DEGREE PROJECT, IN COMPUTER SCIENCE AND COMMUNICATION , SECOND LEVEL
STOCKHOLM, SWEDEN 2015

Estimating 3D-trajectories from Monocular Video Sequences

JONAS SKÖLD

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Estimating 3D-trajectories from Monocular Video Sequences

Estimering av 3D-banor från monokulära videosekvenser

JONAS SKÖLD
jonassko@kth.se

Degree Project in Computer Science and Communication (30 ECTS credits)
Degree Programme in Computer Science and Engineering
School of Computer Science and Communication
KTH Royal Institute of Technology, Stockholm, Sweden
Supervisor: Olov Engwall
Examiner: Anders Lansner

June 24, 2015

Abstract

Tracking a moving object and reconstructing its trajectory can be done with a stereo camera system, since the two cameras enable depth vision. However, such a system would not work if one of the cameras fails to detect the object. If that happens, it would be beneficial if the system could still use the functioning camera to make an approximate trajectory reconstruction.

In this study, I have investigated how past observations from a stereo system can be used to recreate trajectories when video from only one of the cameras is available. Several approaches have been implemented and tested, with varying results. The best method was found to be a nearest neighbors-search optimized by a Kalman filter. On a test set with 10000 golf shots, the algorithm was able to create estimations which on average differed around 3.5 meters from the correct trajectory, with better results for trajectories originating close to the camera.

Sammanfattning

Estimering av 3D-banor från monokulära videosekvenser

Att spåra ett objekt i rörelse och rekonstruera dess bana kan göras med ett stereokamerasystem, eftersom de två kamerorna möjliggör djupseende. Ett sådant system skulle dock inte fungera om en av kamerorna misslyckas med att detektera objektet. Om det händer skulle det vara fördelaktigt om systemet ändå kunde använda den fungerande kameran för att göra en approximativ rekonstruktion av banan.

I den här studien har jag undersökt hur tidigare observationer från ett stereosystem kan användas för att rekonstruera banor när video från enbart en av kamerorna är tillgänglig. Ett flertal metoder har implementerats och testats, med varierande resultat. Den bästa metoden visade sig vara en närmaste-grannar-sökning optimerad med ett Kalman-filter. På en testmängd bestående av 10000 golfslag kunde algoritmen skapa uppskattningar som i genomsnitt skiljde sig 3.5 meter från den korrekta banan, med bättre resultat för banor som startat nära kameran.

Contents

Chapter 1

Introduction

The introduction gives an overview and background to the studied problem, and goes on by defining the scientific question which has been investigated, as well as goals of the study. The study is also put into perspective by giving a brief overview of related work.

1.1 Background

Tracking objects in videos and reconstructing their 3-dimensional trajectory is a problem which occurs in many fields, including sports video analysis [6, 26], robot vision [16, 24], surveillance [3] etc. The focus in this thesis is on sports video analysis and the reconstruction of the flight trajectory of a ball. In particular, a system for tracking golf balls hit from a driving range was used as the target system in the study.

Finding the 3-dimensional trajectory of a flying object can be done by filming the object with two calibrated cameras, finding its position in the camera feeds, and building a 3D-model from the observations. However, if one of the cameras does not detect the object at all, regular stereo vision algorithms fail. The reason that a camera fails to detect an object could be many, for instance poor light conditions, occlusion of the object, temporary camera failure etc. If this happens, it would be beneficial if the system could estimate the 3D-trajectory using only the observations from one camera.

In this study, I have investigated how past observations from a stereo system can be used to recreate trajectories when video from only one of the cameras is available. Multiple methods for solving the task have been implemented and compared.

1.2 Question

How can previous 3D-trajectory observations from a two-camera system be used to adequately estimate new 3D-trajectories using observations from only one of the cameras?

1.3 Goals

The stretch goal of the study was to create completely accurate 3D-estimations. However, the utility of the studied target system would be improved even at much lower accuracy. Therefore, the initial goal was to find the correct starting point of each ball path, so that it is associated with, and displayed for, the correct player on the driving range. Another basic goal was that the estimated 3D-trajectory should roughly look like the real shot, convincing the player that it is his/her shot that has been reconstructed. Moreover, the 3D-trajectory should be displayed almost instantly to the player, so the estimation had to be at least sub-second fast.

1.4 Relation to the existing system

The system that this study was tested on is used for tracking golf shots on a driving range. It consists of two calibrated fixed cameras positioned on each side of the shooting bays, filming the range. The system is able to detect a flying ball and produce one 2D-trajectory per camera, as well as reconstructing its 3D-trajectory by triangulating the two 2D-trajectories. The task in this thesis was to take only one of the 2D-trajectories and produce an approximate 3D-trajectory, with the help of past observations. All image processing is already done by the system, which simplifies the problem of this study to just converting a sequence of 2D-coordinates into a sequence of 3D-coordinates.

1.5 Related work

To my knowledge, there are no published studies where old observations from a stereo camera system have been utilized to approximate 3D-trajectories from a single video camera. However, similar studies have been made which either solve a related problem or use another technique. Some of the most relevant studies are briefly explained in this section.

1.5.1 Sports video analysis

Tracking and 3D-reconstruction of the movement of the ball in various sports is getting more popular. Possible use cases are automated referee systems [22], registration for statistical purposes [6], improving TV-broadcasts with augmented graphics etc. Kumar et al. [22] have created a multi-camera system with a physics model which tracks the trajectory of a tennis ball. The system uses 9 calibrated cameras and triangulates the position of the ball at specific times, and uses a physics model to obtain a smooth trajectory. Chen et al. [6] use a single camera for tracking basketball shots. By using known properties of the court, the depth of the image can be calculated, and an accurate 3D-model of shots can be obtained. Schnyder et al. [26] are also using knowledge about the football field to create a depth map of a monocular video sequence of a football game. The K-Zone system described by Guéziec [14] tracks a pitched baseball in three cameras and uses a Kalman filter to calculate the trajectory of the ball.

1.5.2 Mathematical monocular reconstruction of trajectory

Reconstruction of a trajectory from a single video sequence has been studied in several contexts. Because reconstruction of 3D-coordinates from 2D-coordinates is an ill-posed problem in the general case, constraints or assumptions about the motion must be used to obtain a unique solution. Different constraints and assumptions on the motion have been used to mathematically compute an estimation of the trajectory. Xu et al. [31] assume that the tracked object has a constant velocity between video frames, which lets them approximate the shape of the motion. However, the scale of the trajectory cannot be calculated in this way, so an absolute depth estimation algorithm is needed as well. Ribnick et al. [24] aim to localize projectiles in the context of making a robot catch a ball. They assume that the only external force on the ball is gravity, and prove that observations of the trajectory by one camera at three or four points in time are sufficient to calculate the complete trajectory of the ball.

1.5.3 Learning depth in images and videos

Machine learning has previously been used to estimate the depth map of images and videos. Konrad et al. [21] have implemented two distinct methods for learning image depth from a set of images with known depth maps. The first approach is to learn a function from local features of a pixel to the depth at that pixel. The pixel features used were the color and position of pixels, as well as motion in the case of video. Their second approach is a global approach where they find the most similar images and combine the corresponding depth maps to get a depth map approximation. Saxena et al. [25] utilize a Markov Random Field trained by supervised learning to infer location and orientation of patches in an image. Konda and Memisevic [20] are estimating both depth and motion in a video by learning correspondences between camera pairs and video frame pairs using a synchrony autoencoder with an energy model.

Chapter 2

Theory

Theory that is relevant to this study, and that is needed to understand the following chapters in the report, is presented in this chapter.

2.1 Mono and stereo vision

In order to explain the problem of 3D-reconstruction from a single camera, and what differs from a stereo camera system, this section will cover the basics of mono and stereo vision - the camera model and epipolar geometry.

2.1.1 Camera model

A camera is mapping points in the 3-dimensional real world to points on a 2-dimensional camera plane. In its most simple form, it can be described by the pinhole-model. In that model, there is a camera center, or center of projection, and an image plane positioned at distance f from the camera center. The mapping of a point in 3D to a point in 2D is done by finding the intersection between the image plane and a straight line drawn from the 3D-point to the center of projection. If the 3D-point is given by $(X, Y, Z)^\intercal$, then

$$\left(\frac{fX}{Z}, \frac{fY}{Z}\right)^\intercal \tag{2.1.1}$$

will be the corresponding 2D-point [15].

Consider equation 2.1.2, with c being an arbitrary scalar.

$$c\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \mapsto \left(\frac{cfX}{cZ}, \frac{cfY}{cZ}\right)^\intercal = \left(\frac{fX}{Z}, \frac{fY}{Z}\right)^\intercal \tag{2.1.2}$$

Note that any point which lies on the line $c(X, Y, Z)^\intercal$ will map to the same coordinate in the camera plane. This is where the depth of the scene is lost.

The 2D-coordinates given in 2.1.1 are valid if the image plane has its coordinate origin in the principal point (the image center). If this is not the case, e.g. if the origin is at the corner of the image, an offset must be added to the 2D-point. With p_x being the offset on the x-axis and p_y the y-axis offset, and with the function denoted in matrix format, the new mapping becomes

$$\begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{pmatrix} = \begin{bmatrix} f & & p_x & 0 \\ & f & p_y & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.1.3}$$

where

$$K = \begin{bmatrix} f & & p_x \\ & f & p_y \\ & & 1 \end{bmatrix} \tag{2.1.4}$$

is called the *camera calibration matrix* or *intrinsic matrix* [15]. The points are here given in homogeneous coordinates.

Equation 2.1.3 assumes the point is given in the same coordinate system as the camera. A final addition to the function has to be made if the coordinate system of the real world differs from that of the camera. The relation between the camera world coordinate system and the real world coordinate system is identified by a 3×1 translation vector $\mathbf{t}$ and a 3×3 rotation matrix R respectively. With $(\hat{X}, \hat{Y}, \hat{Z})^\intercal$ denoting a point in real world and $(X, Y, Z)^\intercal$ the same point in the camera world, the mapping between the two systems is

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{pmatrix} \hat{X} \\ \hat{Y} \\ \hat{Z} \\ 1 \end{pmatrix} \tag{2.1.5}$$

where the mapping matrix is called the *extrinsic matrix.* In summary, the projection of the point $\mathbf{X}$ to the camera plane point $\mathbf{x}$ is defined by the function

$$\mathbf{x} = K[R|t]\mathbf{X} \tag{2.1.6}$$

where $\mathbf{x}$ and $\mathbf{X}$ are given in homogeneous coordinates [15].

See also figure 2.1 for a visual presentation of the camera model.

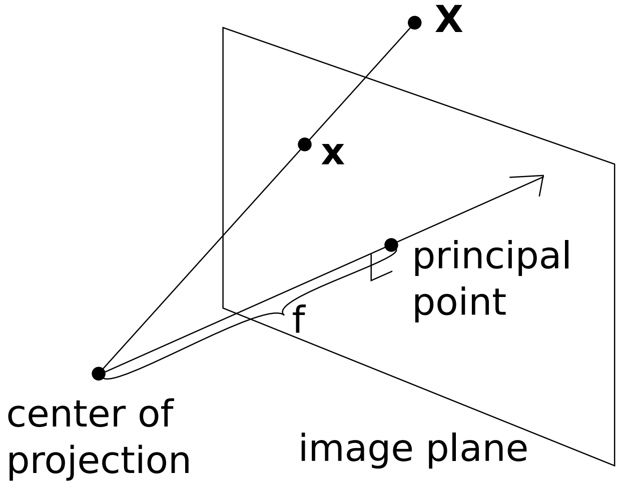


Figure 2.1: A model of the pinhole camera.

2.1.2 Epipolar geometry

Epipolar geometry is explaining how two cameras are related geometrically. It can be described algebraically by a 3×3 matrix F called the *fundamental matrix* [15]. For a point $\mathbf{X}$ in real space which is projected as $\mathbf{x}$ in one camera and $\mathbf{x}'$ in the other, the relation

$$\mathbf{x}'^{\mathsf{T}} F \mathbf{x} = 0 \tag{2.1.7}$$

holds.

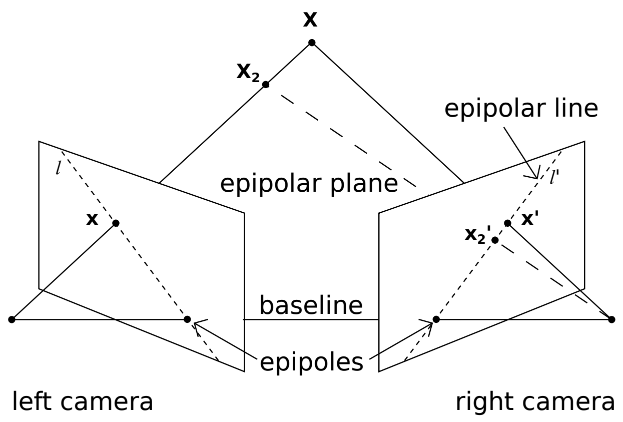


Figure 2.2: An illustration of epipolar geometry

When describing the relation between two cameras, the line between the two camera centers is called the *baseline.* The baseline together with a point $\mathbf{X}$ in the real world define a plane which is called the *epipolar plane.* The epipolar plane intersects the image planes, and this intersection is a line called the *epipolar line.*

The point where the baseline intersects an image plane is called an *epipole*. A point $\mathbf{x}$ on one camera plane will lie on the epipolar line l' on the other camera plane, given by

$$l' = F\mathbf{x} \tag{2.1.8}$$

This means that by knowing the position of a point in the first camera, it is possible to calculate the epipolar line on which the point will lie on in the second camera. However, it is not possible to find the exact position on that line. Consider the two real world points $\mathbf{X}$ and $\mathbf{X}_2$ in figure 2.2, which are projected to the same point $\mathbf{x}$ in the first camera. Their corresponding projections in the second camera, $\mathbf{x}'$ and $\mathbf{x}'_2$, both lie on the epipolar line, but at different locations. Conversely, if both $\mathbf{x}$ and $\mathbf{x}'$ are known, it is possible to calculate the exact position of point $\mathbf{X}$, a method known as triangulation [15].

2.2 Object motion in monocular video sequence

In a video sequence, it is possible to detect how objects move relative to the camera based on their position in consecutive frames. This can be used for calculating optical flow as well as structure from motion. Optical flow describes the projected 2-dimensional motion between two video frames. It can be used for 3D-scene reconstruction, motion detection, video compression etc. [4]. Structure from motion is a method for reconstructing the structure of a scene from images from a moving camera.

Without extra information about the motion, e.g. speed, acceleration, direction etc., it is not possible to determine how an object has moved orthogonally to the camera plane between two video frames. Likewise, determining how far away from the camera the object is cannot be done, unless the real size of the object is known. Additional information or assumptions are therefore necessary to calculate the motion [9].

2.2.1 Global motion

A global motion model assumes that an object moves according to some known function, like a parabola or straight line. If the projection matrix of the camera is known, the problem is reduced to finding the parameters of the motion function which best fits the observations [2, 24].

A common global motion assumption is that flying objects are only affected by gravity and thus move in a parabola. Air resistance and other external forces are omitted. Velocity on the x- and y-axes are hence constant, and velocity on the z-axis is altered by the fixed acceleration caused by gravity. This model is used by Ribnick et al. [24] for reconstructing the trajectories of projectiles.

Avidan and Shashua [2] are able to reconstruct the trajectories of objects moving on straight lines or on conic sections. Their method needs five observations to uniquely determine a straight line trajectory, and nine observations for an arbitrarily shaped conic section. In their study, they use a moving camera where the camera motion is known.

2.2.2 Local motion

A drawback with assuming that an object follows a specific global model is that other motions cannot be handled. An alternative is to only assume constraints which hold locally, i.e. for a short time period like two or three consecutive video frames. One possible local constraint, which is used in the work of Xu et al. [31], is that an object moves with constant velocity over three consecutive video frames. They argue that such a constraint is probable to hold in a general setting, especially if the frame rate of the camera is high enough. Using this constraint, they are able to compute the relative motion of the object between frames. In other words, it is possible to calculate the distance from the object to the camera, relative to the distance at an adjacent observation. Therefore, if they know the exact real-world position of the object at one point in time, they can infer the object's position in the rest of the frames, for any type of motion.

The basic idea of the algorithm is to compare the distance of two consecutive trajectory segments. If the local motion constraint holds, the two real world distances are equally long. If the object has moved in parallel with the image plane, the two projected distances would also be equal. However, perspective effects in the camera make closer objects appear larger. In the case of trajectories, it means that if a projected trajectory segment is larger than another, the real segment must be closer to the camera. For instance, if the second of two consecutive projected segments is longer than the first, the object must have moved towards the camera. See illustration in figure 2.3.

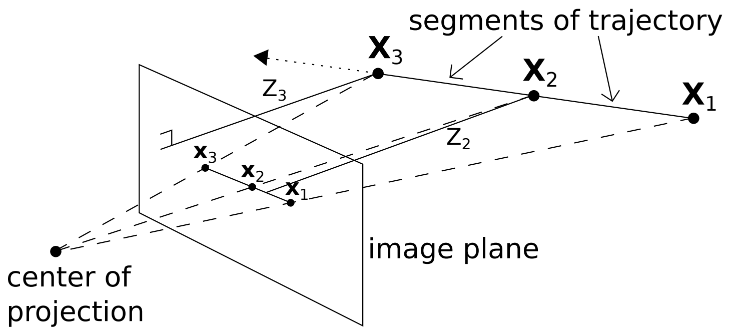


Figure 2.3: Illustration of the idea behind local motion model. The segment between $\mathbf{X}_1$ and $\mathbf{X}_2$ is equally long as the segment between $\mathbf{X}_2$ and $\mathbf{X}_3$. However, since the object is moving towards the camera, the projected segment $\mathbf{x}_1$-$\mathbf{x}_2$ is shorter than $\mathbf{x}_2$-$\mathbf{x}_3$.

By measuring how much longer the projected segment is, it is possible to calculate how much closer the object is at the end of the second segment compared to at the end of the first. The distances to the camera from points 2 and 3, denoted Z_2 and Z_3, are related by equation 2.2.1 [31]. Here, x_i is the x-coordinate of the projected point

$\mathbf{x}_i$, and Z_i is the camera space z-coordinate of the point $\mathbf{X}_i$. Using the y-coordinate of the projected points would give the same result.

$$Z_3 = \frac{2(x_2 - x_1)}{x_3 - x_1} Z_2 \tag{2.2.1}$$

The constant local motion constraint is closely related to inertia, or motion smoothness. Newton's first law of motion says that a moving object is resistant to change its motion, i.e. speed and direction. If the object is not exposed to a sudden external force, it will not change direction abruptly. This can be used to rule out 3D-trajectories that project to the observed 2D-trajectory but does not behave according to Newton's first law.

2.3 Time series

The trajectory of an object observed in a video camera can be represented as a temporally sorted sequence of 2D-coordinates. The time interval between each point is constant and is determined by the frame rate of the camera. This representation is basically a time series, which implies that theories about time series could be used in this context as well. In particular, comparison and prediction of time series are of interest in this study.

2.3.1 Trajectory similarity

Measuring how similar two trajectories are is needed to evaluate the performance of the final system. Moreover, algorithms such as k-nearest neighbors could be used to solve the overall problem, in which case a similarity measure is also needed. The similarity measure of 2D-trajectories should in this case preferably capture the similarity of their respective 3D-trajectories. Not only should the structure of the trajectories be relevant, but the measurement should preferably also take the physical distance between the trajectories into account, in order to say something about the (relative) depth of the trajectories.

Euclidean distance

Euclidean distance is a well-known similarity measure in many areas where a metric distance is wanted. Advantages of the measure is that it is conceptually simple, making it intuitive to analyze as well as easy to implement. The calculation is also linear in the number of sequence points, which enables comparisons of large datasets [28]. A drawback of the method is that it requires the compared sequences to have the same number of sample points, since it compares the sequences point by point [33].

Dynamic time warping

Dynamic Time Warping is a common similarity measure for time series and trajectories. Its main feature is that it is insensitive to variations in speed between two sequences. In the case of trajectories, it ignores the speed by which the object has moved along a path, and only considers the spatial aspect [5]. This is not advantageous in the case of comparing 2D-projections of ball trajectories. In this case, speed is a relevant factor of the trajectory. Two paths with similar 2D-projections may in fact be far away from each other if the temporal length of the sequences are dissimilar.

Longest common subsequence

Longest common subsequence is another similarity measure for sequential data. It compares two trajectories based on how many element pairs from the two sequences are within a certain distance from each other. It is less sensitive to noise compared to dynamic time warping, since it does not require each element to be matched [5].

2.3.2 Time series and machine learning

An alternative to computing the 3D-trajectory based on observations and motion assumptions is to use machine learning to find a mapping from 2D to 3D.

Time series differ from input to regular machine learning algorithms since the input elements have a natural temporal order. Algorithms which take fixed size inputs could be used for time series data if the time series sequence is split up into individual points. However, the dependency between the points are then lost, and the algorithm has less information to learn from.

The machine learning task related to time series that is of interest in this study is to estimate a sequence based on another sequence. In this case, the goal is to estimate the 3D-trajectory based on its 2D-projection. The task can be seen as learning a direct mapping from the 2D-trajectory to the 3D-trajectory. However, another way to look at it is to see the 3D-trajectory as a hidden state sequence which is observed through its 2D-projection. The internal state, or the position of the ball in the real world, depends on its previous state by physical properties. In turn, the observations depend on the internal state by projection geometry. Both the internal state and the observations consist of multiple continuous variables. This setup is often analyzed using Kalman filters [29].

Kalman filter

The Kalman filter was invented in 1960 and has since been used for analyzing and predicting time series. It is able to estimate past and future states of a process based on past observations of the process and an approximate model of the nature of the process. It is a recursive algorithm which step by step finds the best fit of the internal states [29].

The Kalman filter has a model which describes how the process is changing from one step to the next. The model is not exact, but rather a stochastic process with Gaussian noise. The state transition from one time instance to the next is determined by the equation

$$x_t = Ax_{t-1} + w_{t-1} \tag{2.3.1}$$

where A determines the linear transition from state $t-1$ to t and w_{t-1} is the noise at time $t-1$. Both A and w can vary over time or stay the same. The second equation that affects the Kalman filter is the measurement equation

$$y_t = Hx_t + v_t \tag{2.3.2}$$

where y_t is the observed values at time t, H is the linear mapping between the internal state and measurements, and v_t is the measurement noise at time t. As with the transition equation, both H and v can vary over time or stay constant [23, 29].

The process of estimating a sequence starts with a given probability distribution function of the initial state, the prior, which must be determined externally and is usually Gaussian. In each following step of the sequence, the Kalman filter approximates the new state using the transition equation 2.3.1. It then improves the state approximation by comparing the expected measurement with the actual (noisy) observation [29]. The result is a probability distribution function, the posterior, which is Gaussian just like the prior [7].

Like the initial state, the transition and measurement equations have to be estimated if they are not known. In this study, the only property known beforehand is the measurement matrix H, which is simply the projection matrix of the camera. The transition matrix A must be estimated, which could either be done by hand using some assumed physical properties, or learned from past data. The two noise parameters could be learned from past data or determined experimentally.

A drawback with the Kalman filter is that it can only handle linear time series. The Extended Kalman filter solves this, by first linearizing the non-linear sequence model in a way similar to Taylor series expansion [29]. Because of the linearization, the extended Kalman filter only gives a Gaussian approximation of the posterior distribution [7].

Recurrent neural network

Artificial neural networks have been used in a variety of machine learning tasks with good results [27]. They mimic the way the human brain works in order to be able to learn complex functions. An artificial neural network consists of a connected set of nodes forming a network. Some of the nodes take input, others produce output, and the rest are so called hidden nodes. A common way to arrange the nodes is in layers, with one input layer, one or more hidden layers and one output layer. The layers are often connected with weighted directed links to form a directed acyclic graph starting in the input layer and ending in the output layer, in which case the network is called feedforward. Each node perform a computation of the weighted inputs from connected nodes, and send its result to subsequent nodes. In a supervised learning context, the output from the final layer is then compared to ground truth, and the weights between the nodes is updated in a process called backpropagation. This is what makes the network able to learn from data [8].

Unfortunately, standard neural networks are inadequate to work with time series since they assume fixed size input [27]. A special type of network that can handle sequences is the Recurrent Neural Network (RNN). An RNN contrasts to feedforward networks in that it can have connections in arbitrary directions between its units, and not just forwards. The effect is that the network keeps an internal state, or memory, that can learn temporal dependencies of the input. This makes them more suitable for dynamic input like time series [11].

There are several types of recurrent neural networks. A few interesting variants are briefly explained here.

Regular recurrent nets with backpropagation suffer from a potentially very long learning time because of decaying error backflow. Hochreiter and Schmidhuber [17] proposed a novel network called Long Short-Term Memory Network which overcomes that problem. It uses a gradient descent method and has fixed error weights through special internal units called memory cells. As a result, it has a longer short-term memory and can handle long time-lags in sequences. The method has been proven to work well in e.g. sequence-to-sequence-mapping, speech recognition and similar tasks [12, 27].

Jordan networks and Elman networks are two versions of a simple recurrent network which have memory cells, called context cells. The Elman net is based on a feedforward three-layered net with an input layer, a hidden layer and an output layer. In addition to this, a group of context cells are connected to the hidden layer, where each hidden node is connected to one context cell, and the weights of each of these connections are 1. In effect, the context nodes save the state of the hidden nodes and send it back to the hidden layer in the next iteration [8]. Jordan nets instead send the output layer state to the context nodes, and the context nodes are themselves a recurrent net [8].

Nearest neighbors

A possible approach to finding a good approximation of the 3D-trajectory is to compare the 2D-trajectory with all past 2D-trajectories and find the most similar ones (the nearest neighbors). Their triangulated 3D-trajectories could then be combined to create an approximation. Performing a nearest neighbor search on all old trajectories would be too time-consuming if the dataset is large. A way to prune the set of trajectories to compare against is therefore needed. A regular similarity search can be performed in a reasonable time by reducing the complete set of trajectories to a subset of small size, which contain the k nearest neighbors with a high probability.

One approach to quickly finding similar trajectories is to perform an approximative k-nearest neighbors query using locality sensitive hashing (LSH) [13]. LSH is a dimensionality reduction technique which hashes input into buckets. Input items which are similar according to some similarity measure has a high probability of being hashed to the same bucket. One bucket will therefore contain similar items, or in other words, an approximate solution to the k-nearest neighbors problem. A problem with the standard LSH implementation is that it assumes fixed size vectors as input, meaning it cannot be used for variable-length time series [18]. Kale et al. [18] suggest that a new kernelized version of LSH can be used to handle this shortcoming.

Other techniques are based on indexing and include transforming the trajectories into a format more suitable for indexing. Two popular transformations are Discrete Fourier Transform (DFT) and Discrete Wavelets Transform (DWT). DFT transforms segments of a time series to a set of frequencies which together approximate that segment [30]. The first few frequency coefficients are enough to get a rough approximation of the time series, and these coordinates can be used in an index. DWT is similar to the DFT, but the base function is not a sinus function, but an orthonormal wavelet function like the Haar Wavelet [30].

Indexes of the transformed sequences are commonly stored with multidimensional R-trees [1]. A problem of having this type of index is that they can only index sequences of the same length [10]. In order to support variable length sequences, Faloutsos et al. [10] suggest creating a multiresolution index. The idea is that several indexes are created, for different fixed sequence lengths, and that queries can be split up into segments which are then queried individually.

Chapter 3

Implementation

The implementation chapter describes the work that has been done in this study. It begins by motivating the choice of method and then describes how the chosen methods have been implemented.

3.1 Choice of methods

In order to determine how known trajectories can be used to estimate new ones, this study compared a few possible approaches. Four distinct methods, plus two optimization algorithms, have been implemented and tested. The methods have been chosen because they have been successfully used to solve similar problems.

The first approach was to use the nearest neighbors algorithm on the entire trajectory. To find the 3D-trajectory of an observed shot, the most similar previous 2D-observations are first found using the k-nearest neighbors algorithm. Their known 3D-trajectories are then combined to produce an estimation of the new trajectory. This is perhaps the most straightforward method, but it is also time consuming, as explained in the theory chapter. The fact that entire, real, trajectories are used to create the estimation ought to keep estimations reasonably realistic, although achieving the exact shape of the trajectory is dependent on that there are previous shots with similar shape.

The second approach was a mathematical or model-based approach, in which the 3D-trajectory is calculated based on 2D-observations and physical constraints. Wind, spin and other factors play a big part in the motion of a ball, probably making a universal global motion model too approximative. The algorithm assuming constant local motion, explained in section 2.2.2, was therefore chosen. However, the scale of the trajectory cannot be calculated with the local motion algorithm, so the method was complemented by nearest neighbors as a depth estimation algorithm. The motion constraints method does not use any old trajectory observations, apart from the depth estimation step, but it is interesting as a comparison.

The third approach was to use a recurrent neural network to approximate the trajectories, a technique that has worked well on other time series problems. The network generates new trajectories after learning a mapping from 2D- to 3D-points, which is an interesting contrast to the nearest neighbors algorithm, which only combines old trajectories.

The fourth approach was to do a state space estimation with a Kalman filter. The Kalman filter estimated the state of the object, i.e. the position, velocity and acceleration on the x-, y-, and z-axes, based on a physics model and the 2D-observations. The parameters of the filter were selected experimentally to obtain realistic 3D-trajectories. The Kalman method also requires an estimate of the initial state, which was again obtained with the nearest neighbors algorithm on the start of the trajectory.

A special type of Kalman filter for optimizing approximations was also tested. That Kalman filter improved approximations by using both the already estimated 3D-points as well as the original 2D-observations. In addition to this, a Kalman filter using only the estimated 3D-points was also tested to improve output from the local motion model and the recurrent neural network.

3.2 Nearest neighbors

The nearest neighbors algorithm was implemented in two variants. The first was used to approximate a full 3D-trajectory, and the other to approximate the start of the 3D-trajectory, to be used in combination with other algorithms.

3.2.1 Full trajectory

The nearest neighbors algorithm for estimating full trajectories was implemented in a rather standard way, searching through all known trajectories and keeping a list of the k most similar ones. When a single 2D-trajectory is given as input, the algorithm first finds the most similar previous 2D-trajectories. When the k most similar 2D-trajectories are found, their already known 3D-trajectories are combined to produce an estimate of the current shot's 3D-trajectory.

The combination was done in three different ways. The first was a simple pointwise average, where each of the neighbors affected the output equally. The second was a weighted average, where closer trajectories were given higher weights. The third was a pointwise weighted average, where weights were given per point depending on how close it was to the corresponding observation.

Not all trajectories have the same number of observations, so the similarity measure must work with trajectories of different lengths. The implemented nearest neighbors algorithm therefore compared only the first n points of the trajectories, where n is the number of observations in the shorter of the two trajectories. The sum of the differences was then divided by n to get an average difference per point. That way, trajectories of different lengths could be compared in the same unit.

Several different ways of measuring distance between two points were used. The most trivial was to use Euclidean distance, i.e. the shortest distance between the two points measured in pixels. In addition to position, other features like velocity, acceleration, size and direction were experimented with. The distance measure was then the Euclidean distance in the complete feature space, i.e. the square root of the sum of the squared individual features, e.g. $\sqrt{p_x^2 + p_y^2 + v_x^2 + v_y^2}$ in the case of position and velocity. To make sure that each feature had the same impact on the distance measure, all features were standardized to have zero mean and unit variance.

In order to increase the performance of the algorithm, some pruning of the search space was performed. Speed was increased by ignoring candidates whose first points were not similar enough to the first points of the input trajectory. The algorithm also only looked at neighbor trajectories that were at least a certain length of the input trajectory. This was done to make sure that as much as possible of the shape of the trajectory was captured. Since the estimated trajectory is an average of the closest neighbors, and the length of the output trajectory is as long as the shortest neighbor, a short neighbor would result in an unnecessarily short estimated trajectory.

3.2.2 Beginning of trajectory

The nearest neighbors algorithm is not only useful by itself, but it can also be used as a support algorithm to other methods. One such use case is to estimate the position of the first point of a trajectory, and use other algorithms to complete the trajectory.

This modified version of the algorithm was implemented in this study and used for initial position estimation for the local motion model and the Kalman model. The algorithm compared the first part of a trajectory to the same part of every known trajectory to find the k most similar ones. The only difference in this step to the full version was that the number of points to compare was fixed, instead of comparing all points. How many points to look at to get the best accuracy was determined experimentally.

After finding the most similar 2D-trajectories, the algorithm would then calculate the average 3D-position of the first point in the neighbors, which could then be used as input to other algorithms which require an estimation of the initial state.

3.3 Local motion model

The second implemented method used a local motion model to calculate the trajectory. As described in section 2.2.2, the model was an assumption that the moving object does not change its speed or direction between three consecutive frames in a video.

A prerequisite for the local motion algorithm is to know the camera space coordinates of the first 3D-point in the trajectory. The coordinates could be approximated with for instance the nearest neighbors algorithm. The algorithm then uses equation 2.2.1 together with the z-coordinate of the first point to calculate the distance to the camera of each point in the trajectory. When knowing the x and y-coordinates of the projected points, together with the camera space z-coordinates of the 3D-points, the x and y-coordinates of the 3D-points can also be calculated. Finally, the camera space 3D-points are translated to real world coordinates.

3.4 Recurrent neural network

To evaluate the neural network approach, an Elman network was used. An Elman network has an input layer, a hidden layer and an output layer, as well as context cells connected to the hidden layer. As a basis, the input layer had one neuron per point-feature, i.e. position, velocity and acceleration on x- and y-axes, as well as size. The output layer had only one neuron, which represented the distance from the camera to the observed point. It would also be possible to predict all three coordinates of the 3D-point, but the x- and y-coordinates can be calculated from the z-coordinate and the image coordinates, and predicting a single value is easier.

A few modifications to the standard Elman network was implemented. Firstly, the network was made into a nonlinear autoregressive exogeneous network (NARX). The input to the network consisted of both 2D-observations (exogeneous) and the previous 3D-point (autoregressive). The actual 3D-points were used in the training phase, whereas the previously predicted points were used in later steps in the prediction phase.

To boost the memory aspect of the network, a sliding window approach was used for the input. That means that instead of only inputting the last observation, the n latest observations were inputted in each step, both for the 2D-observations as well as for the 3D-points.

The structure of the network is shown in figure 3.1

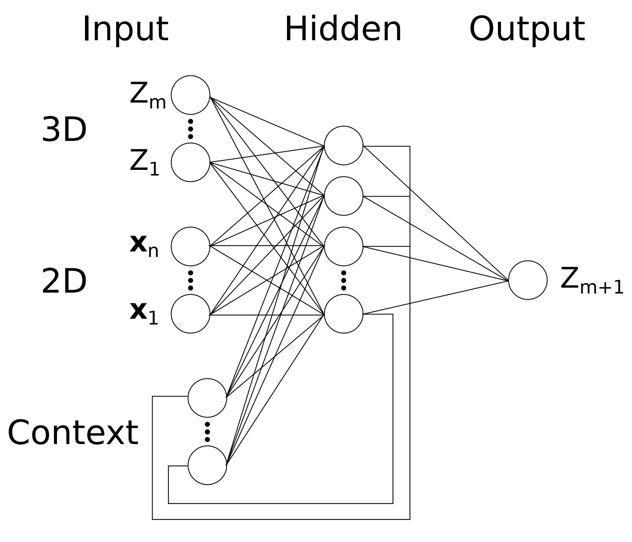


Figure 3.1: A sketch of the structure of the implemented neural network

3.5 Kalman filter

The Kalman filter was used in three different ways: directly estimating a 3D-trajectory from 2D-observations, filtering out noise on an estimated 3D-trajectory, and improving an estimated 3D-trajectory with its 2D-observations.

3.5.1 Estimating 3D-trajectory

The first use case of the Kalman filter was to estimate the 3D-trajectory directly from the 2D-observations using an Extended Kalman filter. The idea was to estimate the state (position, velocity and acceleration on x-, y- and z-axes) of the ball for each point on the trajectory using only the projected observations of the state.

The process model that the Kalman filter used was a simple linear physics model. The ball was assumed to have an initial position, velocity, and acceleration, and the acceleration was assumed to be constant, except for some noise. Position and velocity changed between two observations t and $t+1$ according to physical laws, namely

$$p_{t+1} = p_t + v_t\Delta t + 0.5a_t\Delta t^2 \tag{3.5.1}$$

$$v_{t+1} = v_t + a_t\Delta t \tag{3.5.2}$$

Initial position and velocity was measured or approximated for each shot. However, the initial acceleration was set equal for each shot. Downwards acceleration was set to the gravity constant 9.8 and the horizontal deceleration was set to $5m/s^2$, a value that was found experimentally and is approximating the drag on the ball.

The measurements model was a bit more complex. The camera projection function was used as the measurement function. This function is however not linear, so a regular Kalman filter could not be used. Instead, an Extended Kalman filter was implemented, as explained by Yu et al. [32]. In addition to the projection function for the 2D-measurements, the Extended Kalman filter also uses the Jacobian of the projection function in the state update calculations. The Jacobian matrix has two rows, one for each observation variable, and nine columns, one for each state variable. Since the observations only depend on the positions of the object and not its movement, only the first three columns are non-zero.

A Kalman filter needs to start with an estimation of the initial state. Therefore, the method is not completely standalone. During testing, two ways of getting the initial state estimate was used. The first one simply took the known initial state from the ground truth trajectory. This state would of course not be available in a real situation, but was used in this study to investigate the optimal performance of the Kalman filter. The second way of finding the start state was with a nearest neighbors search on the first part of the trajectory, as explained above. This estimate is worse but can be used on unknown trajectories, and was used to see how the Kalman filter could perform in real situations, and how resistant the method was to noisy initial estimates.

One problem with the often used update algorithms for the Kalman filter is that the covariance matrix might lose its symmetry and positive definiteness properties due to rounding errors [19]. This occurred when using the test data in this study, resulting in a zigzag pattern at the end of the predicted trajectory (see figure 4.5d). The solution to this was to use the original function for the covariance update, called the Joseph's form covariance update [19].

3.5.2 Improving estimations

In addition to estimating the 3D-trajectory from 2D-observations, the Kalman filter was also used for improving estimations. Some of the estimations were quite noisy, i.e. the estimated flight path was not a smooth motion. This was most notable with the recurrent neural network and the local motion model. These could benefit from being post-processed by a smoothing algorithm such as the Kalman filter.

The process model of this filter was the same as that of the estimating filter. The difference is in the measurements model, which in this case is much simpler. The measurements were simply the x, y, and z coordinates of the corresponding point in the estimated trajectory. That is a trivial linear function of the filtered state, so a regular Kalman filter could be used.

The smoothing filter used a noise covariance matrix with quite large values in order to filter out the noise. Trajectories with little noise, like the ones created by the local motion model, could be filtered to obtain completely smooth trajectories which followed the overall shape of the estimation. More noisy trajectories, like the ones produced by the neural network, required higher covariance values and/or several runs through the algorithm in order to filter out the noise.

3.5.3 Multiple input

A third way of using the Kalman filter was implemented, which was a combination of the first two. The same physical model was used, but the measurements consisted of both the original 2D-observations and an already approximated 3D-trajectory. The approximated 3D-trajectory could have been created by any of the methods discussed above.

The measurements hence consisted of five values; x- and y-coordinates of the 2D-observation as well as x-, y-, and z-coordinates of the approximated trajectory. Since the 2D-projection function is nonlinear, an Extended Kalman Filter was used. It was similar to the one used for 3D-estimation, but combined with the trivial 3D-projection function. The Jacobian of the 2D-version was hence extended by three rows, where the first three columns of the extra rows formed an identity matrix and the remaining columns were all zero.

A visual explanation of this version of the Kalman filter is presented in figure 3.2

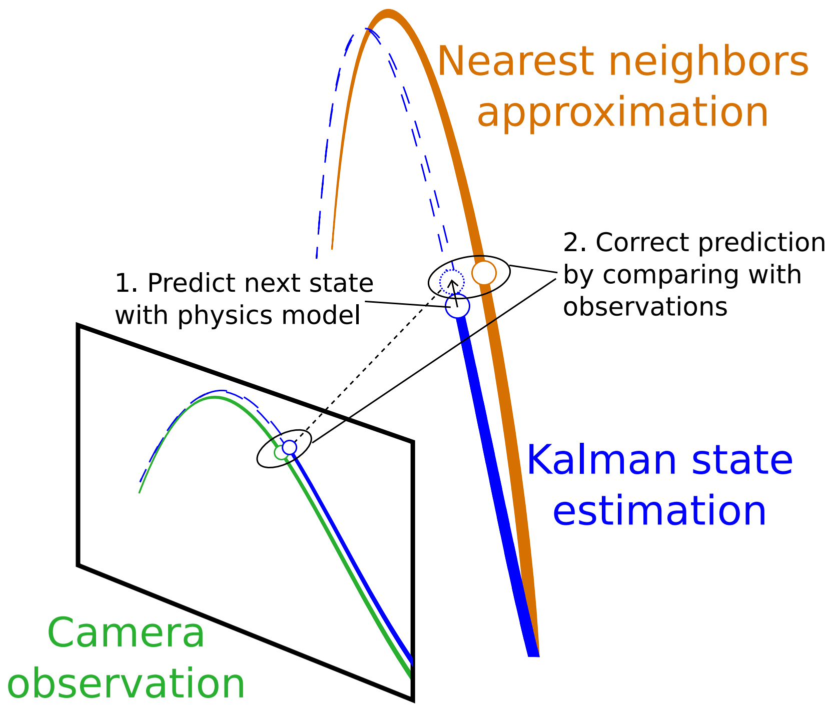

Nearest neighbors approximation
2. Correct prediction by comparing with observations
1. Predict next state with physics model
Kalman state estimation
Camera observation

Figure 3.2: A visual presentation of how the Kalman filter works.

Chapter 4

Evaluation

This chapter covers the performance of the implementations. The experiment setup is explained and the measured results are presented.

4.1 Experiments

As mentioned earlier, the study has been conducted with data from a system tracking golf shots from practice fields, so called driving ranges. Real golf shots have been tracked in a two-camera system and the existing system's 3D-reconstruction of each shot has been used as ground truth. The data from one of the cameras has then been used to estimate a 3D-trajectory with the methods discussed in this report. Each estimated trajectory has been compared to its corresponding ground truth trajectory in order to measure how well the method works.

The implemented methods have been evaluated in two ways, qualitatively and quantitatively.

The qualitative analysis consisted of manual inspection of the produced 3D-trajectories. The known 3D-trajectory and the estimated trajectory were plotted together, in order to visually compare the output to the ground truth. This was done to understand how the final output looks, what types of trajectories the method has problems with etc. A set of trajectories were randomly chosen and used for each method, to cover different types of trajectories. Some of the plotted trajectories are presented in the results section, where the real trajectory is a green dashed line, and the estimated trajectory is a blue solid line. The end of the real trajectory is marked with a white circle. The 2D-observations were taken from a camera positioned at the origin of the coordinate system. Unless otherwise stated, the shots are seen from behind the player, with the x-axis horizontally, the y-axis vertically, and the z-axis going into the page. The scale is meters, and the y-axis has negative values above ground in order to form a standard right-hand coordinate system.

The second evaluation type was a quantitative analysis of how well the different methods worked. A set of 10000 real golf shots from one driving range was used in a cross validation test. The shots were split into 10 partitions, and each partition was used as test data for one test round. The other 9000 were used as training data if the evaluated method required training. The average result from the 10 test rounds was used as the final result of the test.

The metric used for evaluating the performance of a method was based on point-wise Euclidean distance. Due to the discrete nature of a video camera, a trajectory consists of discrete points in space. For each point in the estimated trajectory, the Euclidean distance to the corresponding point in the ground truth trajectory was calculated. However, in the case of tracking a golf shot on a driving range, it is most important that the start of the trajectory is correct. The shooting bays are only a few meters apart, so an error larger than that would mean that the shot would be associated with the wrong player. For this reason, the metric used in the evaluations put a larger cost to errors in the beginning of the trajectory. The weight of an error decreased exponentially by the function

$$E_t = ||x_{et} - x_{rt}||^2 \times e^{0.01t} \tag{4.1.1}$$

where E_t is the error at point t, x_{et} is the position of point t of the estimated trajectory, x_{rt} is the position of point t of the real trajectory, and $||\cdot||$ is the Euclidean norm.

Finally, the square root of the sum of the errors was divided by the number of points in the trajectory to get an average error per point. The complete error function for a trajectory was thus

$$E = \frac{\sqrt{\sum ||x_{et} - x_{rt}||^2 \times e^{0.01t}}}{\text{length}} \tag{4.1.2}$$

where *length* is the number of points in the trajectory.

For each test setup, three statistics about the errors were recorded: the mean, standard deviation, and maximum error. In addition to this, the average unweighted distance between the real and approximated trajectories was recorded, as well as the average Euclidean error of the first point in the trajectory.

4.2 Results

The results section will begin with a summary of the best results achieved for each of the implemented methods. That will be followed by more detailed results for each of the methods.

4.2.1 Method comparison

The implemented methods have all been tested in the same way, and the performance of them are presented in table 4.1. The presented data are the best achieved by each method, i.e. with the best tested parameter configuration. More results are presented below in a separate section for each method.

Explanation of values:

Mean The mean (average) error according to the error measure described in section 4.1

Std.dev. The standard deviation of the error measure

Max The largest error recorded by the method

Distance The unweighted average distance in meters between corresponding points in the real and estimated trajectories

Dist 1st The average distance in meters between the first points in the real and estimated trajectories

Time The average execution time for estimating one trajectory, in milliseconds

Table 4.1: Best performance of each of the implemented methods

Setup	**Mean**	**Std.dev.**	**Max**	**Distance**	**Dist 1st**	**Time**
Kalman optimization of kNN	0.385	0.414	4.361	3.577	1.620	126.309
Nearest neighbors	0.406	0.389	3.967	3.757	1.705	129.848
Kalman filter (known start)	0.450	0.466	5.796	5.190	0	2.772
Kalman filter (kNN start)	0.673	0.517	4.978	6.650	1.523	33.071
Recurrent neural network	1.724	1.330	13.010	17.388	8.521	2.223
Local motion model (known start)	4.091	14.851	302.696	43.256	0.155	0.027

4.2.2 Nearest neighbors

The performance of the nearest neighbors algorithm is presented in table 4.2. Setups with different parameters and techniques are presented, in order to show the effect of each modification. The first row shows the best performance, which has been found experimentally. The following rows show the performance when one property has been altered compared to the optimal setup, in order to show how the performance is affected by each property. The following properties were altered:

Number of neighbors, k How many neighbors to find

Minimum length of neighbor How long a neighbor trajectory must be in order to be used, measured in percent of the new trajectory, e.g. >70%

Neighbor combination How the found nearest neighbors should be combined to form the approximation. The three combination methods used were *un-weighted*, *weighted*, and *pointwise weighted*, as explained in the implementation section

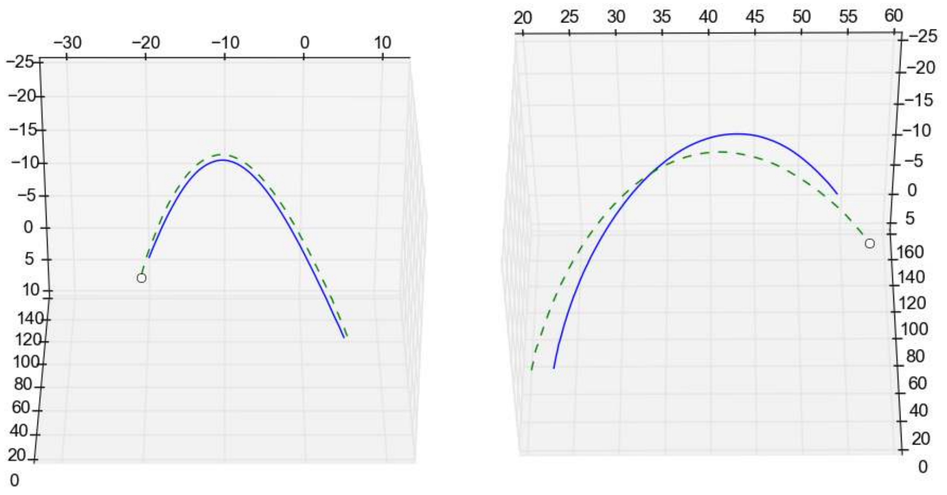
Similarity measure What type of similarity measure to use, and which properties of an observation to use

Table 4.2: Performance of the nearest neighbors algorithm. Best performance uses k=5, >50%, weighted average, and standardized similarity measure with position, velocity, acceleration and size of the ball.

Setup	**Mean**	**Std.dev.**	**Max**	**Distance**	**Dist 1st**	**Time**
Best performance	0.406	0.389	3.967	3.757	1.705	129.848
k=3	0.408	0.404	3.998	3.879	1.692	138.125
k=7	0.411	0.389	4.278	3.709	1.721	139.582
>70%	0.404	0.390	4.525	3.805	1.709	123.306
Pointwise weighting	0.425	0.396	4.185	3.924	1.798	137.834
Unweighted average	0.427	0.395	3.871	3.939	1.806	136.976
Euclidean	0.558	0.580	5.825	4.574	2.396	132.591
No velocity	0.421	0.407	4.626	3.847	1.880	135.209
No acceleration	0.408	0.390	3.987	3.766	1.739	134.096
No size	0.468	0.472	4.527	4.132	1.875	127.994
With direction	0.415	0.406	4.246	3.878	1.746	128.668
Dynamic time warping	0.425	0.395	4.456	3.792	1.738	835.662
Not standardized values	0.544	0.551	5.511	4.544	2.274	60.656

Figure 4.1 is showing two trajectories estimated by the nearest neighbors algorithm. They have been chosen to show the typical output of the algorithm.

A scalability test was done for the nearest neighbors algorithm and the results are shown in figure 4.2. In this test, position, velocity, acceleration, size and direction were used as distance features, and the regular unweighted average was used to combine the neighbors. The algorithm found the 5 most similar neighbors, and only checked candidates which had at least 70% as many observations as the target trajectory.



(a) Some estimations are very close to the real trajectory, especially if they are close to the camera.

(b) Far from the camera, the knn algorithm can still find the overall shape of the trajectory, but it can be a few meters off.

Figure 4.1: Plots of trajectories created by the k-nearest neighbors algorithm

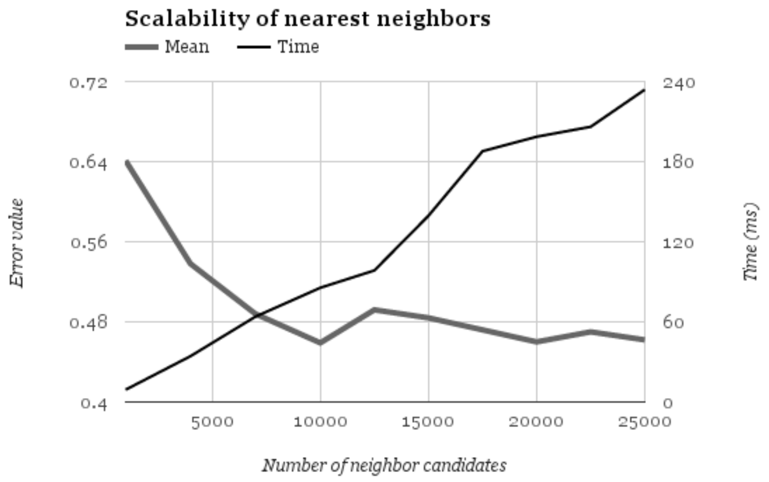


Figure 4.2: A chart showing the performance and time scalability of the full nearest neighbors algorithm. It can be seen that the time complexity is linear and that the accuracy stops to improve at around 10000 candidates.

As can be seen by the chart, the accuracy stops improving at around 10000 candidates, at which size the running time is around 100 ms. That is fast enough for real time processing, which means that there is no need for implementing a faster version of the nearest neighbors algorithm, like the hashed or indexed versions described in section 2.3.2

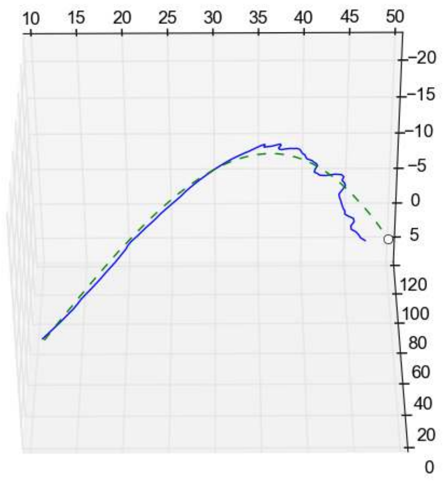
4.2.3 Local motion model

The local motion model was tested with a known initial point in order to find the optimal potential of the method. It performed badly, so it was not tested further with an approximated initial point. In the results table 4.3 below, a handful of trajectories have been omitted when calculating the statistics. They were off by several orders of magnitude and thus affected the values a lot, even though they were few. Those types of obviously wrong estimations can easily be classified as outliers and do not reflect the overall performance of the algorithm, which is why they were omitted in the calculations.

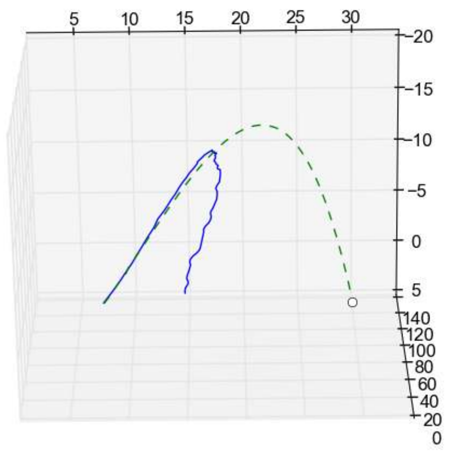
In addition to the results in table 4.3, figure 4.3 shows two estimations made by the local motion method.

Table 4.3: Performance of the local motion model

Setup	**Mean**	**Std.dev.**	**Max**	**Distance**	**Dist 1st**	**Time**
Local motion model	4.091	14.851	302.696	43.256	0.155	0.027



(a) One of the few estimations by the local motion method that is close to the correct trajectory.



(b) Output from the local motion algorithm is usually not close to the real trajectory.

Figure 4.3: Plots of trajectories created by the local motion method

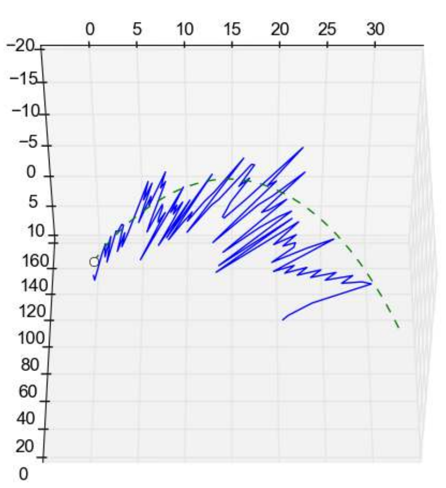
4.2.4 Recurrent neural network

The performance of the NARX Recurrent Neural Network is displayed in table 4.4. The network had the design described in section 3.4. The network was trained with a resilient backpropagation and simulated annealing, and the training was allowed to last for 300 epochs. The number of hidden neurons was set to 30, and a sliding window of size 7 was used.

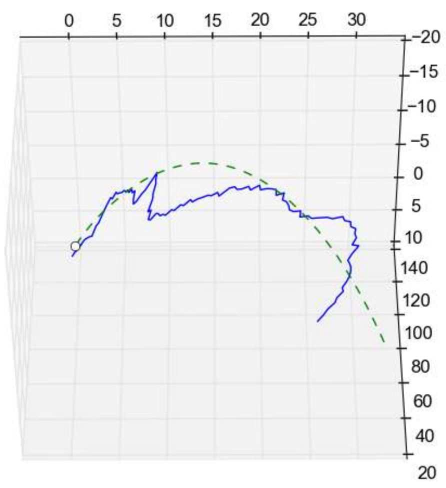
Since the output of the network was very noisy, the raw predictions were also smoothed by a Kalman filter. Results from both the original and the filtered versions are presented, as *Regular RNN* and *Kalman-filtered RNN* respectively. Plots of an unfiltered trajectory and its filtered version are also presented in figure 4.4.

Table 4.4: Performance of the recurrent neural network

Setup	**Mean**	**Std.dev.**	**Max**	**Distance**	**Dist 1st**	**Time**
Kalman-filtered RNN	1.724	1.330	13.010	17.388	8.521	2.223
Regular RNN	2.011	1.497	14.863	20.263	8.521	0.566



(a) One of the best results created by a neural network. It is very noisy but somewhat resembles the actual trajectory.



(b) The same trajectory after being smoothed by a Kalman filter. It still does not resemble a golf shot.

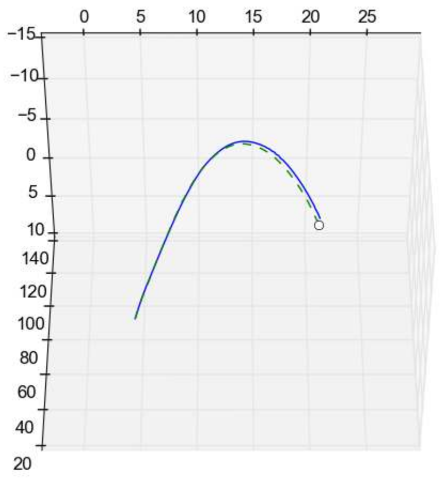
Figure 4.4: Plots of trajectories created by a Recurrent Neural Network

4.2.5 Kalman filter

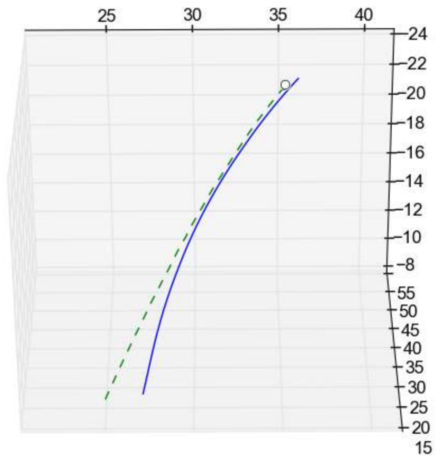
The performance of the standalone Kalman filter estimator is presented in table 4.5. Tests were performed both when the position of the initial point was known and when it was approximated with nearest neighbors. A few plots of estimated trajectories are also presented in figure 4.5.

Table 4.5: Performance of the standalone Kalman filter

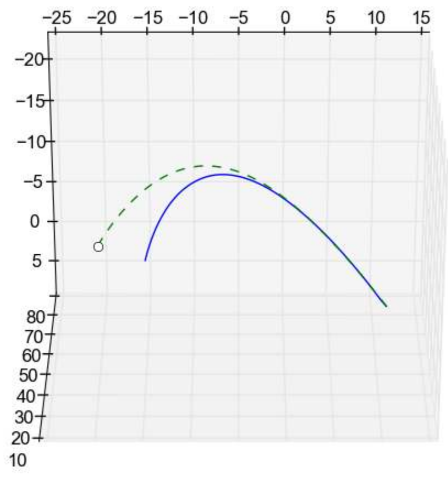
Setup	**Mean**	**Std.dev.**	**Max**	**Distance**	**Dist 1st**	**Time**
Kalman (known initial point)	0.450	0.466	5.796	5.190	0	2.772
Kalman (approximated initial point)	0.673	0.517	4.978	6.650	1.523	33.071



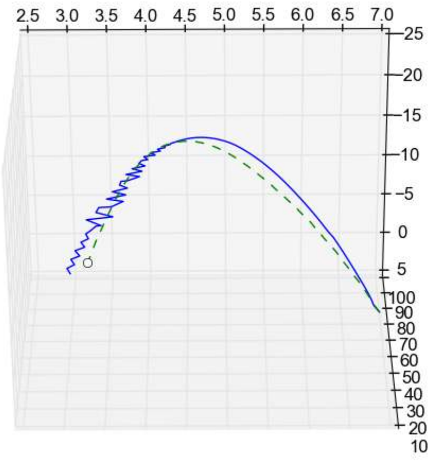
(a) The Kalman filter on its own is capable of creating more or less exact reconstructions of the observed flight path. This trajectory is constructed with the known initial point.



(b) When the initial point is approximated (here with nearest neighbors), the Kalman filter is still capable of converging towards the correct trajectory.



(c) The Kalman filter sometimes has problems with the length of the shot, which might happen if the shot differs too much from the common physics model



(d) This approximation was made using the faster covariance update algorithm, which is not numerically stable and results in a zigzag-pattern in the end of the trajectory.

Figure 4.5: Plots of trajectories created by the standalone Kalman filter method

4.2.6 Kalman optimization of Nearest neighbors

The performance of the nearest neighbors algorithm after it has been optimized by a Kalman filter is presented in table 4.6. The Kalman filter used both 2D- and 3D-measurements, as explained in section 3.5.3. There are also plots of estimated trajectories in figure 4.7.

Tests were also performed to see if the initial position of the ball affects the performance of the estimator. For these tests, the test set consisted of trajectories which were closer to the camera than x meters on the x-axis when they were first observed. The training set still consisted of all known trajectories, regardless of their first observed position. Tests for 5, 10, and 20 meters from the camera were performed. The original test set had trajectories of up to 35 meters from the camera. The result is shown in the table below, labeled *<x meters from camera*. Figure 4.6 is showing how the error is varying with the initial ball position. It is very clear that the error on the x-axis is significantly lower close to the camera.

Table 4.6: Performance of the nearest neighbors algorithm optimized by a Kalman filter

Setup	**Mean**	**Std.dev.**	**Max**	**Distance**	**Dist 1st**	**Time**
All trajectories	0.385	0.414	4.361	3.577	1.620	126.309
<5 meters from camera	0.302	0.302	2.140	2.766	0.681	126.050
<10 meters from camera	0.331	0.338	2.695	3.040	0.770	126.062
<20 meters from camera	0.368	0.377	3.215	3.421	1.102	127.591

For the nearest neighbors plus Kalman method, a manual quality analysis was also performed. The quality of the reconstruction of 100 random trajectories were assessed, and marked acceptable or not acceptable. The criterion for being marked acceptable was that the origin of the trajectory should not differ by more than 1.5 meters, the shape of the trajectory should match the real shape, and the error of the end point of the trajectory should be less than 10% of the length of the trajectory (e.g. 10 meters too short on a 100 meter shot). The inspection showed that 74% of the approximations were acceptable. When only looking at shots from within 10 meters of the camera, the ratio of acceptable shots rose to 95%.

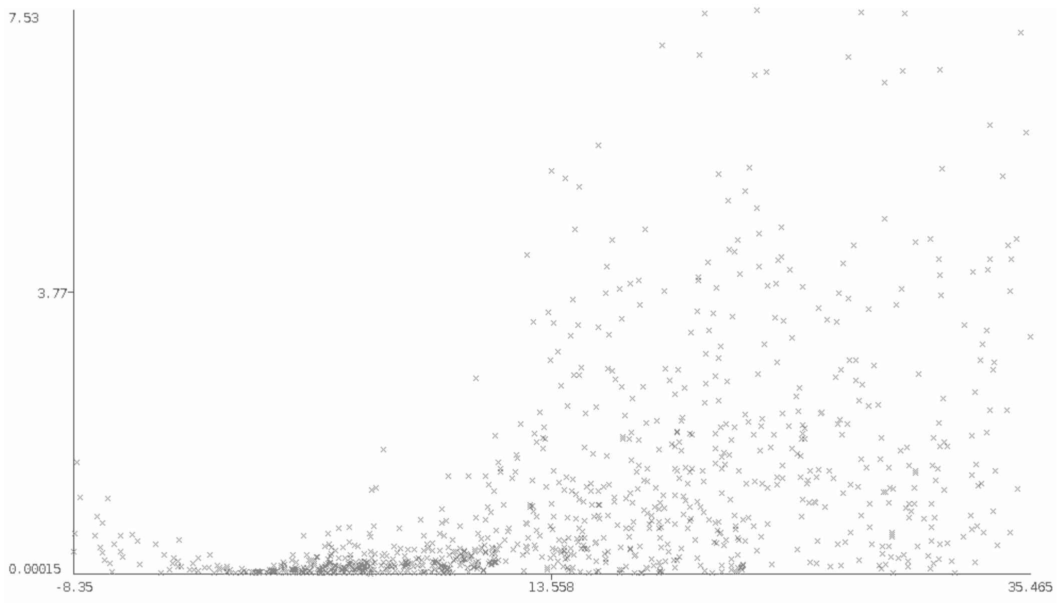
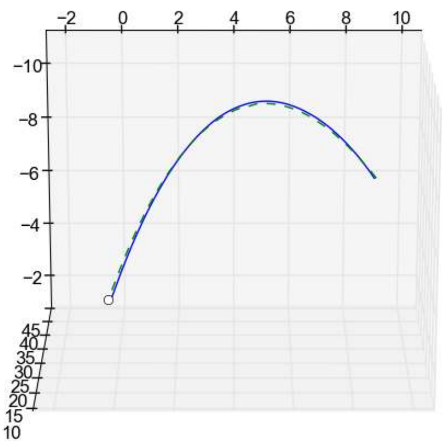
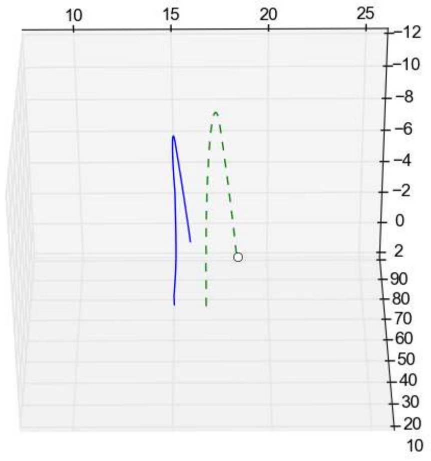


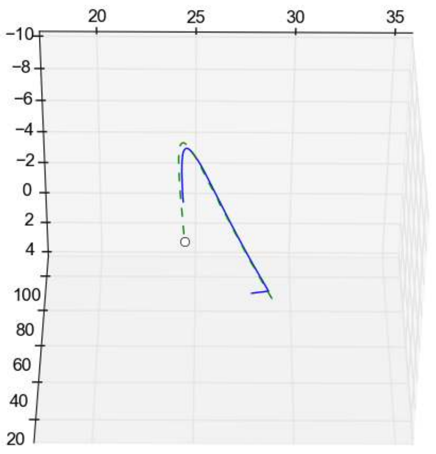
Figure 4.6: Distribution of x-axis error. The horizontal axis is the actual x-axis of the real world coordinate system. The vertical axis shows how far off the x-coordinate of the first point in an estimation is from the real value, in meters. The camera is positioned at x=0.



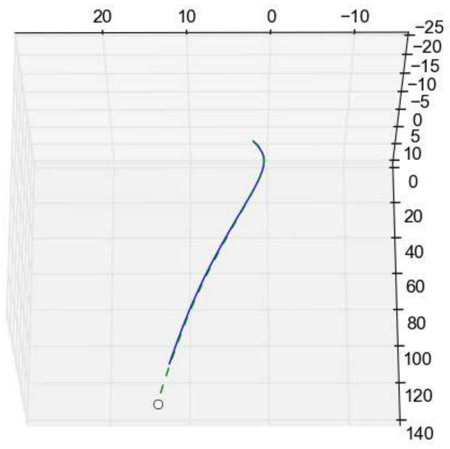
(a) A plot of one of the best approximations created by the knn/kalman method. The error between the approximated (blue solid) and real (green dashed) trajectories is negligible.



(b) The nearest neighbors algorithm sometimes has problems determining the x-position of the trajectory, which the Kalman optimizer cannot always correct. This happens predominantly on shots far away from the camera.



(c) The Kalman filter does not affect the first point of an approximation, but the filter converges to its optimal estimation already at the second point. The first point should therefore be discarded.



(d) The knn/kalman method is also able to match more complex shapes, like this sliced shot (seen from opposite direction compared to other plots)

Figure 4.7: Plots of trajectories created by the Kalman-optimized kNN method. The method produces smooth, realistic trajectories, and the accuracy can be very high. The shape of the reconstructions are with a few exceptions very similar to the real trajectory.

Chapter 5

Outcome

The final chapter presents what has been shown in this study. The chapter starts with an explanation and discussion about the results of the study, which is followed by the conclusion that has been drawn from the experiments. Finally, there is a section about potential future work in the field.

5.1 Discussion

The nearest neighbors algorithm was the single method which produced the most consistent results. There are countless parameters and modifications that can be tweaked to produce better results, for instance the similarity measure including which features to compare, number of neighbors to find, how to combine the neighbors, which parts of the trajectory to use etc. Standardization, features in similarity measure, and weighting of the neighbors seemed to contribute most to the performance. The output from the knn-algorithm was often close to the real trajectory, especially if the trajectory started close to the camera, but it was difficult to achieve perfect results with knn on its own.

The recurrent neural network produced too unstable results to be usable. The output from the neural network was very noisy and even if the noise was reduced with a Kalman filter, the resulting trajectory only roughly followed the correct one.

The local motion model also was too unstable to use; estimated trajectories could be off by several hundred meters at some instances. A likely cause of this is that the underlying assumption, that the ball moves with constant velocity during a short time span, is incorrect. Trying to use another local motion constraint might not work either, since external variables like wind and the spin of the ball make it difficult to state a ground truth about exactly how a ball will move between two camera observations. Also, since each depth estimate is relative of its neighbor observations, even small errors may propagate and become very large in the end of the trajectory. The only positive property of the local motion model is that it is extremely fast, in the order of 10 microseconds.

The Kalman filter as a standalone estimator was working decently on average, but the lack of depth observations could cause quite large deviations from the correct trajectory. It was therefore less reliable than the nearest neighbors approach. However, when the 2D-observations were used together with an already approximated trajectory, the Kalman filter worked much better. The algorithm still tried to estimate the state from 2D-observations, but since approximate 3D-points were given as a guidance, the estimated state deviated much less from the correct trajectory. An important point to note here is that the 2D-observations were given a much smaller measurement noise variance compared to the approximated 3D-trajectory. This forces the filter to trust the 2D-points more than the 3D-points, which is exactly what is desired, since the 2D-coordinates are actual observations and the 3D-coordinates are just rough estimations. The Kalman filter with 2D and 3D input could be used to optimize approximated trajectories from any standalone 3D-trajectory estimator, but the nearest neighbors algorithm was used in this study since it was the most stable approximator of the ones tested.

Theoretically, the 3D-position of an object could be calculated mathematically when knowing the size of the object. However, that requires that the size of the object in the image can be measured precisely. In the case of golf balls shot on driving ranges, measurements are affected too much by external factors like light and weather to be reliable. However, using the image size of the ball as a feature in the similarity measure of the nearest neighbors algorithm did improve those results significantly.

Scientific studies and new technology may touch on aspects regarding ethics, sustainability or society. This study does not handle any sensitive data and does not perform any tests that could be considered unethical. The implemented system is also intended as a backup system to a more precise system, so any potential ethical dubiousness of possible use cases should have already been assessed for the original system.

5.2 Conclusion

The problem of reconstructing a 3D-trajectory based on a single 2D-trajectory and previous known trajectories has been approached in multiple ways in this study. The results have been varied, but a few of these methods were able to fairly consistently produce good enough approximations of 3D-trajectories. The best method turned out to be a nearest neighbors-search of old known trajectories combined with an optimizing Kalman filter. For this method, the average deviation from the real trajectory was around 3.5 meters, and around 1.5 meters at the start of the trajectory. For shots closer to the camera than 10 meters, the error was reduced to 3 and 0.75 meters respectively, which must be considered acceptable. The method also consistently produced reconstructions which had the same shape as the real trajectory. All tested methods were also fast enough for real time use, with the nearest neighbors algorithm being the slowest one at the order 100 ms. The conclusion is therefore that old observations can be utilized by combining the most similar old trajectories, and improving the output with a physics-based state estimator.

5.3 Future work

The Kalman filter proved to be a good tool in the estimation process. However, it is intended primarily for on-line estimations, and it does not utilize the entire observation series in the estimation process. A state space estimation algorithm which uses more of the available observations would be interesting to test, for instance a moving horizon estimator.

It would be interesting to use data other than camera observations, especially to estimate the initial point of the trajectory. For instance, some kind of sensor that registers where shots are made from could help generate a more accurate estimation of the initial position of each shot, which might help reduce the overall error.

This study has performed all tests on data consisting of golf shots on driving ranges. For this particular use case, the results have been shown to be good. It would be interesting to see how well the discussed methods generalize to other types of data, perhaps from other sports or other types of tracking problems.

Bibliography

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO '93)*, pages 69–84, 1993.

[2] S. Avidan and A. Shashua. Trajectory triangulation: 3D reconstruction of moving points from a monocular image sequence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:348–357, 2000.

[3] S. Bahadori and L. Iocchi. A stereo vision system for 3d reconstruction and semi-automatic surveillance of museum areas. In *AI*IA 2003: Advances in Artificial Intelligence, 8th Congress of the Italian Association for Artificial Intelligence*, Pisa, Italy, 2003.

[4] S. S. Beauchemin and J. L. Barron. The computation of optical flow, 1995.

[5] C. Cassisi, P. Montalto, M. Aliotta, A. Cannata, and A. Pulvirenti. Similarity Measures and Dimensionality Reduction Techniques for Time Series Data Mining. In A. Karahoca, editor, *Advances in Data Mining Knowledge Discovery and Applications*, chapter 3. InTech, Sept. 2012.

[6] H. T. Chen, M. C. Tien, Y. W. Chen, W. J. Tsai, and S. Y. Lee. Physics-based ball tracking and 3D trajectory reconstruction with applications to shooting location estimation in basketball video. *Journal of Visual Communication and Image Representation*, 20:204–216, 2009.

[7] J. Civera, A. J. Davison, and J. M. M. Montiel. *Structure from Motion Using the Extended Kalman Filter*, volume 2011. Springer Berlin Heidelberg, 2011.

[8] H. Cruse. *Neural Networks as Cybernetic Systems*, volume 12. 2006.

[9] D. E. DiFranco, T.-J. Cham, and J. M. Rehg. Reconstruction of 3D figure motion from 2D correspondences. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, 1, 2001.

[10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-series Databases. *SIGMOD Rec.*, 23(2):419–429, 1994.

[11] K. Funahashi and Y. Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6:801–806, 1993.

[12] F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 12(6):1333–40, Jan. 2001.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. pages 518–529, Sept. 1999.

[14] A. Guéziec. Tracking pitches for broadcast television. *Computer*, 35:38–43, 2002.

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

[16] R. Herrejon Mendoza, S. Kagami, and K. Hashimoto. Online 3-D trajectory estimation of a flying object from a monocular image sequence. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, pages 2496–2501, 2009.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–1780, 1997.

[18] D. C. Kale, D. Gong, Z. Che, Y. Liu, G. Medioni, R. Wetzel, and P. Ross. An Examination of Multivariate Time Series Hashing with Applications to Health Care. In *2014 IEEE International Conference on Data Mining*, pages 260–269. IEEE, Dec. 2014.

[19] L. Kleeman. Understanding and applying Kalman filtering. *Proceedings of the Second Workshop on Perceptive ...*, 1996.

[20] K. Konda and R. Memisevic. Learning to combine depth and motion. *arXiv preprint arXiv:1312.3429*, 2013.

[21] J. Konrad, M. Wang, P. Ishwar, C. Wu, and D. Mukherjee. Learning-based, automatic 2D-to-3D image and video conversion. *IEEE Transactions on Image Processing*, 22:3485–3496, 2013.

[22] A. Kumar, P. S. Chavan, V. K. Sharatchandra, S. David, P. Kelly, and N. E. O'Connor. 3D Estimation and Visualization of Motion in a Multicamera Network for Sports. In *Machine Vision and Image Processing Conference (IMVIP), 2011 Irish*, pages 15–19, Sept. 2011.

[23] P. Pichler. State Space Models and the Kalman Filter, 2007.

[24] E. Ribnick, S. Atev, and N. P. Papanikolopoulos. Estimating 3D positions and velocities of projectiles from monocular views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:938–944, 2009.

[25] A. Saxena, M. Sun, and A. Y. Ng. Make3D: Learning 3D scene structure from a single still image. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:824–840, 2009.

[26] L. Schnyder, O. Wang, and A. Smolic. 2D to 3D conversion of sports content using panoramas. In *Proceedings - International Conference on Image Processing, ICIP*, pages 1961–1964, 2011.

[27] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In *Neural Information Processing Systems*, 2014.

[28] H. Wang, H. Su, K. Zheng, S. Sadiq, and X. Zhou. An Effectiveness Study on Trajectory Similarity Measures. In *Proceedings of the Twenty-Fourth Australasian Database Conference - Volume 137*, ADC '13, pages 13–22, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[29] G. Welch and G. Bishop. An Introduction to the Kalman Filter. *In Practice*, 7:1–16, 2006.

[30] Y.-L. Wu, D. Agrawal, and A. El Abbadi. A comparison of DFT and DWT based similarity search in time-series databases. *Proceedings of the ninth international conference on Information and knowledge management CIKM 00*, 35:488–495, 2000.

[31] F. Xu, K. M. Lam, and Q. Dai. Video-object segmentation and 3D-trajectory estimation for monocular video sequences. *Image and Vision Computing*, 29:190–205, 2011.

[32] Y. K. Yu, K. H. Wong, and M. Chang. Recursive Three-Dimensional Model Reconstruction Based on Kalman Filtering. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, 35(3):587–592, June 2005.

[33] Z. Zhang, K. Huang, and T. Tan. Comparison of similarity measures for trajectory clustering in outdoor surveillance scenes. In *Proceedings - International Conference on Pattern Recognition*, volume 3, pages 1135–1138, 2006.