial Navigation* is covered. Then, an overview of *IMU Preintegration*, is provided in Section 2.3. Finally, Section 2.4 covers *Pose Estimation* and the common mathematical frameworks used for this task.

2.1 Motion Capture Systems

The first subsection of this section covers a short overview of motion capture. The second section goes into more detail about the motion capture system used for this thesis.

2.1.1 Overview and Types of Motion Capture Systems

Motion Capture is a broad field but is in general considered to be the process of recording real movement data from humans, animals or objects. The recorded data is often used to derive a mathematical representation that better describes the movement, e.g. rigid body poses (the rotation and position). Note that Motion Capture simply refers to the technology used for recording the data and bringing it into a suitable representation. Tasks like using the recorded data to animate virtual characters are not considered to be a part of Motion Capture [1].

Motion capture systems can be clustered into three groups [1]:

- **Outside-in System**: External sensors are used to measure sources on the captured objects, e.g. cameras measuring the position of an LED marker in their image space.
- **Inside-out System**: Sensors placed on the captured object measure external sources,

e.g. electromagnetic sensors measuring an external field.

- **Inside-in System**: Sensors placed on the captured object measure intrinsic properties, e.g. inertial measurement units measuring linear acceleration and angular velocity.

The major advantage of outside-in systems, specifically optical tracking systems, is that there are usually less stricter size, weight and power constraints for the sensors since they are not placed on the tracked object. This not only allows using a higher number of sensors but also using more powerful ones like cameras with high frame rates and resolutions. But this usually requires fixed and controlled capture setups due to the size and number of required components. Also, these systems often require extensive post-processing to produce the best motion solution [1]. Also, with camera-based systems visibility of the captured object is required at all times. Inside-out systems like electromagnetic tracking do not suffer from occlusion but are more limited in the tracking performance they provide due to stricter size, weight and power requirements of body-mounted sensors. They are generally cheaper [1] and provide real-time feedback. The last category, inside-in systems also have the advantage of not suffering from occlusion but are again limited by the stricter hardware requirements. Also, inertial motion tracking systems have to deal with drifting estimates and are therefore often limited to only providing constraints on joints instead of computing a global position. But they are less expensive than optical systems and portable [1].

For the reasons mentioned above optical outside-in systems are preferred for motion capture tasks where precision is most important. Also, modern systems like the *PhaseSpace Impulse X2E* provide real-time capability for immediate feedback. Though the best motion solution can still only be obtained by post-processing.

The system considered in this thesis, the *PhaseSpace Impulse X2E*, is a LED marker based capture system using optical linear detectors to estimate marker positions. In addition to the markers, inertial measurement units are also attached to the captured object, though using their measurements for better estimation is still in development and not yet implemented by the manufacturer. Therefore, the system is a combination of the first and last category. The PhaseSpace X2E motion capture system is described in greater detail in the following subsection.

2.1.2 PhaseSpace Motion Capture System

The PhaseSpace Impulse X2E is a high performance marker-based motion capture system developed by the American company PhaseSpace. It consists of three major components: markers, cameras and a core server. A schematic view of the system is given in Figure 2.1.

Markers

Active LED markers are used for tracking. This means that in contrast to passive systems the individual markers can be controlled, e.g. they can be turned on and off in single frames. This

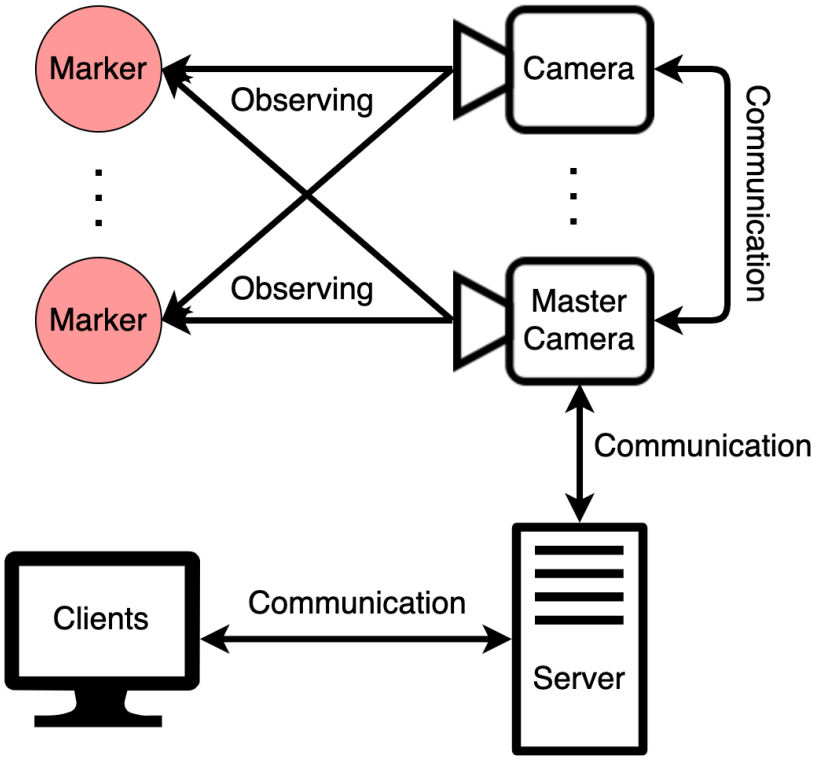

Figure 2.1: Schmematic view of the *PhaseSpace Impulse X2E* optical motion capture system.

allows the motion capture system to uniquely identify every marker throughout the whole capture session. The markers are controlled by microdrivers, small battery driven microcontrollers that communicate via an RF module with a base station. Each microdriver can control several individual LEDs.

Cameras

The Impulse X2E system uses optical linear detectors instead of classical cameras. Each camera module consists of two linear detectors slightly offset and rotated against each other in the camera plane. One of the camera modules acts as a master camera (by PhaseSpace also referred to as Base Station) to which the other cameras and the LED microdrivers are connected. This base station handles the communication with the core server.

Core Server

The core server is a Linux computer that has two main tasks, handling the communication with the other Impulse X2E devices and clients, and performing the tracking. The base station is connected to the server via USB while external clients can access the exposed API through a network socket. The core software running on the server handles the raw data processing and the computation of position estimates for markers or pose estimates for rigid bodies.

2.2 Inertial Navigation

Classical mechanics allow to compute the position and rotation (also called pose) of a rigid body given its acceleration and angular velocity at every point in time, assuming its initial state (velocity, position and rotation) are known. In real world applications we can measure the acceleration and angular velocity of a body using an Inertial Measurement Unit (IMU) at discrete time steps and numerically integrate the data to dead-reckon the current velocity, position and rotation from an initial state. This process is called Inertial Navigation and is often used as one component of more complex navigation systems [2, 3]. The IMU measures the acceleration $\mathbf{a}^b$ and angular velocity $\boldsymbol{\omega}^b$ in a body-fixed frame. This frame is often aligned with the IMU itself to ease computations. The position and rotation are instead defined in a local frame. Such a local frame stays fixed in the world, e.g. fixed with respect to the earth. A common local frames used in aerospace is the NED frame which has his x-axis pointing to earths geographic north pole, its y-axis in the geographic east direction and the z-axis to earths centre. Now, given an inertial measurement the new position $\mathbf{p}_{i+1}$, velocity $\mathbf{v}_{i+1}$ and rotation $\mathbf{R}_{i+1}$ can be computed by the set of equations

$$\mathbf{R}_{i+1} = \mathbf{R}_i Exp(\boldsymbol{\omega}_i^b \Delta t) \tag{2.1}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{g}\Delta t + \mathbf{R}_i(\mathbf{a}_i^b \Delta t) = \mathbf{v}_i + \mathbf{g}\Delta t + \mathbf{a}_i \Delta t \tag{2.2}$$

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{g}\Delta t^2 + \frac{1}{2}\mathbf{a}_i \Delta t^2 \tag{2.3}$$

where variables without superscript denote measurements in the local frame. The function $Exp(\boldsymbol{\omega}_i^b \Delta t)$ is the exponential map that maps an angular velocity to a proper rotation (a more detailed explanation is provided in Section 2.4.2). $\mathbf{g}$ is the gravity vector and is added to the equations to compensate for the gravity measured by the IMU accelerometers. Δt is the time difference in-between two measurements. Note that to update the position, velocity and rotation their previous values are needed. Usually multiple IMU measurements are integrated over time and the resulting state of applying the integration equations of the previous time step is used. Only the first measurement needs special treatment since no previous solution exists. The values used to initialise the integration when no previous state is available are called the initial values for integration or the initialisation point. Also, note that the equations are highly nonlinear due to rotating the acceleration vector. Furthermore, it becomes clear that a wrong initialisation, especially a wrong initial rotation, leads to improper compensation of the measured gravity and therefore to large errors in the updated velocity and position. Therefore, proper initialisation is tremendously important.

While inertial navigation has the advantage that sensors are mounted on the object and measure intrinsic properties, i.e. they do not depend on external sources, the navigation solution (velocity, position and rotation) suffers from drift. Those drifts are cumulative errors caused by the integration of sensor biases. Depending on the quality and calibration of the inertial sensors the computed navigation solution may only be usable for a few seconds after initialisation. Additional sensor measurements are necessary to constrain the drift using sensor fusion algorithms. Common additional sources are GPS data, odometry and visual data from monocular or stereo cameras, the latter combination is in literature often referred to as *Visual Inertial Navigation* (VIN) or *Visual Inertial Odometry* (VIO) [4–8]. The main advantages

of using IMUs as part of a navigation solution are their high operating rate, their ability to capture high dynamic motions and to temporarily constrain the navigation state when other measurements are not available [4].

In order to compute a navigation solution in three dimensional space an Inertial Measurement Unit needs to consist of at least one 3-axis accelerometer and one 3-axis gyroscope. Sometimes, a 3-axis magnetometer might also be part of the IMU. Consumer grade IMUs are usually based on MEMS (micro-electro-mechanical system) sensors [2]. This class of sensors will be used in the *PhaseSpace Impulse X2E* system and therefore considered in the rest of the thesis when inertial sensors are mentioned. Also, we will from now on only consider the three dimensional navigation case, i.e. a body moving in three dimensional space.

The representation of velocity and position of a rigid body is straight forward, each is characterised by a single three dimensional real valued vector. Its rotation can also be uniquely defined by three independent variables, the Euler angles. While this rotation representation is minimal it is often computationally more convenient to use unit quaternions or rotation matrices which can be represented using four and nine values respectively, though not all are independent.

2.3 IMU Preintegration

IMUs usually operate at rates much higher than other sensors used for navigation (e.g. GPS or cameras). To use those high rate measurements in a filter or smoother, measurements in between keyframes (frames where additional measurements from lower rate sensors are available) must either be dropped or additional states must be added. This results in either a loss of information or significantly higher prediction rates. Another difficulty is obtaining proper initialisation of the inertial integration. The high non-linearity of the attitude equations can result in large errors or even non-convergence of the estimate in filters and smoothers when incorrect initialisation is used [4]. More precisely, an incorrect initial attitude leads to wrong compensation of the gravity vector and therefore large errors in the estimated body acceleration.

To avoid adding unnecessary states and to avoid committing to an initialisation point, inertial measurements can be integrated without initial conditions in between two poses to form a single pseudo inertial measurement as first proposed by [4]. This *preintegrated* measurement can then be used to constrain the movement in between two keyframes. Preintegration is especially useful in estimation methods that re-linearise previous states since the preintegrated measurement can easily be applied to new linearisation points. This avoids storing the individual IMU measurements and reintegrating them whenever re-linearisation is applied.

While the work of [4] and [5] pioneered the idea of preintegration, their method used Euler angles for representing rotations which suffer from singularities, and they applied Euclidean averaging and smoothing methods for state and uncertainty propagation. In [7] and [9] those shortcomings were addressed by taking the manifold structure of the rotation group $SO(3)$ into account. Furthermore, they presented an improved method of uncertainty propagation

and a-posteriori bias correction for IMU preintegration. Further advancements were presented in [8, 10] by deriving a closed form solution to the preintegration equations. Instead of using discrete integration as the previous works they present two analytical models: piecewise constant measurements and piecewise constant local true acceleration.

2.4 Pose Estimation

Below the fundamentals of estimating a pose given multiple measurements are discussed. First a quick introduction to filtering and smoothing is given. Then, the special structure of the group of poses and the modified optimisation machinery they require is covered. Finally, factor graphs and their use for modelling the pose estimation problem are discussed. The theory outlined in the following three sections is based on [11] which gives a great introduction to factor graphs and their use for inference in robotics.

2.4.1 Filtering and Smoothing

Often more than one sensor is used to measure the pose of an object to produce a more reliable estimate. The process of fusing those measurement to obtain a single pose is called *Pose Estimation*. The historically best known approach for Pose estimation, specifically fusing inertial and GPS measurements, is the EKF (Extended Kalman Filter) [12]. While it is computationally fast for small state vectors and performs well for moderate nonlinear systems, the estimate can fail to converge for highly nonlinear dynamics and the computational complexity becomes intractable for high dimensional states [11]. Also, it suffers from accumulation of linearisation errors since all past estimates and their uncertainties are marginalised into a single state [5] and can not be re-linearised when the linearisation point changes, though this is disputed and it might rather be the case that the marginalised distribution is much less expressive than the true underlying joint distribution of all states and measurements. Smoothing approaches can be used to improve estimates of nonlinear systems. Past measurements are kept and the estimate is computed by optimising over all previous measurements and states, i.e. the MAP (Maximum A-Posteriori) estimate

$$X_{MAP} = \arg\max_{X} l\left(X; Z\right) p\left(X\right) \tag{2.4}$$

where $l\left(X; Z\right)$ is the likelihood of the states X given the measurements Z and $p\left(X\right)$ is the prior over the states X, is computed. Assuming zero-mean Gaussian distributed measurement noise leads to the nonlinear least-squares problem

$$X_{MAP} = \arg\min_{X} \sum_{i} \|h_i(X_i) - z_i\|^2_{\Sigma_i} \tag{2.5}$$

with the possibly nonlinear measurement function $h_i(X_i)$ and covariance Σ_i associated with the measurement z_i.

Nonlinear optimisation methods like Gauss-Newton or Levenberg-Marquardt can be used to

solve nonlinear least-squares problems. But they require the measurement function h_i to be linearised. This can be done using a first order Taylor-Expansion such that

$$h_i(X_i^0 + \Delta_i) \approx h_i(X_i^0) + \mathbf{J}_i \Delta_i \tag{2.6}$$

where Δ_i is a small increment around the linearisation point X_i^0 and $\mathbf{J}_i$ is the Jacobian of h_i. This linearised version can then be used to take small steps towards an optimal solution using gradient descent. Now, instead of optimising for X the least-squares problem 2.5 can be reformulated as

$$\Delta^* = \arg\min_{\Delta} \sum_i \left\| h_i(X_i^0) + \mathbf{J}_i \Delta_i - z_i \right\|^2_{\Sigma_i} \tag{2.7}$$

where Δ^* is the optimal linear increment for the current linearisation point. An optimal X can then be computed by repeated solving and applying the increments Δ^*. Given good initialisation points for the optimisation the global optimum can usually be recovered.

2.4.2 On-Manifold Optimisation

The linearisation strategy outlined in the previous section works great for variables that live in a vector space where vector addition is defined. But now consider a three dimensional pose $\mathbf{P} = (\mathbf{R}, \mathbf{t})$ ($\mathbf{R}$ is the rotation and $\mathbf{t}$ the position) and a small pose increment $\boldsymbol{\xi}$. How would one go about adding those two, more specifically how would we define the addition of two rotations. Vector-like addition clearly does not work in this case since adding two rotation matrices does not result in a third rotation matrix. The same holds for other representations of rotations as well. This is since the rotations in three dimensional space form a Lie Group, a differential manifold. This group is called $SO(3)$, Special Orthogonal Group, and forms together with the position (a three dimensional vector space) the six dimensional Lie Group $SE(3)$, Special Euclidean Group. For $SO(3)$ two elements are combined through matrix multiplication. For $SE(3)$ combining two poses $\mathbf{P}_1$ and $\mathbf{P}_2$ yields $\mathbf{P}_1 \oplus \mathbf{P}_2 = (\mathbf{R}_1\mathbf{R}_2,\ \mathbf{R}_1\mathbf{t}_2 + \mathbf{t}_1)$.

To optimise over $SE(3)$ we need to define a notion of an incremental change $\boldsymbol{\xi}$ around a pose $\mathbf{P}_0$. For this purpose we define the local coordinate vector

$$\boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \omega_x & \omega_y & \omega_z & v_x & v_y & v_z \end{bmatrix}^T \tag{2.8}$$

where $\boldsymbol{\omega}$ is an incremental rotation and $\mathbf{v}$ an incremental position change. Applying the hat[1] operator to $\boldsymbol{\xi}$ gives us

$$\boldsymbol{\xi}^\wedge = \left[\begin{array}{ccc|c} 0 & -\omega_z & \omega_y & v_x \\ \omega_z & 0 & -\omega_x & v_y \\ -\omega_y & \omega_x & 0 & v_z \\ \hline 0 & 0 & 0 & 0 \end{array}\right] \tag{2.9}$$

which is an element of the Lie Algebra of $SE(3)$, also referred to as $\mathfrak{se}(3)$. All vectors $\boldsymbol{\xi} \in \mathbb{R}^6$ form a vector space that is isomorphic to the 6 dimensional vector space defined by $\mathfrak{se}(3)$. This vector space captures the local structure of $SE(3)$ around its identity and is often called the

[1]The hat operator can be used to map an element of $\mathbb{R}^6$ to a corresponding element of $\mathfrak{se}(3)$, i.e. $\boldsymbol{\xi}^\wedge : \mathbb{R}^6 \mapsto \mathfrak{se}(3)$.

tangent space. An element $\boldsymbol{\xi}^\wedge$ of the tangent space can be mapped back on to the manifold through the exponential map $e^{\boldsymbol{\xi}^\wedge} : \mathfrak{se}(3) \mapsto SE(3)$. We will use $\boldsymbol{\xi}$ and $\boldsymbol{\xi}^\wedge$ interchangeably for better readability.

Now that we have a vector space of incremental pose changes we can adapt Equation 2.6 to work with the manifold structure of $SE(3)$. A Jacobian $\mathbf{J}_i$ needs to be found that satisfies

$$h_i\left(P_0 e^{\xi}\right) \approx h_i(P_0) + \mathbf{J}_i \boldsymbol{\xi} \tag{2.10}$$

where vector addition was substituted by the appropriate binary group action. In Equation 2.7 instead of Δ_i the increments in the tangent space $\boldsymbol{\xi}_i$ will be used. The estimation algorithms also need to be modified to use the proper group operation and exponential map when applying increments to poses.

2.4.3 Factor Graphs for Inference

Factor graphs were found to be an efficient structure for modelling the factorised joint density of MAP estimation. They are bipartite graphs that represent the factorisation of a function through two types of nodes, factors and variables. If a factor is a function of a variable they are directly connected through an edge. An example of a simple chain-like factor graph compared to a Kalman Filter is shown in Figure 2.2. It becomes obvious that the factor graph stores the entire joint distribution of the inference problem while the Kalman Filter condenses the previous states and measurements into a single prior. For MAP estimation the factors are

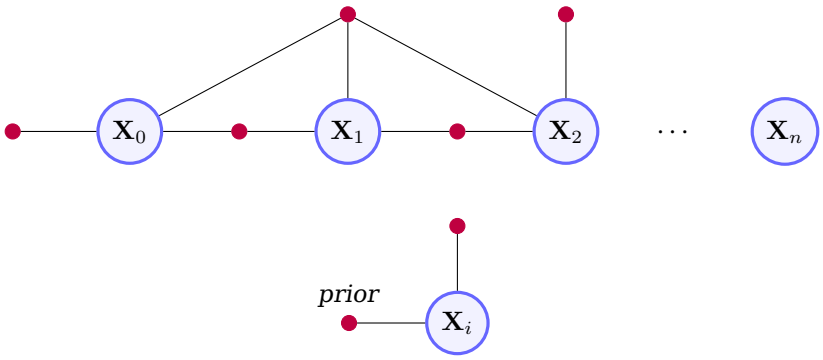

Figure 2.2: Temporal structure of a chain-like factor graph (top) and a Kalman Filter (bottom). The blue circles are variables and the purple dots denote factors. Note that the Kalman Filter reduces all previous variables into a single prior. The other factor acting upon $\mathbf{X}_i$ defines the current set of measurements.

the likelihood functions $l_i(X_i; Z_i)$ and the priors $p_i(X_i)$. The factor graph then represents the factorised function

$$\Phi(X) = \prod_i l_i(X_i; Z_i) \cdot \prod_i p_i(X_i) \tag{2.11}$$

of all likelihoods and priors. Note that $\Phi(X)$ is only a function of the variables X and not of the measurements Z since the latter are fixed. Assuming zero-mean Gaussian measurement noise leads again to the MAP problem in Equation 2.5.

Factor graphs can accommodate both linear and non-linear functions, conserve the sparsity of the problem (if it is sparse in the first place) and can efficiently be converted to a Bayes Net for performing inference.

The MAP estimate of a factor graph based smoothing problem can then be computed using nonlinear batch optimisation. But often measurements arrive sequentially and performing batch optimisation each time a new measurement is available is computationally expensive. An efficient approach called iSAM (Incremental Smoothing and Mapping) for reusing and updating previous computations, i.e. the matrix factorisation, was first presented in [13]. ISAM2 improves upon this method by introducing a novel data structure, the Bayes Tree, that better relates to the underlying structure of the sparse matrix factorisation and allows for more efficient computations [14].

Chapter 3

Simulating Motion Capture Data

This chapter describes the approach to simulating both the measurements of the motion capture system and the measurements of the IMUs. Section 3.1 discuss the motivation behind simulating motion capture data. The high-level structure of the simulation pipeline is described Section 3.2. Section 3.3 covers simulating rigid body systems using a game engine followed by simulation of IMU measurments in Section 3.4. The simulation of linear detectors is covered in Section 3.5 followed by the derivation of the estimation equations for triangulation in Section 3.6.

3.1 Motivation: The advantages of simulating motion capture and IMU data

It would be a naive approach to just record data in the motion capture system with IMUs attached and compare the raw motion capture estimates with the fused estimates. It is impossible to tell which method performs better since no baseline for comparison is available. Data is needed where the true positions of the markers are known. Recording such data in a motion capture system is difficult since determining the true position of a point in space is inherently a hard task. For this reasons a simulation pipeline was developed which can be used to generate marker trajectories, compute their corresponding IMU measurements and simulate the individual camera outputs. Using a simulation not only eliminates a lot of the preparation necessary for physical experiments but also gives tight control over the noise characteristics of the individual sensors.

An additional reason for simulating the motion capture system is the 2020 COVID-19 pandemic. Due to the nationwide lockdown it was not possible to get access to the capture system at the university.

3.2 Simulation Pipeline

Simulating the motion capture system consists of several steps as shown in Figure 3.1. First,

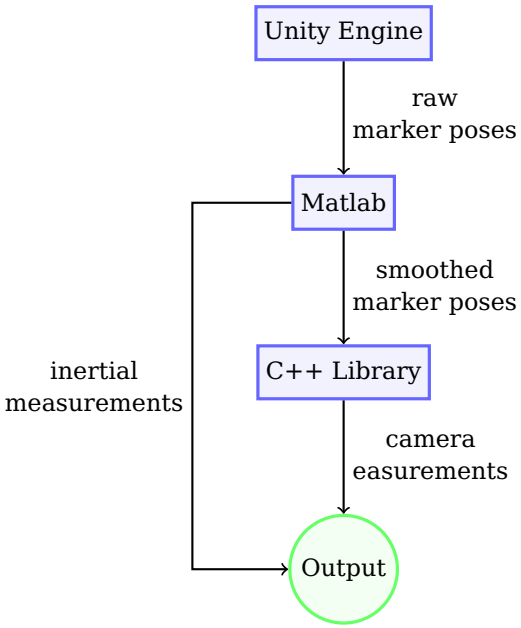

Figure 3.1: Visualisation of the simulation pipeline.

the tracked rigid bodies are simulated. This is done using methods applied in game development. The rigid bodies are animated which means their pose is defined at several key frames and the movement in-between is interpolated. Next, the markers need to be positioned on the tracked bodies. The game engine Unity is used for this task. When all markers have been placed the animation is played in the game engine and the marker trajectories are recorded using a custom developed framework. The raw position and rotation trajectories are then post-processed in Matlab. The linear acceleration and angular velocities are computed and used for IMU simulation. The results are written to a file. Finally, the true poses are fed to a custom C++ library which simulates the camera measurements.

3.3 Rigid Body Simulation

Rarely the goal of motion capture is to obtain the trajectories of individual markers, rather their positions are used to derive the poses of rigid bodies. For this reason the trajectories of the markers are not directly simulated but computed from the pose of the object they are attached to.

Deriving the position of a marker from the pose of the rigid body it is attached to is a trivial task and can be solved using rigid body kinematics. The actual difficulty lies in generating the trajectories of the rigid bodies in the first place. Force-based approaches to rigid body simulation are common in mechanical engineering. They try to answer the question of how a system reacts to internal and external forces, e.g. how the position and rotation of a satellite

changes when firing its reaction control thrusters. But those approaches are not suited for this thesis. Forces and torques are not known and not of importance in this context, rather the positions and rotations of the bodies are given and the goal is to derive the corresponding accelerations and angular velocities for simulating the IMU measurements.

In video game development characters and objects are often moved through the use of animations. Their pose is only given at discrete keyframes and the movement in between those is interpolated through splines. This allows simulating the pose without the need to consider accelerations and velocities. Also, since the movement is interpolated in between keyframes, motions can be described using only a small number of known poses but sampled at much higher rates. Finally, the animations can be used in a game engine to move the characters and objects. But in our context game engines are not only useful for generating rigid body movements but can also be used to aid the simulation of the motion capture systems cameras, which is explained in greater detail in Chapter 3.5.

The rigid body simulation is implemented using the Unity game engine. Unity uses a hierarchical design where *GameObjects* are the main building blocks. A GameObject is an object that can be placed in the scene to fulfill a role or task. It can also act as a container holding other GameObjects as children. *Components* are used to provide functionality to GameObjects. Unity supplies a variety of predefined and configurable components for common game development tasks like animation, collision or physics. Custom components can be implemented using the C# language and the UnityEngine library.

For the rigid body simulation, each body that should be tracked is equipped with markers just like it would be done in reality. For convenience a red spherical marker prefab[1] was added. An example of how they can easily be attached to an object can be seen in Figure 3.2. Setting up the cameras is as simple as setting up the markers. They are simply added to the

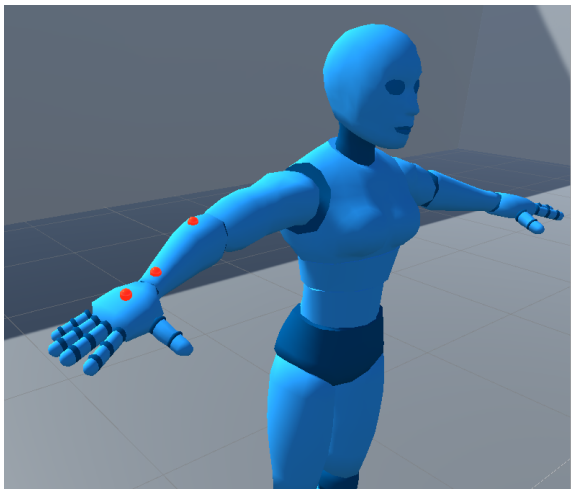

Figure 3.2: Three markers attached to the right arm of a human model in Unity using the simulation framework.

scene and positioned in space as shown in Figure 3.3. A custom script for handling marker occlusion detection is added to the container holding the cameras. This script uses ray casting to determine if a marker is occluded by another object. Ray casting is a common method in

[1] A prefab is a template GameObject.

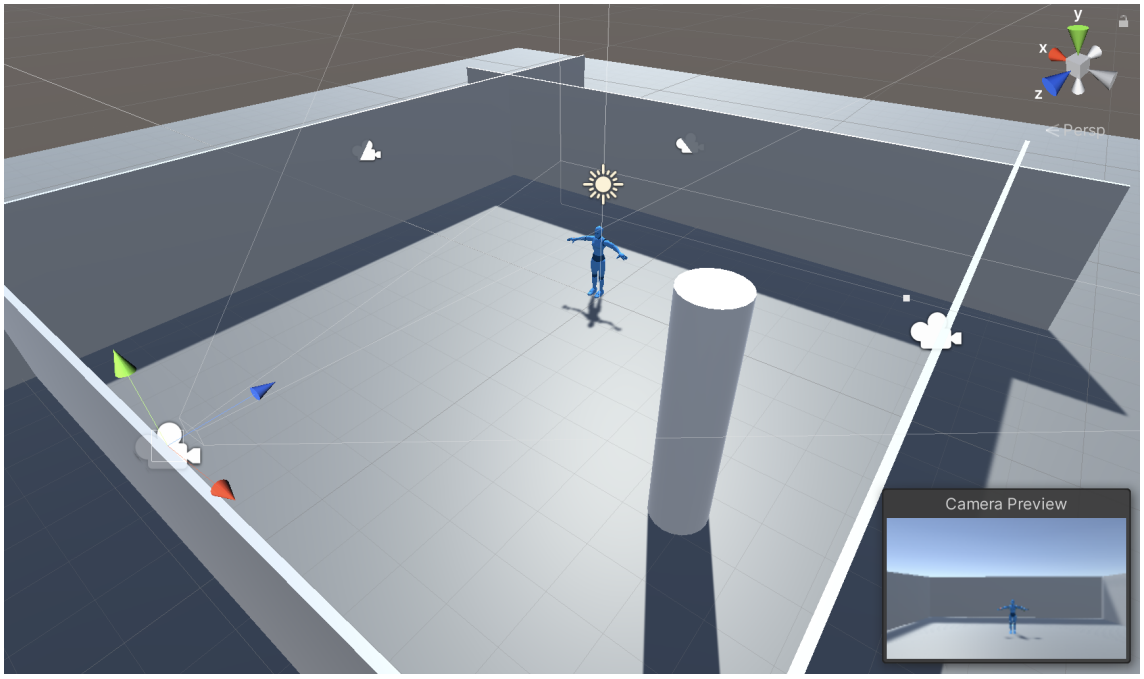

Figure 3.3: An example of a capture system set up in Unity using the simulation framework.

game development where a ray is shot from an origin in a direction and its collisions with other objects is computed. Finally, the bodies to track, the container holding the cameras, the output path and the frame rate are set in the *SimulationSettings* component which is shown in Figure 3.4. When running the simulation, the capture rate is fixed to the one specified in

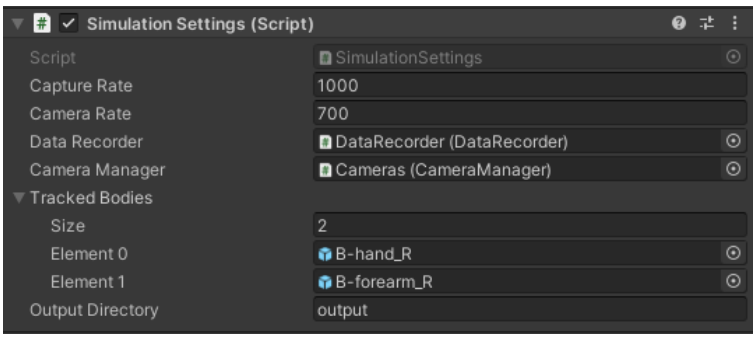

Figure 3.4: *SimulationSettings* component interface.

the simulation settings and the measurements are written to a CSV file. Also, a JSON file is created holding information about the simulation setup, e.g. the camera poses.

The developed Unity framework provides a visual and quick approach to simulating marker trajectories of any number of rigid bodies or rigid body systems.

3.4 IMU Simulation

In order to simulate IMU measurements we first need to obtain the true acceleration and angular velocity of the markers. The rigid body simulation detailed in Chapter 3.3 is used as

a basis. A markers full kinematic state can be derived from the rigid body it is attached to. Since the Unity simulation computes rigid bodies pose instead of accelerations and velocities the marker poses need to be numerically differentiated.

Using a central difference to compute the velocity and the acceleration from the position results in noisy measurements when using high sample rates as shown in Figure 3.5. Those

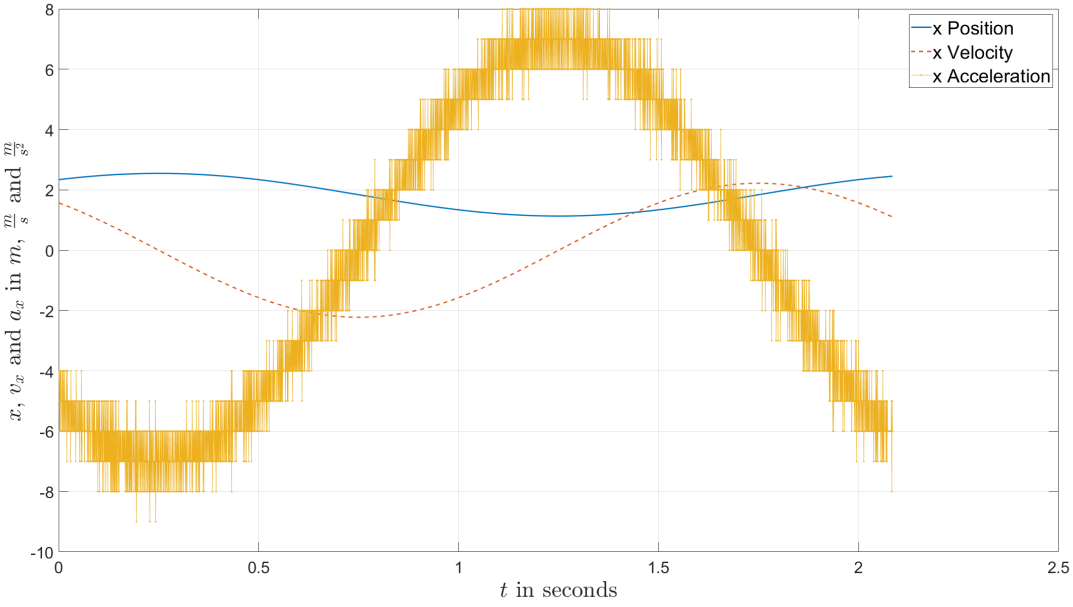

Figure 3.5: x position, velocity and acceleration in m, $\mathrm{m\,s^{-1}}$ and $\mathrm{m\,s^{-2}}$ of a marker based on a rigid body simulation run in Unity. A capture rate of $1\,\mathrm{kHz}$ was used. The velocity was obtained by differentiating the position and the acceleration was obtained by differentiating the velocity. In both cases central differences were used.

are caused by the internal use of 4-byte floating point numbers for pose animation in the Unity engine. The high sample rates result in small differential movements between frames that exhaust the floating point precision and in turn cause quantisation errors which are amplified through differentiation. The resulting acceleration measurements are too noisy to be used for simulating the IMU.

Since the Unity-internal number format cannot be changed the measurements need to be smoothed. A first approach was smoothing the acceleration itself. A method based on total-variation regularisation proposed in [15] was tested. The result of applying the method to a noisy set of simulated measurements is shown in Figure 3.6. While the noise was significantly reduced the resulting acceleration still suffers from sharp fluctuations. Also, reintegrating the regularised acceleration results in deviations from the original positions, hence introducing new errors.

Instead a different approach is chosen. The position of each marker is approximated by three cubic smoothing splines (one for each coordinate direction). A cubic smoothing spline $s(t)$ is

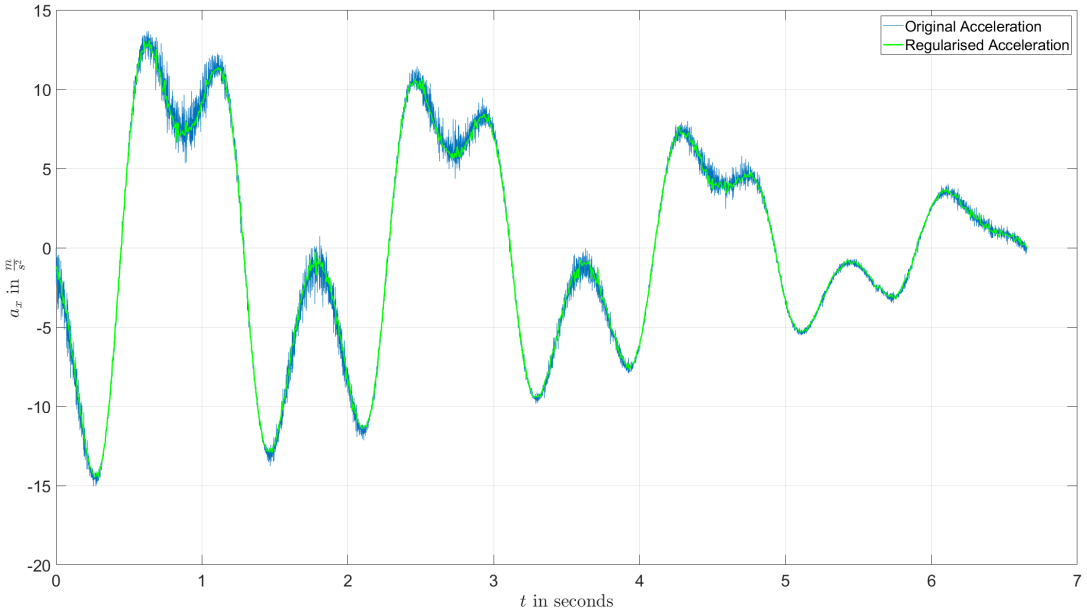

Figure 3.6: Original acceleration obtained from numerically differentiating (central difference) the velocity and acceleration obtained by using the regularisation method proposed in [15]. Both are measured in $\mathrm{m\,s^{-2}}$.

one that minimises the cost function

$$c = p \sum_i \left(x_i - s(t_i)\right)^2 + (1-p) \int \left(s''(t)\right)^2 dt \tag{3.1}$$

Here p is a smoothing parameter between 0 and 1 which controls the amount of smoothing. Larger values of p result in a tighter fit while lower values result in smaller second derivatives and therefore a looser but also smoother approximation. To compute the goodness of the fit the root mean square error (RMSE) is computed by

$$e_{RMS} = \sqrt{\frac{\sum_i \left(x_{i,true} - x_{i,spline}\right)^2}{N}} \tag{3.2}$$

where x_{true} is the position (one direction of the position vector is considered at a time) obtained from the rigid body simulation, x_{spline} is the position as obtained from the smoothing spline and N is the total number of frames. The smoothing spline optimisation was implemented using Matlab and its Spline Fitting Toolbox. For our specific case values of p very close to 1 have shown to provide the best results since they provide close to perfect interpolation while smoothing out the numeric irregularities. Figure 3.7 shows how fitting a smoothing spline to the simulated position can result in smoother accelerations without introducing large errors in the position measurements. For long trajectories, i.e. trajectories with more than approximately 20000 frames, the optimisation starts taking long computation times.

The angular velocity can be obtained from the rotation and is usually sufficiently smooth without further processing.

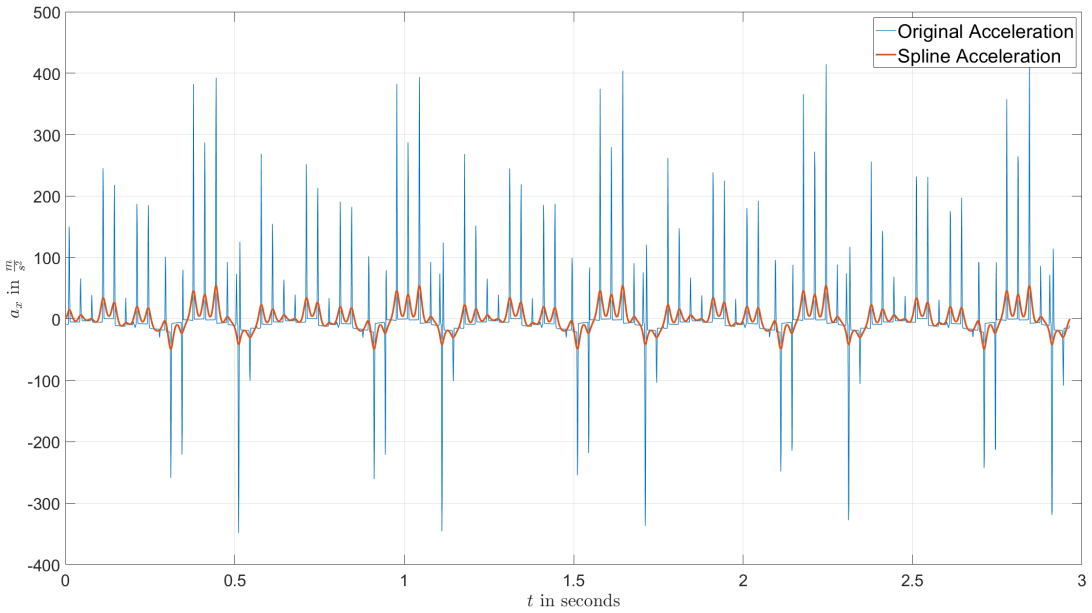

Figure 3.7: Original acceleration obtained from numerically differentiating (central difference) the marker position twice and acceleration obtained by analytically differentiating the smoothing spline fitted to the marker position twice. Both are measured in $\mathrm{m\,s^{-2}}$. A smoothing parameter of $p = 0.999999$ was used. The position-spline fitting error (RMS) is in the order of $10^{-3}\mathrm{m}$.

The IMU measurements can be simulated using the computed acceleration and angular velocity. This process consists of first transforming the measurements to the IMUs body-fixed coordinate system and adding the gravity vector to the accelerations. Then sensor characteristic errors like noise, biases, measurement range etc. are added. The Matlab Navigation Toolbox provides a simulated IMU model which can be configured to have specific noise and measurement characteristics. Once configured the IMU model can be fed true linear accelerations, angular velocities and rotations to compute the simulated IMU measurements.

All the computations outlined in this section have been implemented as a Matlab class. For convenience a configurable script was provided that eases the workflow by setting up the Matlab simulation and calling the appropriate methods.

3.5 Camera Simulation

One of the motivations of fusing IMU data with conventional motion capture measurements is to counter camera occlusion. Occlusion can be simulated using a method called ray casting. A ray is shot from a source in the direction of a target and its collisions with objects are computed. If the first objects that it collides with is the marker, it is visible from the camera, if not then another object occludes the view. Ray casting is a method used extensively in game development, e.g. for determining if a mouse pointer click hits an object in a three dimensional world. Thus, the same game engine used for simulating the rigid body movements

can be used to simulate camera occlusion. The source of the ray is simply selected as the camera and the ray is shot in the direction of the marker. This is repeated every simulation step for every camera-marker combination.

Another aspect to consider are the measurement errors of the camera sensors, or more precisely the errors of the linear detector sensors that make up the PhaseSpace cameras. Errors to consider are irregularities in the projected intensity profiles of the LED markers on the sensors, processing inaccuracies of these intensity profiles when extracting the intensity spikes, quantisation of the measurements and lens distortion. Surely this is not an exhaustive list but due to the limited scope of this project not all error sources can be considered and even not all of the ones mentioned can be simulated in detail. Rather for sake of simplicity, lens distortion is omitted and the errors related to intensity profiles and processing are assumed to behave as zero-mean Gaussian noise on the true sensor measurement. Quantisation errors are implemented and are based on the pixel resolution of the linear detectors.

Finally, to simulate the camera measurements the true sensor measurements need to be obtained. For this purpose the pinhole camera model is adapted to accommodate the reduced dimension of the linear detector sensor, i.e. to a one dimensional sensor. The resulting projection matrix $\mathbf{C}$, that maps a point $\mathbf{x}$ from the three dimensional local coordinate system to the sensor frame, can be obtained as follows:

$$\hat{\mathbf{u}} = \begin{bmatrix} \hat{u} \\ \lambda \end{bmatrix} = \mathbf{C} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{I}\,\mathbf{R}\,\mathbf{T} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{3.3}$$

Here, $\mathbf{I}$ is the intrinsic matrix with the focal length f and sensor centre offset c, $\mathbf{R}$ is the rotation matrix that rotates a vector from the local frame to the sensor frame and $\mathbf{T}$ is the translation matrix with the position $\mathbf{x_0}$ of the linear detector in the local coordinate system. Note that the absolute position of the marker on the sensor can then be obtained by normalising the homogeneous point $\hat{\mathbf{u}}$:

$$u = \frac{\hat{u}}{\lambda} \tag{3.4}$$

3.6 Motion Capture Simulation

The simulated linear detector measurements can now be used to compute a new position estimate. A more naive approach would be to use the true positions obtained from the rigid body simulation and add noise to them to simulate the position estimation to avoid the projection and estimation steps. But the difficulty lies in finding the appropriate probability distribution that fits the underlying uncertainty. Since the measurement noise is introduced on the linear detector sensor it needs to be projected back into three dimensional space. But note that the projection matrix $\mathbf{C}$ of a marker on the sensor is not invertible, hence the true marker position cannot be reconstructed from an individual sensor measurement, but can be constrained to a plane in three dimensional space. Assuming zero-mean normal distributed measurement noise on the sensor, it becomes clear that the back-projected noise does not follow a trivial distribution. For this reasons it is simpler to first project the marker position onto the linear

detectors, add the noise and re-triangulate its position.

The triangulation algorithm is adapted from [16] to work with linear detectors. A solution $\mathbf{x}$, given the sensor measurements u_i, is a vector that solves the projection equation 3.3 of each individual linear detector. Visually speaking, we find the intersection of the back-projection planes. Under the presence of noisy measurements it is highly unlikely that an intersection exists. Therefore, the solution that minimises the algebraic error is determined. This equates to finding the least square solution of the equations

$$\hat{\mathbf{u}}_i = \alpha\, \mathbf{C}_i \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \alpha\, \mathbf{C}_i \hat{\mathbf{x}} \tag{3.5}$$

where $\hat{\mathbf{u}}_i$ is the sensor measurements of the i-th linear detector in homogeneous coordinates, $\mathbf{C}_i$ is the corresponding projection matrix and α is a similarity factor that results from the use of homogeneous coordinates. Note that $\hat{\mathbf{u}}_i$ and $\mathbf{C}_i\hat{\mathbf{x}}$ are related through the scalar α, i.e. they have the same direction but different scale. Instead of solving the equations 3.5, which involves solving for α in addition to $\mathbf{x}$, we define a new objective using the similarity constraint. Let us express the matrix $\mathbf{C}_i$ in terms of its rows:

$$\mathbf{C}_i\, \hat{\mathbf{x}} = \begin{bmatrix} - \mathbf{c}_{i,1} - \\ - \mathbf{c}_{i,2} - \end{bmatrix} \hat{\mathbf{x}} = \begin{bmatrix} \mathbf{c}_{i,1} \cdot \hat{\mathbf{x}} \\ \mathbf{c}_{i,2} \cdot \hat{\mathbf{x}} \end{bmatrix} \tag{3.6}$$

Since $\hat{\mathbf{u}}_i$ and the vector defined by equation 3.6 are related by the scale factor α, the matrix composed from those two must be rank deficient. Now we can define the new objective

$$det\left(\begin{bmatrix} \mathbf{c}_{i,1} \cdot \hat{\mathbf{x}} & u_i \\ \mathbf{c}_{i,2} \cdot \hat{\mathbf{x}} & 1 \end{bmatrix} \right) = \mathbf{c}_{i,1} \cdot \hat{\mathbf{x}} - u_i\, \mathbf{c}_{i,2} \cdot \hat{\mathbf{x}} = (\mathbf{c}_{i,1} - u_i\, \mathbf{c}_{i,2}) \cdot \hat{\mathbf{x}} = 0 \tag{3.7}$$

Each linear detector provides one equation of this form, which shows that at least three independent measurements are necessary to triangulate a three dimensional point. The resulting homogeneous system for measurements from n linear detectors is

$$\bar{\mathbf{C}}\, \hat{\mathbf{x}} = \begin{bmatrix} \mathbf{c}_{1,1} - u_1\, \mathbf{c}_{1,2} \\ \vdots \\ \mathbf{c}_{n,1} - u_n\, \mathbf{c}_{n,2} \end{bmatrix} \hat{\mathbf{x}} = \mathbf{0} \tag{3.8}$$

Assuming the system is consistent the solution is the nullspace of $\bar{\mathbf{C}}$. If it is inconsistent, as it is usually the case with noisy measurements, the least square estimate is computed. Both cases can then be numerically solved, e.g. using singular value decomposition.

The camera and motion capture simulation was implemented as a C++ library called PhaseSpaceSimulator (PSS). C++ was chosen since most libraries for factor-graph based inference are also implement in C++. The structure of the PSS library was adapted from the real world setup of the PhaseSpace system. A PhaseSpace camera consists of two linear detectors each and several cameras make up a capture system. The processing and communication is then handled by the Core server. Therefore, the three most important classes of the PSS library are the `LinearDetector`, `Camera` and `Core` classes.

The `LinearDetector` class encapsulates the pinhole camera model presented in Chapter 3.5. It provides methods for projecting a point onto the sensor and for generating the estimation equation required for triangulation. Generating the estimation equations for a point involves checking if the point lies in sight of the camera, project the point onto the sensor, adding noise and assembling and returning the estimation equation if the measurement is valid, i.e. if it lies on the actual sensor. A `LinearDetector` object can be constructed from the camera intrinsic parameters and its pose. Additionally, a constructor from the detector field of view is provided for convenience. The `LinearDetector` class also supports using a calibrated pose such that the real pose is used for simulating the measurement and the calibrated pose is used for estimation.

Two `LinearDetector` objects can then be used to create a `Camera` instance. This class provides an interface to obtain the estimation equations for triangulation from both its sensors. A convenience constructor from camera intrinsics and the camera pose is provided that assumes both linear detectors are arranged perpendicular to each other, i.e. one rotated 90 degrees around the view-axis of the other.

Once all `Camera` objects are defined they can be used to initialise an instance of the `Core` class. This class holds a map of all the cameras in the system and their identifier. Most importantly, the `Core` class implements the camera triangulation algorithm. Given a point in space and a list of camera identifiers from which it should be estimated, the `Core` object will gather all the estimation equations from the cameras, assemble them to a matrix and finally perform the singular value decomposition of the matrix to solve the homogeneous linear system.

Additional classes that have been implemented for utility or convenience are `Rot3`, `Pose3` and `SimulationContext`. The first two are used to represent a rotation and a pose in three dimensional space. The `SimulationContext` class is provided for convenience. An instance can be constructed from a path to the metadata JSON file, a path to the measurements CSV file and a path to the desired output file. On construction it will then create a `MetaData` struct from the JSON file, prepare the CSV reader and prepare the output CSV file. The `SimulationContext` instance provides a method for parsing a single line of the measurements file per call and returns a `Measurement` struct. Also, it defines a convenience method for writing an estimate to the output file. A `SimulationContext` instance can also be used to construct a `Core` object, making setting up a simulation environment as simple as:

```
1 SimulationContext simContext{ metaPath, measurementsPath, outputPath };
2 Core core{ simContext };
```

This way no manual construction of linear detectors and cameras needs to be done.

Two custom exceptions were implemented to provide context for when the linear system for triangulation is underdetermined and for when a projection of a point fails.

The PSS library is independent of other compiled libraries. Its only dependencies are three header-only libraries: *Eigen*[2] for linear algebra, *ben-strasser Fast C++ CSV Parser*[3] for CSV

[2] `https://eigen.tuxfamily.org/`
[3] `http://https://github.com/ben-strasser/fast-cpp-csv-parser`

parsing and *nlohmann JSON*[4] for JSON parsing.

Major features of the PSS library are covered by unit tests. The *googletest*[5] framework was used for this purpose. A custom test fixture that handles creating dummy files for testing the `SimulationContext` class was created.

In addition to unit tests, *Valgrind*[6] tools where used to test the library for memory bugs like memory leaks or usage of uninitialised variables.

A full CMake configuration is provided for the PSS library and the pose estimation executable. It allows selecting whether the unit tests and the estimation executable should be built and automatically gathers the individual dependencies if selected. This way the PSS library without tests and executable can be built without any further compiled dependencies. The build process was tested for msvc 16.7, g++ 9.3.0 and Apple clang 11.0.3.

The PSS library is fully documented using *Doxygen*[7]. The configuration for running doxygen on the project is provided in the *Doxyfile*.

[4] `http://https://github.com/nlohmann/json`
[5] `https://github.com/google/googletest`
[6] `https://valgrind.org/`
[7] `https://www.doxygen.nl/`

Chapter 4

Visual-Inertial Marker Tracking

In this chapter a new approach to estimating marker positions in high performance motion capture systems is presented. First, a short introduction to the modified capture system is given in Section 4.1. The factor graph used to model the sensor fusion and inference problem is described in Section 4.2. Finally, the implementation of the estimator is presented in Section 4.3.

4.1 Visual-Inertial Capture System

The motion capture system as described in Chapter 2.1.2 was modified by adding consumer-grade IMUs to each LED marker. This modification was done by PhaseSpace as part of their development work. While they integrated the IMUs such that their data can be streamed through the systems API they did not yet implement a sensor fusion and estimation solution. The simulation framework presented in Chapter 3 is based on this modified version of the PhaseSpace X2E motion capture system.

4.2 Graphical Model

As outlined in Chapter 2.4, factor graphs are well suited to model nonlinear inference problems. A general graph design is adapted from [17] where IMU and GPS measurements are used to estimate an objects pose and velocity as well as the IMU biases. The graph used in this thesis is shown in Figure 4.1. Instead of a GPS system the measurements from the motion capture camera system are used, which are similar in the sense that they provide a direct drift-free measurement or constraint of the markers position. Indeed, when first estimating the marker position using the method shown in Chapter 3.6 (or any other method that produces a position estimate from camera readings) the point estimate can be added to the graph

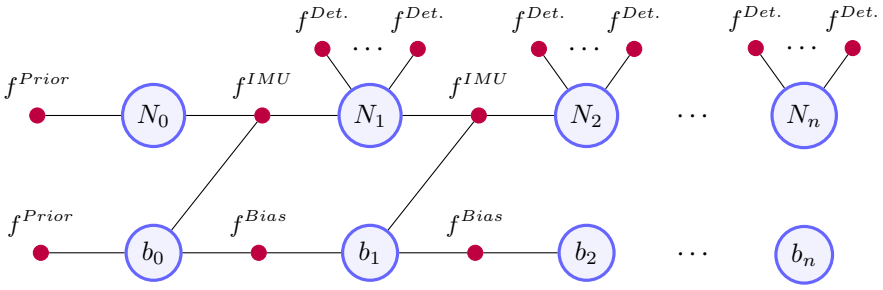

Figure 4.1: Factor graph used for estimating the poses and velocity of a single marker and the bias vector of the IMU. Here N_i is not the pose but the NavState (pose and velocity) of the marker. While the implementation differs slightly, i.e. pose and velocity are separate variables, this equivalent representation was chosen for the purpose of improved readability.

as a single loosely coupled[1] GPS-factor. Such a factor acts on a single variable, in our case on a single pose $\mathbf{P}_i$, and constraints the markers position. This approach requires the marker to be visible from at least three linear detectors (see Chapter 3.6) to produce a position estimate and a valid factor for the graph. But when only one or two linear detector measurements are available no camera-based estimate can be produced and therefore no GPS-like factor can be added even though the measurements would provide some constraints on the markers position. Thus, a better method is to directly use the individual measurements of the linear detectors in a tightly coupled manner such that for each measurement an individual factor is added to the graph. To implement such a factor the measurement function $h(\mathbf{P})$ and its Jacobian $\mathbf{J}$ need to be provided for use in the least-squares optimisation. The measurement function was already derived in Equation 3.3 and 3.4. Both equations can be combined to the measurement function

$$h(\mathbf{P}) = \frac{\hat{u}}{\lambda} = \frac{c_{11}x + c_{12}y + c_{13}z + c_{14}}{p_{21}x + c_{22}y + c_{23}z + c_{24}} \tag{4.1}$$

where c_{ij} are the entries of the projection matrix $\mathbf{C}$. Note that $h(\mathbf{P})$ is nonlinear due to the homogeneous normalisation. Also, $h(\mathbf{P})$ is optimised over the group of poses, a 6 dimensional Lie Group called $SE(3)$ (see Chapter 2.4.2). Therefore, the special structure of $SE(3)$ needs to be respected when deriving the Jacobian. The pose increments $\boldsymbol{\xi}$ are defined in the tangent space of $SE(3)$ at the identity and mapped back on the manifold through the exponential map. The Jacobian $\mathbf{J}$ at a linearisation point $\mathbf{P}_0$ needs to satisfy

$$h\left(\mathbf{P}_0 e^{\boldsymbol{\xi}}\right) \approx h(\mathbf{P}_0) + \mathbf{J}\boldsymbol{\xi} \tag{4.2}$$

which is similar to the first order linearisation of a function defined over a vector space. The Jacobian for the measurement function $h(\mathbf{P}_0 e^{\boldsymbol{\xi}})$ is defined as

$$\frac{\partial h\left(\mathbf{P}_0 e^{\xi}\right)}{\partial \boldsymbol{\xi}} = \begin{bmatrix} \mathbf{0}_{1\times 3} & \mathbf{J}_t \mathbf{R}_0 \end{bmatrix} \tag{4.3}$$

where $\mathbf{J}_t$ is the part of the Jacobian related to the position defined as

$$\mathbf{J}_t = \begin{bmatrix} \frac{c_{11}\lambda - c_{21}\hat{u}}{\lambda^2} & \frac{c_{12}\lambda - c_{22}\hat{u}}{\lambda^2} & \frac{c_{13}\lambda - c_{23}\hat{u}}{\lambda^2} \end{bmatrix} \tag{4.4}$$

[1] A *loosely coupled* GPS integration is one where the GPS position estimate is fused with the navigation solution. In contrast, a *a tightly coupled* integration uses the raw GPS pseudo-range and Doppler measurements in the estimator.

and $\mathbf{R}_0$ is the rotation matrix associated with the pose $\mathbf{P}_0$. Note that $\mathbf{J}_t$ is simply the Jacobian of $h(\mathbf{P})$ with respect to the position. The right-multiplication of $\mathbf{J}_t$ with $\mathbf{R}_0$ results from the Jacobian of the term $\mathbf{P}_0 e^{\xi}$ [18]. The measurement function 4.1 and its Jacobian 4.3 can then be used to define a new *Linear Detector Factor* for use in factor graph based inference.

While the linear detector factor derived above only acts upon a single pose, we also need a factor to constrain the movement in-between two poses. The IMU measurements are used for this task and incorporated into the factor graph using IMU factors. There exist different IMU factors based on the preintegration theories presented in Chapter 2.3. The IMU factor is also connected to an IMU bias variable. The changes in between two IMU bias variables can be constrained using a single factor that is based on a constant bias model. Alternatively, more complex models can be used if the bias characteristics are better known. The initial states of the variables are constrained by a simple prior.

Depending on the application it may be required to perform real-time tracking. Batch optimisation of the inference problem becomes too computationally expensive. Therefore an incremental approach like iSAM2 [14] is required. For high prediction rates even iSAM2 might not be able to perform in real-time. In those cases an incremental fixed-lag smoother can be applied.

4.3 Implementation

For testing the performance of the proposed pose graph of Chapter 4.2 a C++ executable was created that uses the PSS library to simulate the camera system given the meaurements obtained from the simulation pipeline. The implementation heavily depends on the GTSAM[2], Georgia Tech Smoothing and Mapping, library which provides classes for creating and performing inference on factor graphs. It also comes with a variety of predefined factors and variable types.

As IMU factor the default one provided by the GTSAM library was used. This factor is a more efficient implementation of [7] and performs integration on the tangent space of the NavState manifold. The custom linear detector factor from Chapter 4.2 was implemented as a new class `LinearDetectorFactor` which derives from the GTSAM class `NoiseModelFactor1`. `LinearDetectorFactor` implements the virtual function `evaluateError(...)` which assigns the measurement Jacobian at the current estimate to a pointer passed as argument and returns the error between the predicted measurement and the true measurement. GTSAMs `PriorFactor` was used to define the priors and the `BetweenFactor` was used as IMU bias factor. The fixed lag smoother was implemented using the `IncrementalFixedLagSmoother` class of the GTSAM-unstable library.

When running the executable first the `SimulationContext` is set up and then used to construct a `Core` instance. Next, the factor graph is set up which involves defining noise models for the factors and adding priors. Then the executable iterates through the measurements and creates the appropriate factors. If no linear detector measurement is available no variable

[2] `http://https://github.com/borglab/gtsam`

is created and the measurement is simply preintegrated. If new variables were created in the current iteration, initial values for them are inserted using the previous estimates and the current preintegrated IMU measurements. Next, the incremental smoother is used to marginalise old variables and to generate an estimate for the current time step. Finally, the new estimate is written to the output file and the iteration for the next measurement is started.

While the executable compiles under msvc 16.7, g++ 9.3.0 and Apple clang 11.0.3, there seems to be a runtime bug occurring under macOS which results in `Camera` objects not being found in the `Core` instances `CameraMap` when using graph-based estimation.

Chapter 5

Evaluation

This chapter evaluates the performance of the factor graph based estimation method, as presented in Chapter 4 by comparing it with camera-only triangulation based on the method derived in Section 3.6. The three simulation scenarios used for evaluation are described in Section 5.1. The estimator performance without occlusion is tested in Section 5.2. Scenarios with occlusion are evaluated in Section 5.3. Finally, the results are discussed in Section 5.4.

5.1 Simulation Scenarios

Three simulation scenarios are investigated:

1. A single marker placed on the corner of a cube. The cube is smoothly moving up and down while rotating around its vertical axis. Both one translation cycle and one rotation take $2\,\mathrm{s}$. The resulting trajectory is shown in Figure 5.1.
2. A single marker placed on the corner of a cube. The cube is smoothly moving up and down but stays at the maximum elevation for $0.5\,\mathrm{s}$. At the same time it is rotating around its vertical axis. Both one translation cycle and one rotation take $2\,\mathrm{s}$. The resulting trajectory is shown in Figure 5.2.
3. A single marker placed on the top of the hand of a dancing humanoid character, see Figure 3.2. The trajectory of the movement is shown in Figure 5.3.

Scenario 1 is the one with the simplest kinematics while Scenario 3 is the most demanding due to comparably high linear accelerations and angular velocities.

An 8-camera setup as shown in Figure 5.4 was used for generating the marker trajectories. The standard deviation of the linear detector measurement noise was set as $1\,\mathrm{mm}$. Each linear detector had a field of view of $120°$ and a resolution of 30000 pixels. Each scenarios IMU measurements were simulated a single time in Matlab. The IMUs noise and measurement

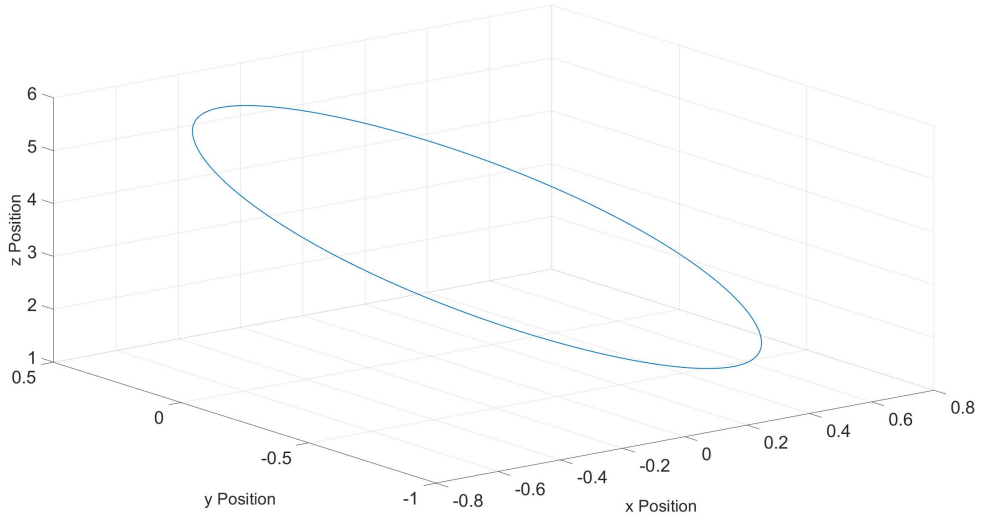

Figure 5.1: Trajectory of Scenario 1.

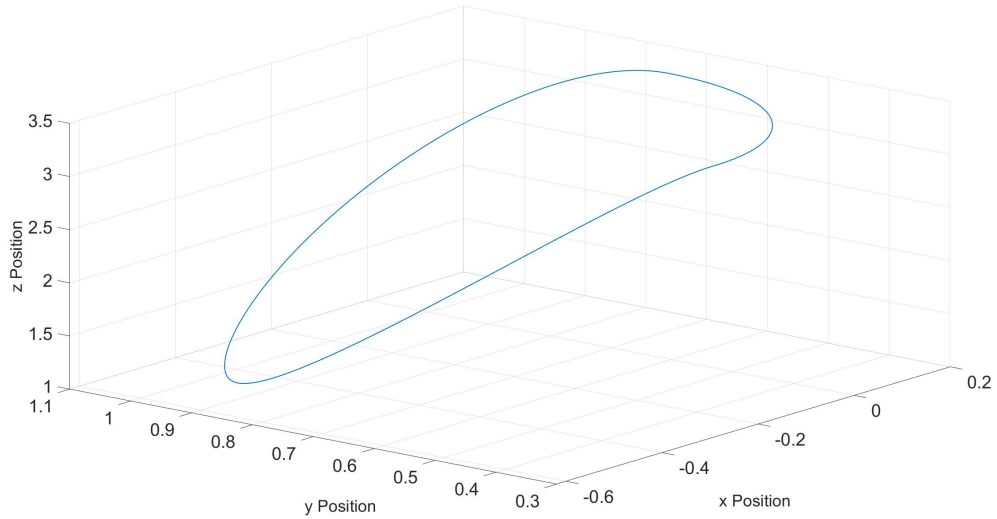

Figure 5.2: Trajectory of Scenario 2.

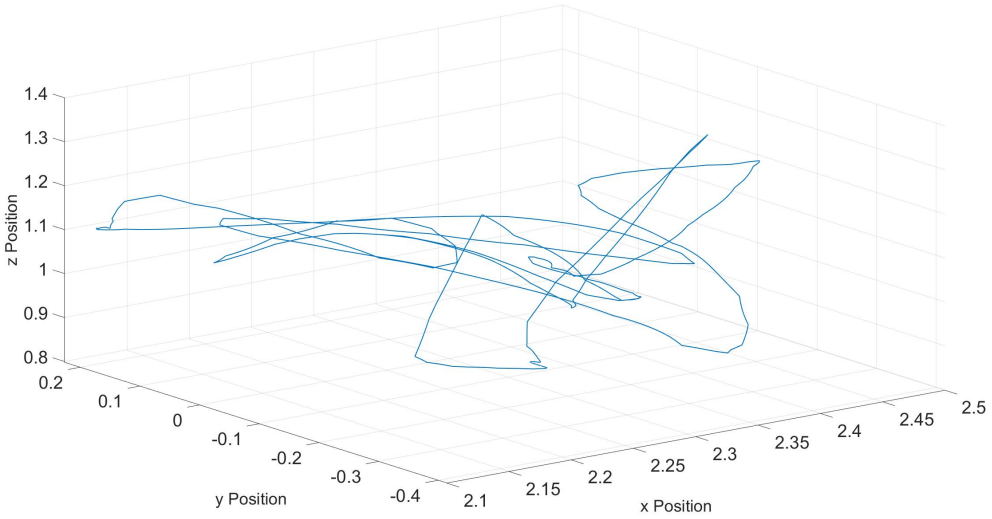

Figure 5.3: Trajectory of Scenario 3.

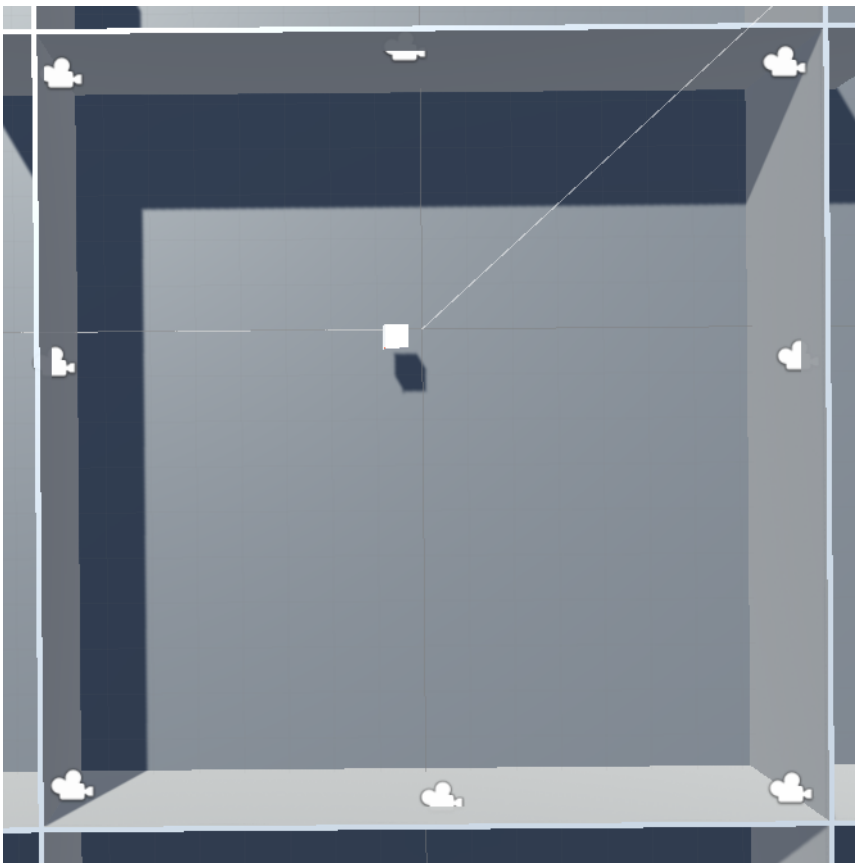

Figure 5.4: Top-down view of the Unity simulation setup used for generating the marker trajectories. 8 cameras are positioned on the walls. The object in the middle is the cube used for Scenario 1 and 2.

characteristics were kept the same among the three scenarios, see Appendix C for the specific values. A sampling rate of $1\,\mathrm{kHz}$ for the IMU and $700\,\mathrm{Hz}$ for the linear detectors was used. Each scenario was estimated using three different methods:

1. Camera-only triangulation (see Section 3.6).
2. Real-time incremental fixed lag smoothing based on iSAM2 with a 10 Frames lag.
3. Post-processing using batch optimisation that uses the incremental solution from Method 2 as initial values. A Levenberg-Marquardt based optimiser from the GTSAM library was used.

5.2 Tracking without Occlusion

This section evaluates the estimator performance for the three simulation scenarios outlined in the previous section under the assumption that no occlusion occurs, i.e. the markers are visible at all times from each camera. The performance was compared by computing the error for each estimated point as

$$e_i = \|\hat{\mathbf{x}}_i - \mathbf{x}_i\|_2 \tag{5.1}$$

where $\hat{\mathbf{x}}$ is the true position obtained from the Matlab smoothing spline fitting and $\mathbf{x}$ is the estimated position. The rotation estimates were not evaluated since for direct marking tracking (in contrast to rigid body tracking) only the position is of importance. The errors were plotted as box plots where the central red bar indicates the mean, the upper and lower bounds of the box indicate the 75th and 25th percentile and the whiskers the maximum and minimum values that are not considered outliers. The values within the whiskers cover approximately 99.3 percent of the data.

First, the camera-only and real-time performance were compared in Figure 5.5 and 5.6. The camera-only error is consistent among all three scenarios with only minor variations. Method 1 resulted in Scenario 1 in a lower mean error and variance while for the other two scenarios the errors increased. Especially in Scenario 3 a significant increase in the error of about a magnitude could be observed.

Figure 5.7 shows a comparison of the error for the three estimation methods for every simulation scenario. Batch optimisation consistently resulted in a significant reduction of the mean estimation error and the variance. In Scenario 1 batch optimisation further improved the estimates compared to fixed-lag smoothing. Noteworthy is that batch optimisation in Scenario 2 resulted in better estimates than camera-only triangulation while fixed-lag smoothing resulted in worse. In Scenario 3 batch optimisation did not lead to better estimates than camera-only triangulation but still considerably reduced the mean error and variance.

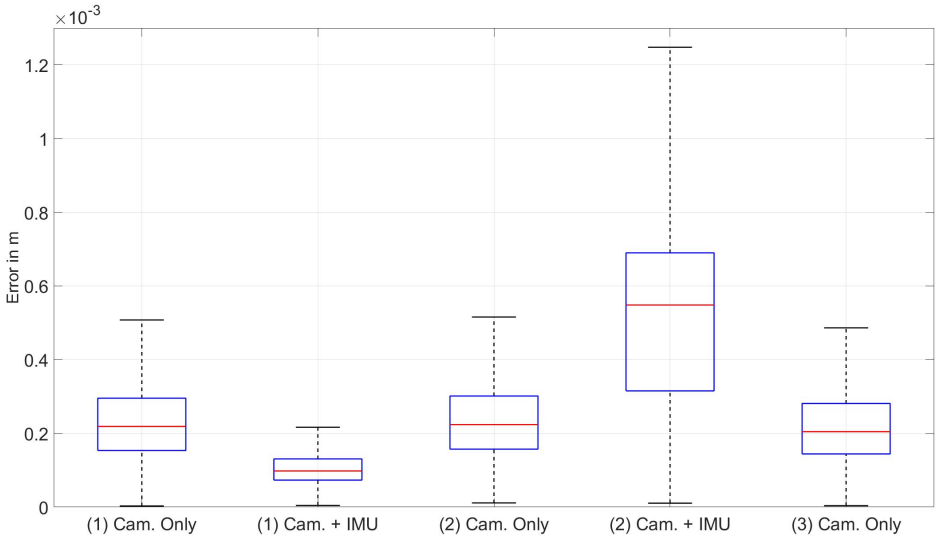

Figure 5.5: Error of the real-time estimated trajectories of Scenario 1, 2 and 3 for a single simulation run of each. *Cam. Only* refers to Method 1 and *Cam. + IMU* to Method 2. The number in parentheses refers to the simulation Scenario.

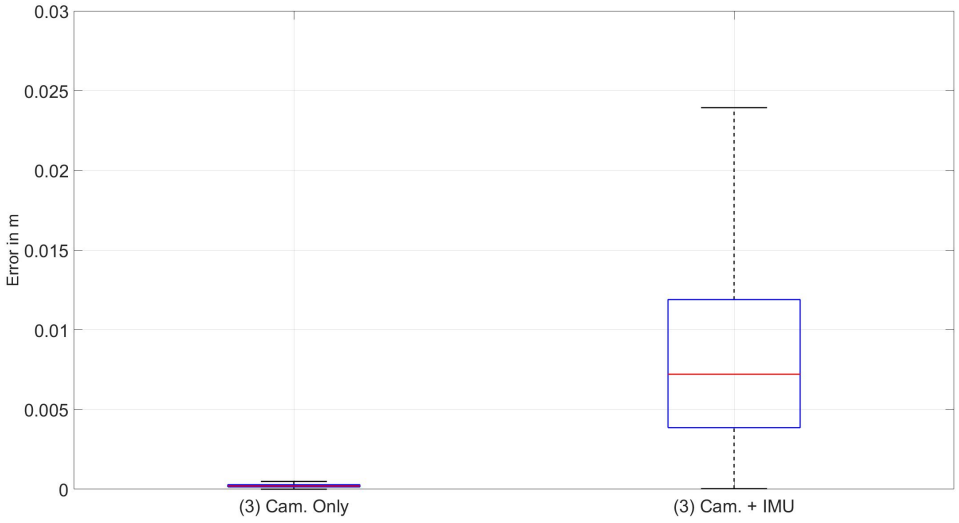

Figure 5.6: Error of the real-time estimated trajectories of Scenario 3 for a single simulation run. *Cam. Only* refers to Method 1 and *Cam. + IMU* to Method 2.

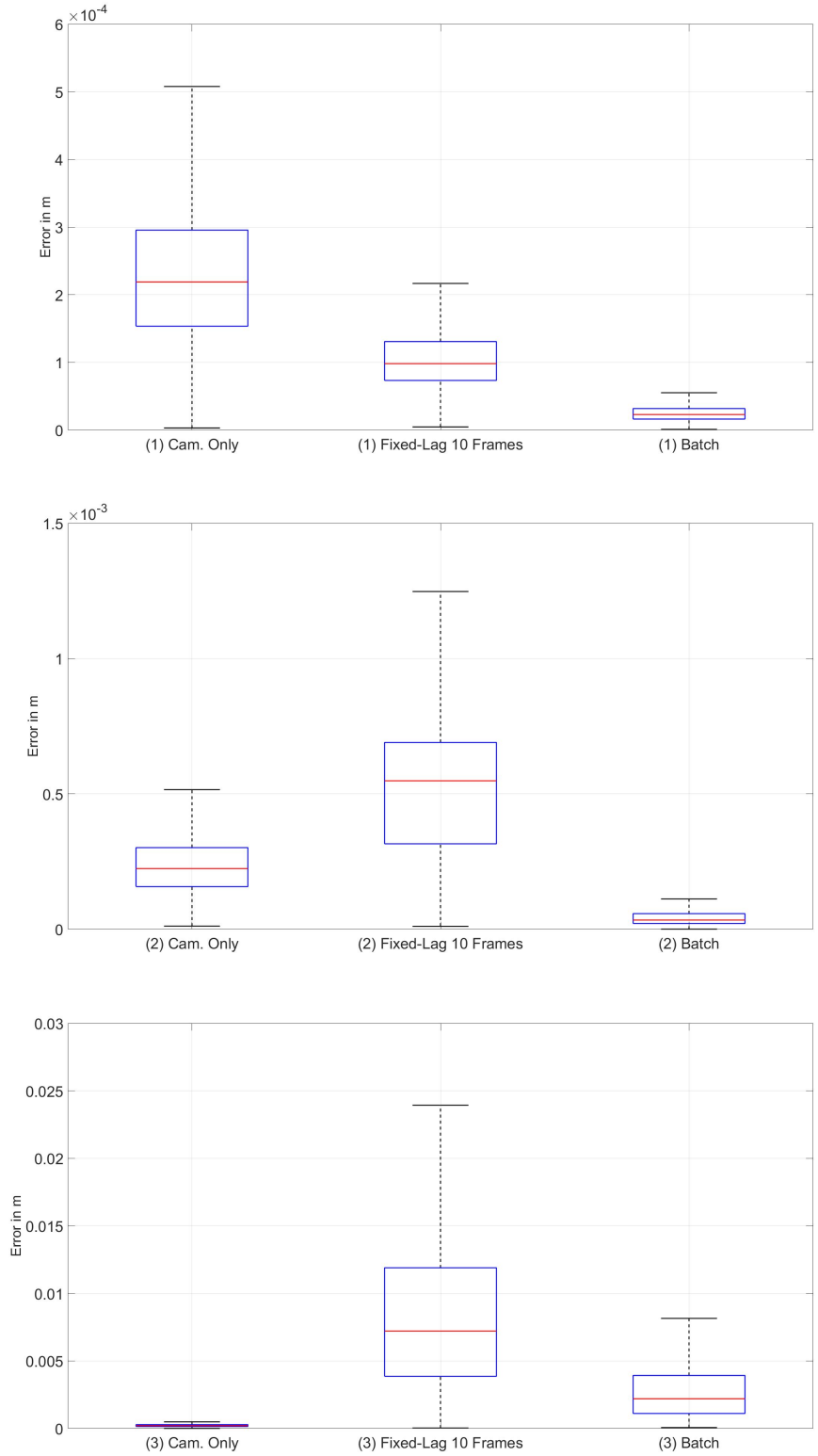

Figure 5.7: Comparison of errors for the three estimation methods applied to all three Scenarios. *Cam. Only* refers to Method 1, *Fixed-Lag 10 Frames* to Method 2 and *Batch* to Method 3. The number in parentheses refers to the simulation Scenario.

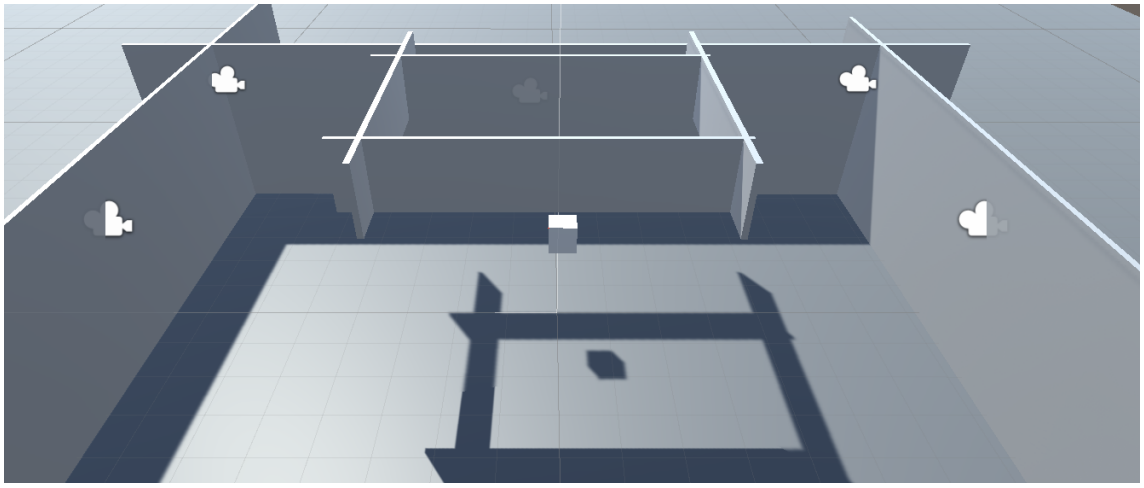

Figure 5.8: Unity setup used for simulating marker occlusion. The cube moves according to Scenario 1 and 2 resulting in temporal occlusion while passing the internal walls.

5.3 Tracking with Occlusion

This section evaluates the capability of the visual-inertial estimator to fill in gaps where the marker is occluded, i.e. it is not visible by any linear detector. The data was generated by setting up a set of internal walls in the Unity simulation environment as shown in Figure 5.8. Again, only the position estimates were considered. Occlusion was simulated for Scenario 1 and 2. Both Scenario 1 and 2 where tested with occlusion.

For Scenario 1 the internal wall was positioned such that all cameras lost visibility for approximately $100\,\mathrm{ms}$ while translating upwards and translating downwards. Though the time where there is limited visibility, i.e. less than three linear detectors capture the marker, was much longer with around $350\,\mathrm{ms}$. Figure 5.9 shows the estimated trajectories. Method 2 was able to approximately recover the true trajectory but slightly drifted off when vision by the cameras was lost. Method 3 was able to properly estimate the position during occlusion without major drifts.

For Scenario 2 the internal wall was positioned such that the marker was not visible by any camera during the upper part of its trajectory. The time for full occlusion was $700\,\mathrm{ms}$ at a time. Though the time of only partial visibility by less than three linear detectors was longer with $830\,\mathrm{ms}$. The estimated trajectories for all three estimation methods can be seen in Figure 5.10. Method 2 failed to compute a usable estimate during occlusion. The solution produced by Method 3 captures the general structure of the true trajectory during occlusion but drifted.

5.4 Discussion

It could be shown that for simple kinematic situations an improved real-time estimate can be obtained by fusing inertial measurements using an incremental fixed-lag smoother based on

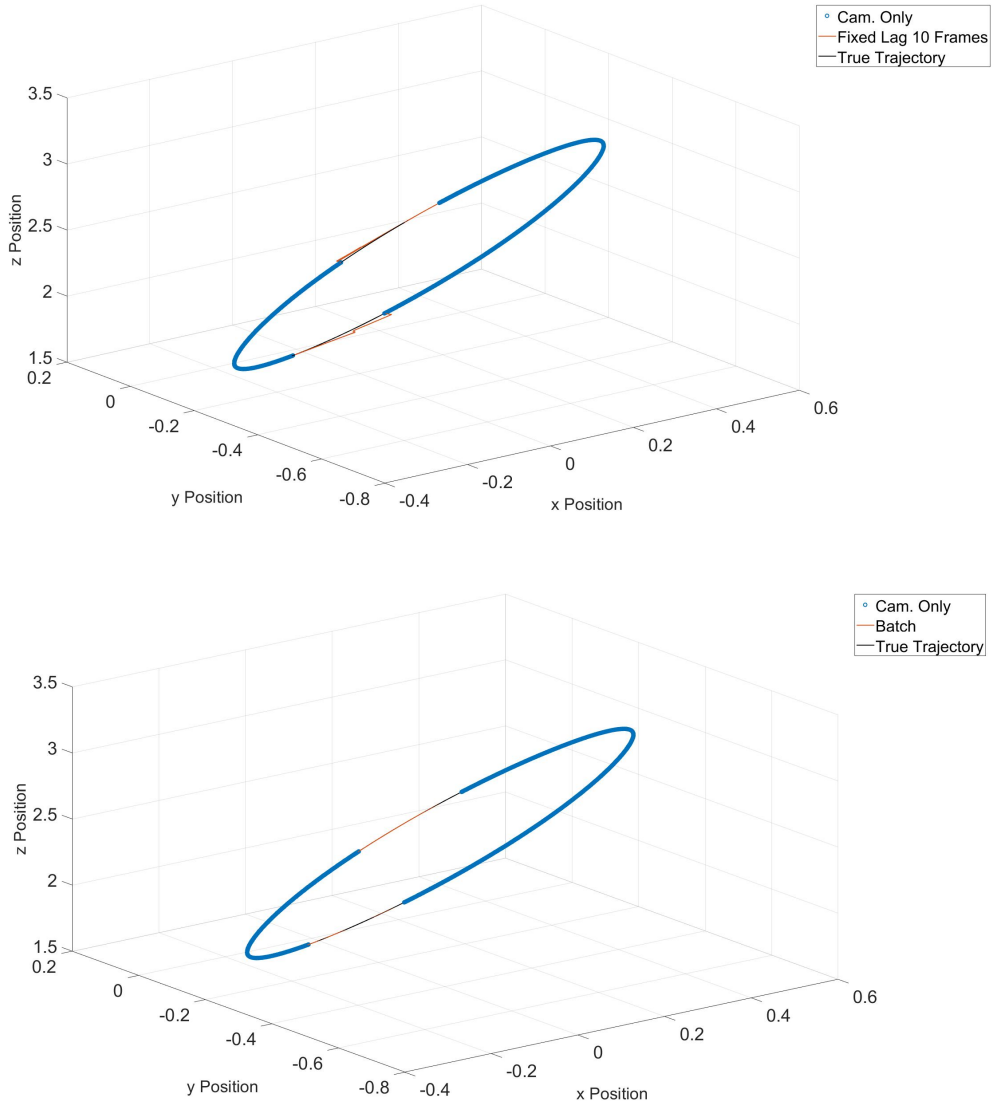

Figure 5.9: Trajectories for Scenario 1 with occlusion. The same simulated data was estimated with both a fixed lag smoother based on iSAM2 with a 10 frames lag and batch optimisation.

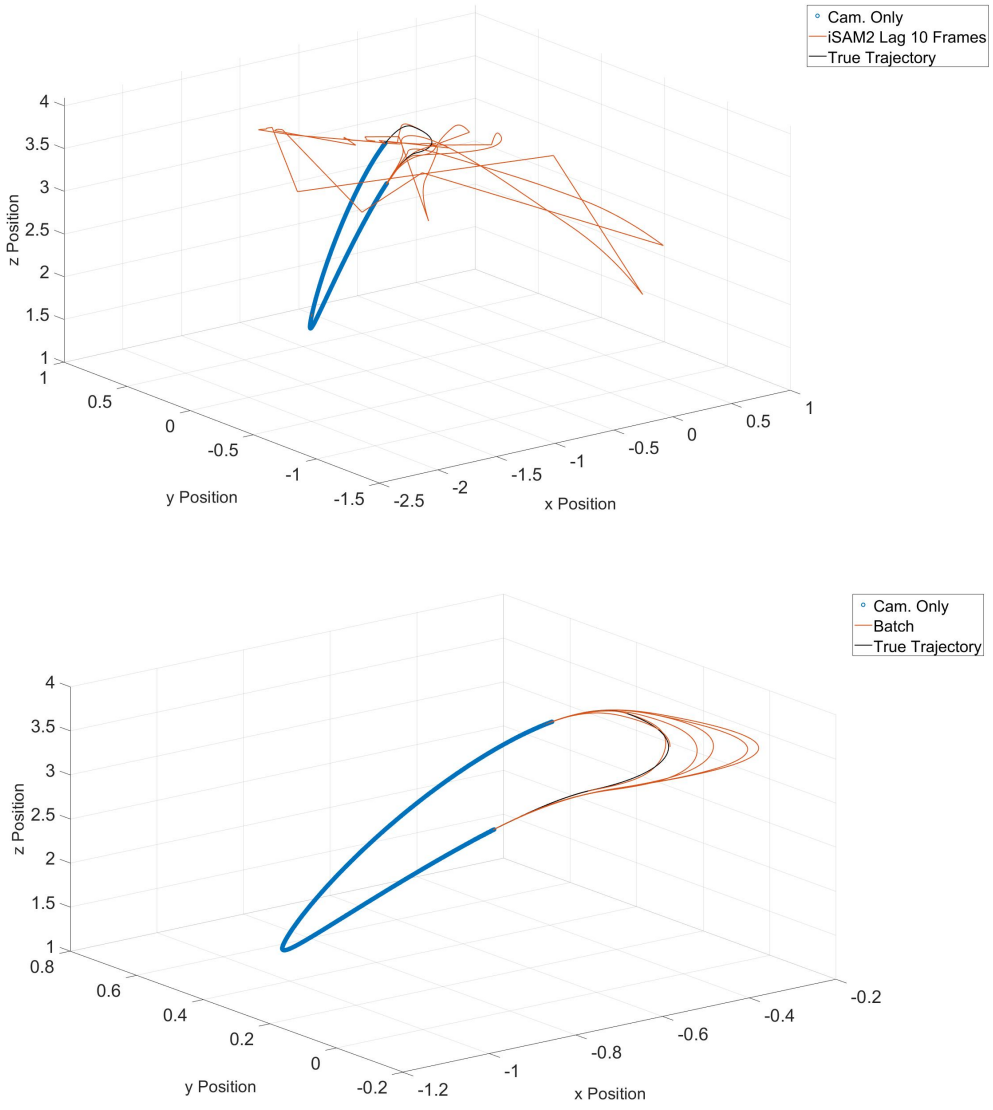

Figure 5.10: Trajectories for Scenario 2 with occlusion. The same simulated data was estimated with both a fixed lag smoother based on iSAM2 with a 10 frames lag and batch optimisation.

iSAM2 . The benefit was lost in more complex environments like Scenario 2. The rapid movements and rotations of Scenario 3 even resulted in a significant degeneration of the estimate compared to camera-only triangulation. This behaviour might either be caused by inaccuracies in the simulation, e.g. quantisation noise in the numerical differentiation of the rotation, or by errors introduced through integration of the inertial measurements. Since the inertial integration (see Section 2.2) assumes constant acceleration and angular velocity in-between measurements, large measurement spikes, as they occur in Scenario 3, cause integration of those spike over longer time periods as they actually occur. This in turn results in large errors in the estimated pose. Now, using this erroneous pose measurement for estimation without increasing the measurement covariance leads to larger estimation errors. Therefore, one solution might be to dynamically adjust the inertial covariance in the estimator depending on the magnitude of measured linear acceleration and angular velocity.

Batch optimisation was able to improve estimates for Scenario 1 and 2. Therefore, it is a viable approach to use visual-inertial data for post-processing of marker position estimates in simple to moderate kinematics environments. Only for demanding situation, like present in Scenario 3, batch optimisation might not be able to provide a benefit. This might be caused by the same reasons explained in the previous paragraph.

As demonstrated in Section 5.3 visual-inertial measurements can be used to fill-in short gaps where no camera measurements are available. Though this highly depends on the total length of the gap since without visual information only integrated inertial measurements are used for estimation which suffer from drift. In the worst case the estimates can become completely unusable, as shown in the first plot of Figure 5.10. Though this specific scenario needs to be investigated since the chaotic behaviour of the solution seems unrealistic given the short time frame of no visibility. On the other hand the batch optimised solution managed to recover the general structure of the trajectory. Nevertheless, the chaotic solution of the incremental fixed-lag smoother is an anomaly and requires further attention.

The improved estimates of batch optimisation compared to real-time incremental fixed-lag smoothing were expected. The full graph used for batch optimisation is much more expressive since it includes all available measurements compared to only the 10 most recent ones used for fixed-lag smoothing. Furthermore, when using a smoother for real-time tracking only the measurements up to the most recent state are available for optimisation. Compare this to post-processing where all measurements are used to optimise all states and it becomes clear why post-processing will generally provide better estimates.

Chapter 6

Conclusion

A full pipeline for simulating visual-inertial measurements of a commercial high performance motion capture system was presented. A visual framework for generating marker trajectories based on the Unity game engine, an easy to use Matlab script for IMU simulation and a minimal dependency C++ library for simulating the motion capture system was developed. The simulation pipeline was then used to generate three testing scenarios with increasingly challenging marker kinematics and trajectories. Three estimation algorithms were implemented: camera-only triangulation, real-time capable incremental fixed lag smoothing and batch optimisation for post-processing. The latter two use factor graphs as the underlying structure for modelling the inference problem. A custom factor graph including a custom factor that uses raw linear detector measurements were developed. The three simulation scenarios and two graph-based estimation algorithms were then used for a first evaluation of visual-inertial motion capture based on graphical models, where camera-only triangulation was used as a baseline. It could be shown that inertial measurements can improve the estimates but may also lead to a less accurate solution for rapid human motion. Furthermore, the possibility of using inertial data to bridge gaps where no visual measurements are available was tested and proven effective for short time periods and moderate marker kinematics.

This work merely acts as a groundwork for further research in the field of high performance visual-inertial motion capture and provides some interesting points for further consideration, some of which are:

- How accurate is the IMU simulation and how can it be improved?
- Can the integration covariance of the IMU factor be dynamically adjusted to improve estimator performance in rapid movements?
- Characterise the estimator performance for full occlusion in more detail.
- Evaluate the estimator performance for situations where there are only one or two linear detector measurements available, i.e. not enough linear detectors for triangulation but more than zero.

- Is it possible to develop a method for automatic calibration of the visual-inertial motion capture system.
- Develop a method for factor graph based rigid body tracking using multiple markers.

Especially the last point is interesting since the method presented in this work only tracks single markers. The markers rotation is only measured by the IMU since the linear detectors can only provide position constraints. But usually more than one marker is attached to a single rigid body like the arm of a human. The markers that are attached to the same rigid body could be used to generate rotation constraints for each other. Furthermore, the marker positions are often just a mean for computing rigid body poses in post-processing. Multiple markers and their IMUs could instead be used in the estimator to directly optimise the pose of the rigid body they are attached to.

In addition to more rigorous simulations real world experiments are also necessary to evaluate the performance of the proposed visual-inertial motion capture system. Unfortunately the ongoing COVID-19 pandemic did not allow for performing such experiments in the laboratory.

Bibliography

[1] A. Menache, *Understanding motion capture for computer animation / Alberto Menache.* Burlington, MA: Morgan kaufmann, 2nd ed. ed., 2011.

[2] D. Titterton and J. Weston, *Strapdown inertial navigation technology*. Institution of Electrical Engineers, 2004.

[3] P. D. Groves, "Principles of GNSS, inertial, and multisensor integrated navigation systems, 2nd edition [book review]," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 2, pp. 26–27, 2015.

[4] T. Lupton and S. Sukkarieh, "Efficient integration of inertial observations into visual SLAM without initialization," *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[5] T. Lupton and S. Sukkarieh, "Visual-inertial-aided navigation for high-dynamic motion in built environments without initial conditions," *IEEE Transactions on Robotics*, vol. 28, no. 1, p. 61–76, 2012.

[6] A. I. Mourikis and S. I. Roumeliotis, "A multi-state constraint kalman filter for vision-aided inertial navigation," *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 2007.

[7] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "IMU preintegration on manifold for efficient visual-inertial maximum-a-posteriori estimation," *Robotics: Science and Systems XI*, 2015.

[8] K. Eckenhoff, P. Geneva, and G. Huang, "Closed-form preintegration methods for graph-based visual–inertial navigation," *The International Journal of Robotics Research*, vol. 38, no. 5, p. 563–586, 2019.

[9] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-manifold preintegration for real-time visual–inertial odometry," *IEEE Transactions on Robotics*, vol. 33, no. 1, p. 1–21, 2017.

[10] K. Eckenhoff, P. Geneva, and G. Huang, *High-Accuracy Preintegration for Visual-Inertial Navigation*, pp. 48–63. Cham: Springer International Publishing, 2020.

[11] F. Dellaert and M. Kaess, "Factor graphs for robot perception," *Foundations and Trends in Robotics*, vol. 6, no. 1-2, p. 1–139, 2017.

[12] J. Crassidis, “Sigma-point kalman filtering for integrated GPS and inertial navigation,” *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2005.

[13] M. Kaess, A. Ranganathan, and F. Dellaert, “iSAM: Incremental smoothing and mapping,” *IEEE Transactions on Robotics*, vol. 24, no. 6, p. 1365–1378, 2008.

[14] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, “iSAM2: Incremental smoothing and mapping using the bayes tree,” *The International Journal of Robotics Research*, vol. 31, no. 2, p. 216–235, 2011.

[15] R. Chartrand, “Numerical differentiation of noisy, nonsmooth data,” *ISRN Applied Mathematics*, vol. 2011, p. 1–11, 2011.

[16] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge: Cambridge University Press, 2004.

[17] V. Indelman, S. Williams, M. Kaess, and F. Dellaert, “Factor graph based incremental smoothing in inertial navigation systems,” in *2012 15th International Conference on Information Fusion*, pp. 2154–2161, 2012.

[18] J. L. Blanco, “A tutorial on SE(3) transformation parameterizations and on-manifold optimization,” tech. rep., University of Malaga, 09 2010.

Appendix A

Unity Scene and Output File Structure

To track a rigid body markers need to be attached to it in the Unity scene. First, an empty GameObject with the name *Markers* needs to be added as a child. The actual markers are added as childs of the *Markers* GameObject as shown in Figure A.1. The markers can theoret-

Figure A.1: Two rigid bodies that have markers attached to them. The *Markers* GameObject holds the individual marker instances.

ically have any colour and shape as long as they are equipped with a collider component. This is required for the raycasting algorithm to detect hits. Also, each tracked rigid body needs to be equipped with a `BodyManager` component which can be used to start animations or bind other behaviour.

The cameras of the motion capture system need to be added as children of an empty GameObject. This parent object acts as a container and is equipped with a *CameraManager* component, which is a custom scripts that handles the ray casting. Theoretically any non-moving GameObject can be used as camera since no features of the Unity camera component are used. The cameras are simply used to obtain their pose and to act as origin for the ray casting.

The output CSV file has the columns

```
frame, t, marker_id, cameras, x_true, y_true, z_true, q0, q1, q2, q3
```

where the q-values are the marker rotation in quaternions and *cameras* are the identifiers of the cameras from which the marker is visible separated by semicolons. The metadata JSON file has the structure:

```
{
    "cameras": [{
      "id": "Camera_1",
      "position": {
         "x": 0.0,
         "y": 2.5,
         "z": 9.5
      },
      "rotation": {
         "q0": -0.028176553547382356,
         "q1": 0.0,
         "q2": -0.9996029734611511,
         "q3": 0.0
      }
   }, ...],
    "markers": ["Marker_1", ...],
    "samplingRate": 1000
}
```

Appendix B

Matlab Code and Output File Structure

All the functionality required for processing the Unity output is implemented in a single class called `PhaseSpaceSim`. First an object is initialised using the input and output directory of the Unity output files. Then the method `readInput()` can be called to parse the CSV and JSON and store their content in the object. Next, `setCameraMetaData(...)` will set some additional metadata required for the C++ PhaseSpaceSim (PSS) library. The kinematics, i.e. linear acceleration and velocity and angular velocity, are computed by the `computeKinematics(...)` method which also fits the smoothing spline to the raw marker trajectories. A simulated IMU can be created using `createImu(...)`. Finally, the method `generateImuMeasurements(...)` will compute the simulated measurements using the created IMU object and it will write the measurements to a CSV file and the metadata to another JSON file. The CSV file has the columns

```
frame,t,markerID,cameras,x,y,z,q0,q1,q2,q3,ax,ay,az,wx,wy,wz,vx,vy,vz
```

which are similar to the input CSV file only that the acceleration a, velocity v and angular velocity w were added. The JSON file has the structure

```
{
    "cameras": [{
      "id": "Camera_1",
      "position": {
         "x": 0.0,
         "y": 2.5,
         "z": 9.5
      },
      "rotation": {
         "q0": -0.028176553547382356,
```

```
         "q1": 0.0,
         "q2": -0.9996029734611511,
         "q3": 0.0
      },
      "fieldOfView": 120,
      "sensorWidth": 0.1,
      "sensorVariance": 1E-6,
      "resolution": 30000,
      "calibratedPosition": {
         "x": 0,
         "y": 2.5,
         "z": 9.5
      }
   }, ...],
    "markers": ["Marker_1", ...],
    "samplingRate": 1000
}
```

where some camera-specific metadata was a added.

Appendix C

IMU Characteristics

The following IMU characteristics were used for Chapter 5:

	Range	Resolution	Bias	Noise Density
x-Accelerometer	$16g$	16 bit	$0.003g$	$1.60 \times 10^{-4} g \frac{1}{\sqrt{\text{Hz}}}$
y-Accelerometer	$16g$	16 bit	$0.005g$	$1.60 \times 10^{-4} g \frac{1}{\sqrt{\text{Hz}}}$
z-Accelerometer	$16g$	16 bit	$-0.004g$	$1.60 \times 10^{-4} g \frac{1}{\sqrt{\text{Hz}}}$
x-Gyroscope	$2000 \frac{\text{deg}}{\text{s}}$	16 bit	$0.12 \frac{\text{deg}}{\text{s}}$	$0.008 \frac{\text{deg}}{\sqrt{\text{Hz}}}$
x-Gyroscope	$2000 \frac{\text{deg}}{\text{s}}$	16 bit	$-0.05 \frac{\text{deg}}{\text{s}}$	$0.008 \frac{\text{deg}}{\sqrt{\text{Hz}}}$
x-Gyroscope	$2000 \frac{\text{deg}}{\text{s}}$	16 bit	$-0.1 \frac{\text{deg}}{\text{s}}$	$0.008 \frac{\text{deg}}{\sqrt{\text{Hz}}}$

Table C.1: IMU characteristics used for simulation.