Visual Tracking of a Swinging Pendulum

A comparison of the particle filter and the Kalman filter

Kristoffer Nesland

2018-12-11

Contents

List of Figures

List of Tables

Preface

The process of writing a project report has been enjoyable. Personally, I believe computer vision is one of the most interesting scientific topics today. Advances in the field are happening these days and they are happening fast. The idea of "letting robots see" is fascinating and the wide array of use cases make it feel like you are part of an important evolution.

I would like to thank my supervisor Olav Egeland for the interesting and relevant challenges he has given to us students. Further, PhD candidate Aksel Sveier has been important in my work and has provided me with insights into the particle filter, Kalman filter and writing code for GPUs. Also, Torstein Myhre and Håkon Hystad's work on the topic of visual tracking of swinging conveyors has been important in my work. Lastly, my gratidtude goes to my fellow students for great cooperation and a pleasant working environment.

Summary

Integration of vision technology in industrial manipulators can significantly expand their capabilities. It allows for interaction with objects of unknown poses. Yet, this is not a simple task because real-time pose estimation of objects from camera images requires high performing algorithms and efficient implementations.

In this report, a relatively simple experiment is conducted. A video of a pendulum with a fixed pivot swinging in a plane is captured. The task is to estimate the angle of this pendulum and a particle filter is chosen to cope with the challenge. Alongside this, the report is meant as a practical introduction to the particle filter. Background theory necessary to understand the workings of a basic particle filter is presented together with code implementations. The report also demonstrates how the particle filter can be accelerated by implementing some of the necessary functions on a graphics processing unit.

A more traditional approach to filtering is the Kalman filter. To investigate if a Kalman filter can achieve an accuracy comparable to the particle filter, a Kalman filter is also implemented. The experiment shows that both the particle filter and the Kalman filter is able to track the pendulum with a high accuracy. However, it is discovered that localization of the pendulum within independent image frames, without any filtering, provides similar results. Therefore, it can be argued that the experiment was too simple. A more cluttered background environment and estimation of of a higher dimensional state could lead to greater differences between the two filtering techniques.

Chapter 1.

Introduction and Related Work

1.1. Background and motivation

Industrial automation can leverage from the use of vision. By analyzing incoming frames from a video camera, it is possible for industrial robots to perform operations on parts not only when the pose is predetermined. An example of the need for such a system is in the operation of loading and unloading parts. More specifically, Torstein Myhre, a former PhD candidate at the old Department of Production and Quality Control (NTNU), identified the task of loading electrical heaters onto a roof-mounted conveyor line, as illustrated in figure 1.1. As the conveyor accelerates, the hangers are sent into a pendulum motion. To account for this swinging motion, there is a need for a vision system to successfully load the parts using an industrial robot. If one manages to implement a system like this, the repetitive work of loading the electrical heaters can be automated, helping to drive down costs.

The material presented in this report is mainly meant as an introduction to the particle filter and the Kalman filter. Both the mathematical background and an example of an implementation is presented for the filters. The implementation is written for the purpose of tracking a 2D pendulum with a fixed pivot. Hence, the scope of the report is limited to estimation of the angle and the angular velocity of a pendulum oscillating in the plane. As it turns out, the particle filter is suitable for parallel computations and the report shows how parts of the filter can be accelerated on a GPU. In general, much effort has been devoted to programming different implementations, experiment with settings of filter parameters and making the code easy to work with. Lastly, the performances of the two filters are compared. What are the advantages of implementing a particle filter for the

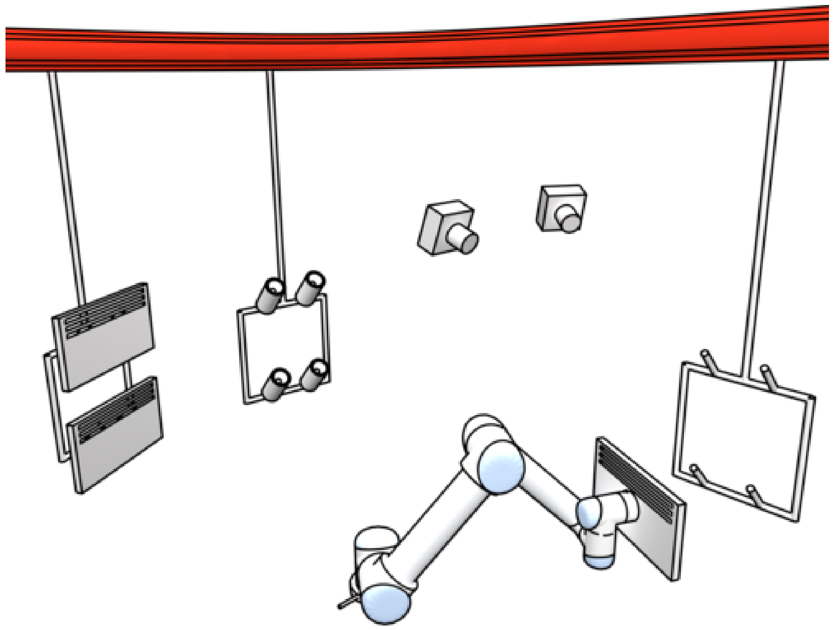

Figure 1.1.: Loading of electrical heaters onto roof-mounted conveyor line [17].

purpose of visual tracking compared to the more traditional Kalman filter? The report will try to answer this question along with the practical introduction to the two filtering techniques.

1.2. Previous work

Work related to autonomous loading and unloading of parts on an overhead conveyor has been researched by the old Department of Production and Quality Control at NTNU. In the doctoral thesis of Olivier Roulet-Dubonnet [19], computer vision is used to detect loading positions. Morten Lind also used computer vision in his doctoral thesis [15] for loading of carriers. Both these theses are inspirations in Thorstein Myhre's work on *Vision-Based Control of a Robot Interaction with Moving and Flexible Objects* [17] that lays the foundation for this project.

Myhre applied a particle filter to the visual tracking problem [18]. In contrast to earlier efforts he specialized the filter to track an overhead conveyor by modeling the dynamics as a spherical pendulum instead of a simpler linear motion approximation. The measurements needed in the particle filter are retrieved by comparing camera images to a mathematical representation of the conveyor with the *Condensation Algorithm*, inspired by Isard and Blake [9]. Additionally, Myhre implemented an on-line method for calculating the center of mass of the pendulum. This was important because the center of mass changes during loading of part onto the conveyor. Later, Håkon Hystad extended Myhre's work by allowing the pivot to move during tracking by estimating the pivot's position and velocity [8]. This gave a total of 12 variables to be estimated, six for pendulum orientation and angular velocity relative to the pivot and six for the position and velocity of the pivot. In other words, a challenging estimation.

For manipulation using visual sensing, Kragic proposes to divide the process into two [14]. The first step is the initial coarse alignment "using crude image features such as center of mass". For this prediction and estimation Kragic applies an $\alpha-\beta$-filter. The correlation of image regions compared to a template is evaluated, colour similarity analyzed in the chromatic colour space and the perceived motion in regions are all variables that are used in the weighting step of the filter. For the second and more accurate step of the visual sensing a model-based approach is initialized around the estimate of the first step. As in the *Condensation Algorithm*, a mathematical model of what will be tracked is projected onto the image and how well the projected model fits with the underlying image is used as an indication of how accurate the projection is.

For localization of objects in images, a wide array of methods exist. *The Generalised Hough Transform* can be used to find arbitrary objects in images by a voting system [11]. *Viola-Jones* [21] is traditionally used for face detection, but can be trained to localize desired objects of any shape. The mentioned methods, along with many others, can be used as sensor input to filtering processes and in this way contribute to visual tracking, not only localization of objects in independent images.

Major breakthroughs have been achieved in the field of computer vision the last years. After a convolutional neural network called *AlexNet* won the *ImageNet Large Scale Visual Recognition Challenge* in 2012, the field has been dominated by neural networks. This type of machine learning has proven to perform extremely well in image classification, object detection and segmentation tasks. Neural network techniques have also been applied for videos. State of the art methods for video semantics classification [22] are all leveraging on neural networks. However, it should be noted that the neural networks have had problems when it comes to real-time analysis of video. An interesting approach to the task of real-time video tracking has been presented by a group of researchers from Stanford [7]. By feeding the network with the previous frame and the current frame of a video, they have been able to track arbitrary objects at 100 frames per second.

Chapter 2.

Background Theory

2.1. Vectors

When tracking an object, a representation of the object's position is needed. The position can be described by a vector in a coordinate frame. A three-dimensional Cartesian coordinate frame is defined by three orthogonal unit vectors, $\vec{e}_1$, $\vec{e}_2$ and $\vec{e}_3$. As a linear combination of the orthogonal unit vectors, all possible positions can be described, $\vec{v} = v_1\vec{e}_1 + v_2\vec{e}_2 + v_3\vec{e}_3$, figure 2.1. Further, given a coordinate system, a position can be represented by a column vector,

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}.$$

2.2. Coordinate transformations

In many applications, it is necessary to be able to know how to transform a vector from one coordinate frame to another. As an example, we could think of the interaction between a camera and an industrial manipulator. If the camera has determined the position of an object that the manipulator should grab, the vector in the frame of the camera should be transformed to the frame of the manipulator. This transformation will be a function of the difference in position and orientation of the coordinate systems of the camera and the manipulator. The same position can be described in two different coordinate frames, a and b,

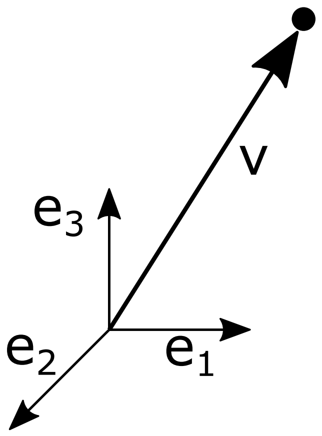

Figure 2.1.: A vector in a Cartesian coordinate system.

as

$$\vec{v} = v_1^a \vec{a}_1 + v_2^a \vec{a}_2 + v_3^a \vec{a}_3$$

and

$$\vec{v} = v_1^b \vec{b}_1 + v_2^b \vec{b}_2 + v_3^b \vec{b}_3.$$

If we use the column vector description, we must specify if the coordinated are expressed in frame a or b, $\vec{v}^a$ or $\vec{v}^b$.

2.3. Rotation matrices

A three-dimensional vector can be transformed from a coordinate frame to another coordinate frame with a different orientation by premultiplying it with a matrix,

$$\vec{v}^a = R_b^a \vec{v}^b,$$

where R_b^a is called a rotation matrix. This particular one is a coordinate transformation matrix from frame b to a.

Transformation matrices for simple rotations of an angle about one of the axes of the coordinate frame are as follows

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

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

$$R_z(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If frame b is rotated $\frac{\pi}{4}$ around the z-axis relative to frame a, the column vector in frame a can be found as $\vec{v}^a = R_z(\frac{\pi}{4})\vec{v}^b$. This could also be seen as rotating a point by $\frac{\pi}{4}$ inside the same coordinate system.

More complicated rotations can be achieved by chaining rotations about the x-, y- and z-axes. A rotation matrix for a rotation about the x-axis by ϕ, then around the y-axis by θ and lastly around the z-axis by ψ can be found as $R = R_z(\psi)R_y(\theta)R_x(\phi)$.

2.4. Homogeneous transformation matrices

Sometimes not only rotation from a coordinate frame to another is desirable, but also a translation. This is achieved through the concept of the homogeneous transformation matrix.

$$T_b^a = \begin{bmatrix} R_b^a & \vec{r}_{ab}^a \\ \vec{0}^T & 1 \end{bmatrix}$$

Here, $\vec{r}_{ab}^a$ is the vector describing the position of the origin of frame b relative to the origin of frame a. R_b^a is the same rotation vector as already seen. Homogeneous transformation matrices can be combined in the same way as rotation matrices.

T_b^a will be a 4×4 matrix and we need a four-dimensional vector to multiply with it. This is simply obtained by adding an entry of 1 at the end of the original vector,

$$\tilde{\vec{v}} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 1 \end{bmatrix}.$$

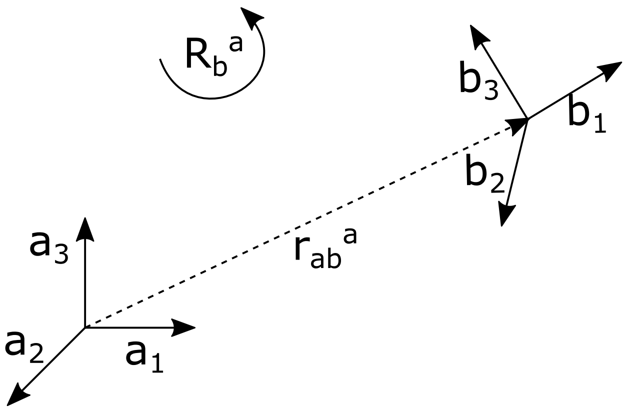

Figure 2.2.: A coordinate transformation from frame *b* to *a*.

2.5. The camera model

In visual tracking, how a point in space is projected onto the image captured by the camera must be known. If we want to track a point and believe it is at a position

$$\vec{r}^{p}_{cp} = \begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

it must be possible to relate this position to a specific pixel in the image. Then, the image can be examined around this pixel. For this to work, the difference in position and orientation of the coordinate system of the camera and the point has to be determined. A homogeneous transformation matrix between the object space and the camera space is needed,

$$T^c_o = \begin{bmatrix} R^c_o & \vec{t}^c_{co} \\ \vec{0}^T & 1 \end{bmatrix}.$$

R^c_o is a rotation matrix that contains the orientation difference between the object frame and the camera frame. $\vec{t}^c_{co}$ is the translation from the camera frame to the object frame in the coordinates of the camera frame. The geometry can be seen in figure 2.3. In total, the homogeneous transformation matrix T^c_o allows for a coordinate transformation from the object space to the camera space,

$$\tilde{\vec{r}}^c_{cp} = T^c_o \tilde{\vec{r}}^o_{op}. \tag{2.1}$$

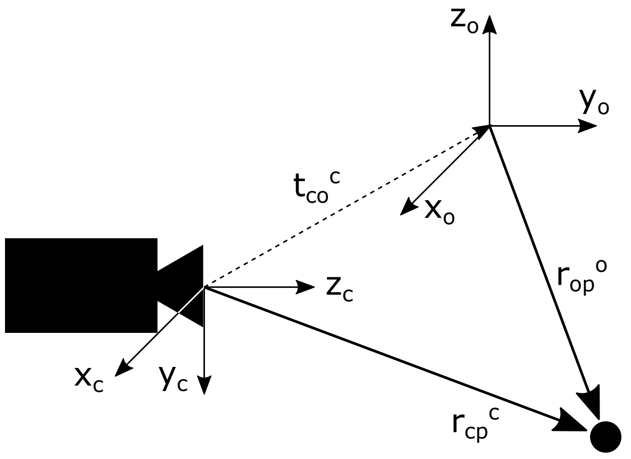

Figure 2.3.: The relation between the camera frame and the object frame.

When we now have established a transformation from the object frame to the camera frame, the next step is to normalize the camera frame coordinates. $\tilde{\vec{s}}$ is introduced as the normalized image coordinates,

$$\tilde{\vec{s}} = \begin{bmatrix} \frac{x}{z} \\ \frac{y}{z} \\ 1 \end{bmatrix} = \frac{1}{z}\vec{r}^{c}_{cp}, \tag{2.2}$$

where x, y and z are the elements of $\vec{r}^{c}_{cp}$. This normalization is a prerequisite before we can perform the last step of the projection onto the image plane.

The last step is to relate the normalized image coordinate to a pixel in the image. Because cameras can have different lenses and different image sensors, this transformation is different from camera to camera. Let u_0 be the number of pixels from the left edge of the image to the center and v_0 be the number of pixels from the top to the center. ρ_w is the horizontal width of a pixel and ρ_h is the vertical height of a pixel. f is the focal length, the distance from the camera frame to the image plane. Together, these form the five intrinsic parameters of the camera and they make up the camera parameter matrix,

$$K = \begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix}.$$

All that remains now is to calculate the actual pixel positions,

$$\tilde{\vec{p}} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\tilde{\vec{s}}. \tag{2.3}$$

Note that these pixel coordinates should be rounded to obtain the final set of two integers that tells us at which pixel position we can find $\vec{r}^{o}_{op}$ in the image. The whole procedure can be summarized as 2.1, 2.2 and finally 2.3.

Chapter 3.

Video and Calibration

3.1. Setup

To record a test video, a Sony A6300 with a 16-50 mm lens was used. The camera was put in manual mode with a fixed shutter speed, aperture, ISO and focal length. Then, a 50 FPS 1920 by 1080 pixel video was recorded with the camera on a tripod. The first part of the video consisted of a checkerboard with 30 by 30 mm squares being placed at different positions, shown in figure 3.1. The second part of the video recorded the pendulum swinging parallel to the image plane, in the $z_o = 0$ plane.

3.2. Calibration

Calibration was performed in *MATLAB*. The *Camera Calibrator* in the *Image Processing and Computer Vision* toolbox lets you input a set of images taken with a checkerboard placed at different positions in the frame. As output you obtain the intrinsic parameters for K and the transitions T_i^c from the camera to all the checkerboard-frames. 13 screenshots from the video, like the ones shown in figure 3.1, was provided to the algorithm.

The calibration procedure produced,

$$K = \begin{bmatrix} 2813.07 & 0 & 960.18 \\ 0 & 2809.37 & 563.55 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

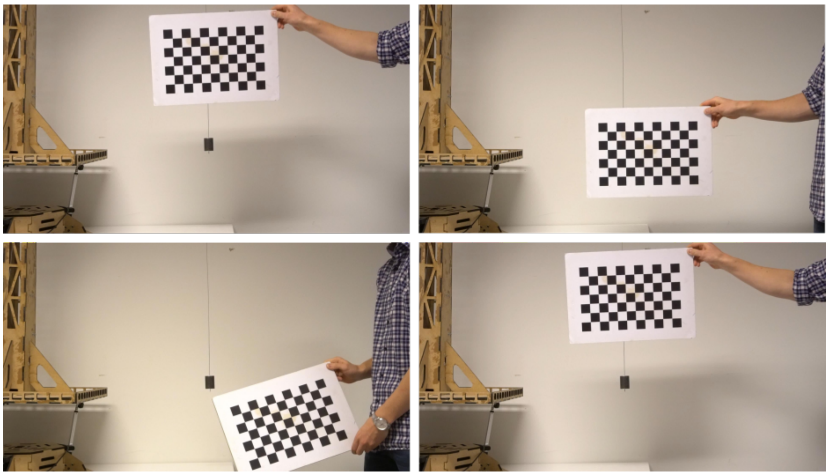

Figure 3.1.: Some of the images provided to the calibration algorithm.

and

$$T_o^c = \begin{bmatrix} 1 & 0 & 0 & 0.007 \\ 0 & -1 & 0 & -0.892 \\ 0 & 0 & -1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.2}$$

In figure 3.2, the definition of the coordinate frames are shown. Since the origin of the object frame was defined as the mounting point of the pendulum, but this point was not in the video, some trial and error was necessary to obtain the correct translation from the camera frame to the object frame.

3.3. Parameter estimation

The distance from the pivot to the pendulum's center of gravity is needed to model the pendulum's motion. This length was calculated from

$$L = \frac{gT^2}{4\pi^2},$$

where g is the gravitational acceleration and T is the period of an oscillation. Calculations revealed a rope length of $L = 0.9611$ m.

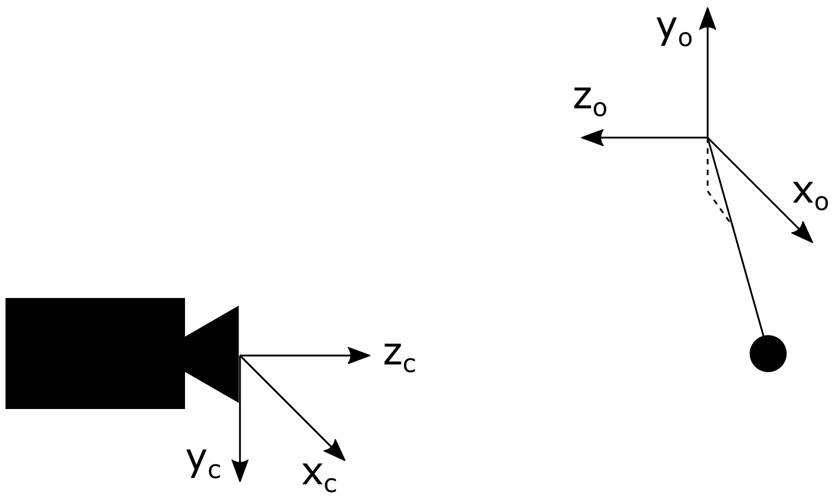

Figure 3.2.: Relationship between the camera frame and the object frame.

3.4. Labelling data

To allow for later analysis of tracking accuracy, the ground truth states was needed. For every frame of the video there should exist a label telling the true angle of the pendulum. This way, comparison of different tracking methods is possible.

MATLAB allows for labelling of videos through the *Video Labeler*-app. A pixel label was assigned to the mass center of the pendulum for all of the 347 frames of the 6.944 second long video. Then, the x and y position of the labeled pixel in every frame was extracted. Lastly, every pair of x and y was projected to the object space using the camera matrix, K, and the translation matrix, T_o^c, before the angle was calculated. The resulting array of angles was stored in a CSV textfile that can be imported to scripts where the tracking accuracy should be analyzed.

Chapter 4.

The Particle Filter

With the particle filter, we want to estimate the state of the system given all measurements so far. Whereas Kalman filters can be used for linear Gaussian problems, particle filters are also applicable for nonlinear systems. The filter has been used with success in a wide range of applications, one of the best known being the SLAM problem [16]. Here a robot is set out to generate a map of its environment and localize itself within this map simultaneously.

The goal of this chapter is to bridge this gap between theory and implementation by presenting a particle filter with a focus on a practical implementation. A more to the point explanation can be found in [6]. As the particle filter is a numerical implementation of Bayesian filtering, the equations of this filtering is presented in section 4.4. Section 4.6 to 4.9 will first explain the steps of the particle filter and then tie this together with Bayesian filtering.

4.1. The case study

In order to explain the particle filter, the setup of chapter 3 will be used. Here, we will consider the case of tracking a pendulum with one degree of freedom, oscillating in the xy-plane. As input we have a stream of images and we want to estimate the position of the pendulum. In the following, we suppose that the pivot is fixed and its position known. Also, the length of the mass-less rope is assumed known.

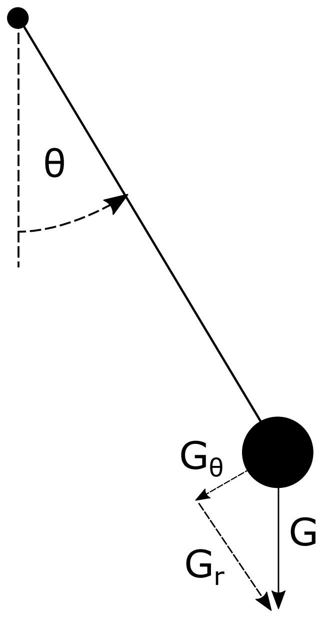

Figure 4.1.: Gravitational forces acting on a 2D pendulum.

4.2. Pendulum motion

Let us imagine that we are able to perfectly determine the position and velocity of the pendulum in the picture. However, we are only able to take a picture every 80 ms. What if we want to predict the position every 40 ms? We would then have to fill in the gaps between the pictures. To do this we need a physical model of the system.

The equations of motion for a 2D pendulum can be derived from Newton's second law of motion.

$$\sum \vec{F} = m\vec{a}$$

We assume a mass-less rope and that friction in the joint, air resistance and all other outer forces have little effect. Further, we use a polar coordinate system with the origin at the pivot.

$$\sum F_\theta = ma_\theta$$

$$mgsin(\theta) = ma_\theta$$

$$a_\theta = -gsin(\theta)$$

Expressing the acceleration by angular acceleration and radius,

$$\alpha L = -gsin(\theta),$$

where L is the length of the rope. Utilizing Euler's method, we have

$$\omega_{t+1} = \omega_t + \alpha_t \Delta t \tag{4.1}$$

and applied once again to predict the new angle

$$\theta_{t+1} = \theta_t + \omega_t \Delta t. \tag{4.2}$$

Using $\Delta t = 40$ ms, we are now able to predict the position of the pendulum, also in-between when the pictures are taken. The state vector will be the following,

$$\vec{x} = \begin{bmatrix} \theta \\ \omega \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}.$$

Written in vector notation, we obtain the update

$$\vec{x}_{t+1} = \begin{bmatrix} \theta_{t+1} \\ \omega_{t+1} \end{bmatrix} = \begin{bmatrix} \theta_t \\ \omega_t \end{bmatrix} + \begin{bmatrix} \omega_t \\ -\frac{gsin(\theta_t)}{L} \end{bmatrix} \Delta t. \tag{4.3}$$

4.3. Measurement

We have assumed that we are able to determine the pendulum's position in the pictures. For a computer, this is not a simple task and in reality we cannot do this with 100 percent confidence. Instead, we can have a hypothesis of where in the image the pendulum is. Let us have a hypothesis that $\theta = \frac{\pi}{8}$. Assuming that we are able to relate an angle with a position in the picture, we can look at this area in the image. In our toy-example, we have a dark pendulum on a bright background. Given our hypothesis of where the pendulum is in the picture, we can extract a patch around this point and examine it. The method is illustrated in figure 4.2. The more dark pixels we find in this patch, the higher the likelihood that this is the true position of the pendulum. This was just one simple example of how to analyze the images. The important point is that we are able to output a likelihood of an observation given a hypothesis of the current state of the system.

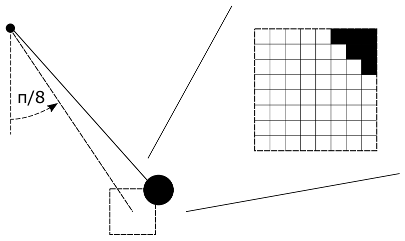

Figure 4.2.: A patch is extracted from the picture around where we believe the pendulum is.

4.4. Bayesian filtering

To fully understand the particle filter, it is necessary to know Bayesian filtering. For Bayesian filtering we need a model of how the system evolves over time,

$$\vec{x}_{k+1} = f_k(\vec{x}_k, \vec{w}_k), \tag{4.4}$$

the system equation and to be able to determine how likely we are to observe a measurement,

$$\vec{y}_k = h_k(\vec{x}_k, \vec{v}_k), \tag{4.5}$$

the measurement equation. The process noise $\vec{w}_k$ is assumed to be a white noise sequence with known probability density function, pdf [20]. The same holds for the measurement noise $\vec{v}_k$. k describes which time step we are at. Equation (4.4) will let us calculate $P(\vec{x}_{k+1}|\vec{x}_k)$, how the state of the pendulum changes over time. Equation (4.5) will let us calculate $P(\vec{y}_k|\vec{x}_k)$, how likely we are to observe a measurement given a state of the system. For example, if we count the number of dark pixels in a region far away from the pendulum's position, we would expect a low score.

The a priori distribution, $P(\vec{x}_{k+1}|\vec{y}_{1:k})$, is our best guess of the system's new state before receiving a new measurement. As described by Gustafsson [6], the goal of the Bayesian filtering is to calculate the a posterior distribution, $P(\vec{x}_{k+1}|\vec{y}_{1:k+1})$, our updated belief of the system's new state after receiving a new measurement.

For doing this, we utilize Bayes' theorem,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(A,B)}{P(B)}. \tag{4.6}$$

We have,

$$P(\vec{x}_{k+1}|\vec{y}_{1:k+1}) = P(\vec{x}_{k+1}|\vec{y}_{1:k}, \vec{y}_{k+1}), \tag{4.7}$$

which using equation (4.6) gives us

$$P(\vec{x}_{k+1}|\vec{y}_{1:k}, \vec{y}_{k+1}) = \frac{P(\vec{x}_{k+1}, \vec{y}_{1:k}, \vec{y}_{k+1})}{P(\vec{y}_{1:k}, \vec{y}_{k+1})}. \tag{4.8}$$

Also, using Bayes' theorem we have

$$P(\vec{x}_{k+1}, \vec{y}_{1:k}, \vec{y}_{k+1}) = P(\vec{y}_{k+1}|\vec{x}_{k+1}, \vec{y}_{1:k})P(\vec{x}_{k+1}, \vec{y}_{1:k}), \tag{4.9}$$

and

$$P(\vec{x}_{k+1}, \vec{y}_{1:k}) = P(\vec{x}_{k+1}|\vec{y}_{1:k})P(\vec{y}_{1:k}). \tag{4.10}$$

Inserting (4.10) into (4.9) and the result into (4.8), we obtain

$$P(\vec{x}_{k+1}|\vec{y}_{1:k}, \vec{y}_{k+1}) = \frac{P(\vec{y}_{k+1}|\vec{x}_{k+1}, \vec{y}_{1:k})P(\vec{x}_{k+1}|\vec{y}_{1:k})P(\vec{y}_{1:k})}{P(\vec{y}_{1:k}, \vec{y}_{k+1})}. \tag{4.11}$$

The Markov sensor assumption,

$$P(\vec{y}_{k+1}|\vec{x}_{k+1}, \vec{y}_{1:k}) = P(\vec{y}_{k+1}|\vec{x}_{k+1}), \tag{4.12}$$

tells us that the current measurement only depends on the current state, not the prior measurements. Equation (4.12) simplifies (4.11) into

$$P(\vec{x}_{k+1}|\vec{y}_{1:k}, \vec{y}_{k+1}) = \frac{P(\vec{y}_{k+1}|\vec{x}_{k+1})P(\vec{x}_{k+1}|\vec{y}_{1:k})P(\vec{y}_{1:k})}{P(\vec{y}_{1:k}, \vec{y}_{k+1})}. \tag{4.13}$$

Again, using Bayes' theorem we have, $P(\vec{y}_{1:k}, \vec{y}_{k+1}) = P(\vec{y}_{k+1}|\vec{y}_{1:k})P(\vec{y}_{1:k})$. Inserting into (4.13) and canceling out the terms, we get

$$P(\vec{x}_{k+1}|\vec{y}_{1:k}, \vec{y}_{k+1}) = \frac{P(\vec{y}_{k+1}|\vec{x}_{k+1})P(\vec{x}_{k+1}|\vec{y}_{1:k})}{P(\vec{y}_{k+1}|\vec{y}_{1:k})}. \tag{4.14}$$

Equation (4.14) is the key equation and allows us to recursively update the belief, $P(\vec{x}_{k+1}|\vec{y}_{1:k})$ when we obtain a new measurement $\vec{y}_{k+1}$.

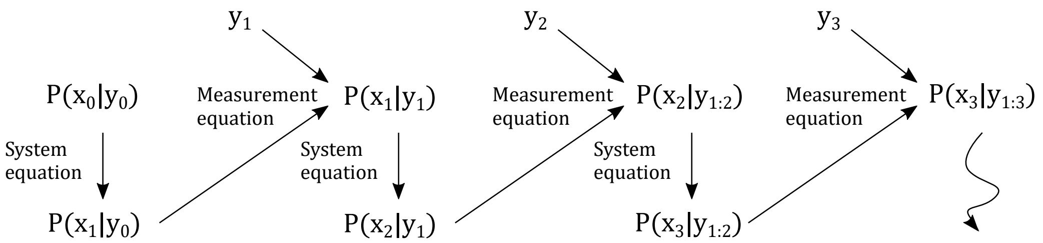

Figure 4.3.: The system equation propagates the estimate in time and the measurements update the estimate with the measurement equation.

The denominator of (4.14) is simply a normalization term and is found by

$$P(\vec{y}_{k+1}|\vec{y}_{1:k}) = \int P(\vec{y}_{k+1}|\vec{x}_{k+1})P(\vec{x}_{k+1}|\vec{y}_{1:k})d\vec{x}_k, \tag{4.15}$$

an integral over the numerator of equation (4.14) for all state vectors. $P(\vec{y}_{k+1}|\vec{x}_{k+1})$ in the equation above is available from the measurement equation (4.5).

$P(\vec{x}_{k+1}|\vec{y}_{1:k})$ in (4.14) and (4.15) is found from

$$P(\vec{x}_{k+1}|\vec{y}_{1:k}) = \int P(\vec{x}_{k+1}|\vec{x}_k)P(\vec{x}_k|\vec{y}_{1:k})d\vec{x}_k. \tag{4.16}$$

An intuitive explanation of this is that the probability of transitioning into a new state is the probability of being in a prior state times the probability of a transition from this prior state to the new one, summed up for all possible prior states. $P(\vec{x}_{k+1}|\vec{x}_k)$ in the above equation is available from the system equation (4.4). Now, when we have a transition, we calculate $P(\vec{x}_{k+1}|\vec{y}_{1:k})$ with (4.16). If we receive a new measurement, we use (4.14) to calculate $P(\vec{x}_{k+1}|\vec{y}_{1:k}, \vec{y}_{k+1})$ as illustrated in figure 4.3.

4.5. Particles

As mentioned, the particle filter is a numerical implementation of the Bayesian filtering. The idea is to generate a set of hypotheses, utilize the system equation to predict where each particle will move, weight the particles according to the measurement equation, discard the ones that are unlikely and generate new hypotheses around the ones that are most likely. Then, the process is repeated, starting from the system equation. Our best guess of the true state will be the weighted mean state of all the particles. In our example, this will be the weighted average angle and angular velocity. Note, however, that a particle can represent a state-space with a higher dimensionality than two. As an example, we can imagine our pendulum being replaced by a hook. Now, not only the angle and angular velocity of the pendulum is interesting, we also want to know how the hook is rotated around the radial axis and how quickly it rotates. We can represent this in a particle by letting it hold a value for both the angle and velocity in the θ-direction and for the angle and velocity around the radial axis. In this case, the dimensionality of the state-space has increased from two to four.

4.6. Initialization of particles

When we first start tracking the pendulum, it must be somewhere on the circular sector with a distance of L from the pivot. Where on the circular sector, we do not know. If we assume that the pendulum will obtain a maximum angular offset of $\frac{\pi}{6}$, we can initialize our particles by a uniform sampling from $[-\frac{\pi}{6}, \frac{\pi}{6}]$. The initial angular velocity is chosen to be zero. Also, for each particle, i, we assign a weight, w^i. The weights should be distributed in such a way that for all, N, particles, the weights sum to one, $\sum_{i=1}^{N} w^i = 1$. Initially, we set $w^i = \frac{1}{N}$, for all particles as we have not performed any measurements yet. If we knew the initial distribution $P(\vec{x}_0)$, we could have distributed the particles according to this.

For a particle, x_k^i, the superscript, i, indicates which of the N particles we are referring to and the subscript, k, indicates which time step we have come to.

4.7. The transition model

Between each measurement, when we analyze the picture, the pendulum will move. To account for this movement, we apply the transition model, an approximation of the system equation. Since we are working on a pendulum motion, we can use the pendulum dynamics to predict where the particles will have moved. The equations for this have already been developed in section 4.2. Equation (4.1) will predict the new angular velocity of the pendulum. (4.2) will predict the new angle of the pendulum, as shown in figure 4.4. Assuming that we had an accurate state estimate, the transition model will propagate the estimate in time and make sure that next time we perform a measurement, we still have a more or less accurate estimate. The measurement is then used to update the belief in the particles after the transition.

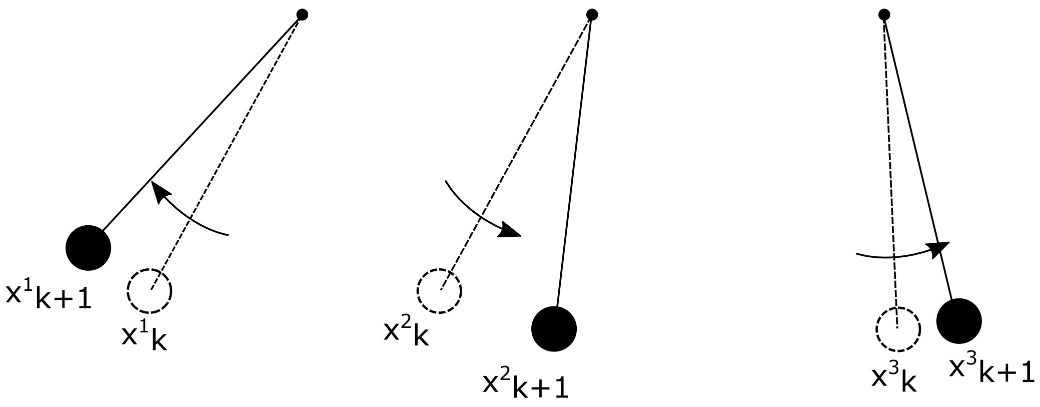

Figure 4.4.: The particles are propagated in time.

4.8. The sensor model

We want a more formal way of telling which particles that are probably close to the correct state than what was presented in section 4.3. In our simple sensor model, we use the pixel intensities directly to update the weights. For every particle, we use its estimate of the angle to project it into the image plane. Then, we count the number of dark pixels surrounding the projected point in the image. This value we multiply by the current weight of the particle, before we normalize. Now, all the particles have an updated weight describing how likely they are to correctly estimate the true state. The proposed method is an implementation of the measurement equation (4.5).

4.9. Resampling

Using the sensor model, we assigned a weight to every particle. These weights are used in the resampling step of the algorithm. According to the sensor model, particles with a low weight are less probable to correctly estimate the true state of the system than particles with higher weights. The idea is that to better utilize all the particles, particles are moved to the location of particles with high weight. Actually, in some of the literature, the particle filter goes under the name "Survival of the fittest" [10]. This is due to the resampling step of the particle filter where the next generation of particles build on the ones that we believe are best adapted to the current state of the system.

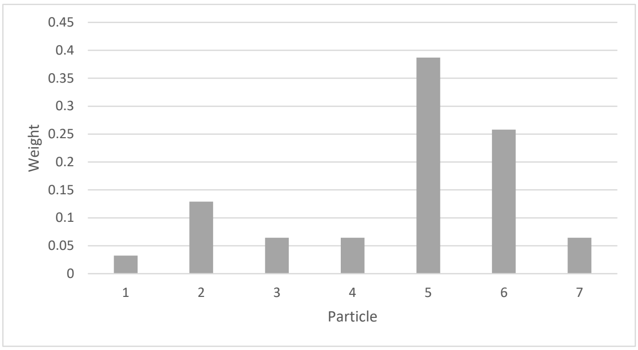

Figure 4.5.: An example of seven particles with weights assigned.

Different methods for choosing where to move particles exist. Systematic resampling [12] is a well-established option. A cumulative sum of the weights is calculated and a uniformly distributed weight, $r\epsilon[0,1]$ is chosen. In figure 4.6, we can see how the weights of figure 4.5 are accumulated and a random weight, r, is visualized by a horizontal line. Note that the first time the line crosses a column is at particle 5. In figure 4.5, we see that this corresponds with the particle with the highest weight. The higher the weight of a particle, the larger the step in the cumulative weight vector will be, and the more likely it is that r will be somewhere in the step up to this particle.

4.10. The mean and convergence

After the resampling step, we can calculate the weighted mean, $\sum_{i=1}^{N} \vec{x}_k^i w_k^i$, of the particles. This mean is our best guess of what is the true state of the system. Naturally, our goal is that this mean will be as close to the true state of the system as possible. Typically, the particles will be spread out over the state space when the filter is initialized, but as time passes and measurements are gathered, the filter should start to converge.

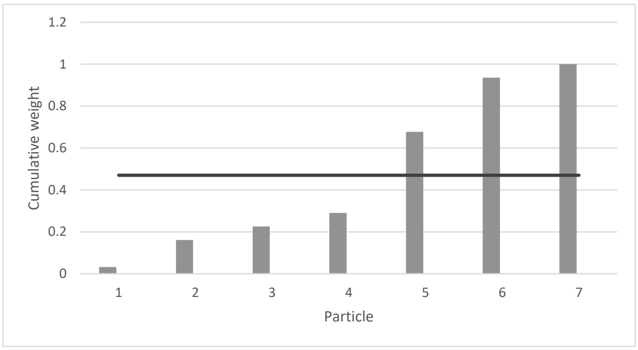

Figure 4.6.: The cumulative distribution of weights seen in figure 4.5 and a randomly chosen value.

4.11. Implementation issues and tuning

In this section, some of the challenges of the particle filter and some established remedies are discussed.

Assume that we do not account for any noise at all. Particles that are assigned a high likelihood score from the sensor model will observe that other particles are resampled as exact copies of themselves. As they now have the same value for θ and ω, they will transition in exactly the same way and obtain the same likelihood scores. The particles have become copies of each other and they give no more power in the estimation than what a single particle would have given. The process is called sample impoverishment [20] and, in the end, further resampling will cause every particle in the filter to become equal.

Roughening [5] is a method for avoiding sample impoverishment. The idea is simply to add random noise to each particle after the resampling step. In our pendulum example, we do not have a model for the process noise or the measurement noise. If we had such a model, we could have added noise after the transition and in the measurement. However, roughening is about adding artificial process noise in order to spread out the particles in the state space. For the pendulum, this could mean modifying both the angle and the angular velocity of the particles a little right after the resampling step. Typically, the noise added during roughening is drawn from a zero-mean Gaussian distribution. The standard deviation of the Gaussian should be tuned to the particular implementation.

Another parameter of the particle filter that can be tuned is the number of particles. More particles means more hypotheses of the state of the system and a higher probability that many of the particles accurately estimate the state. Especially, if we are dealing with a high-dimensional state space, we need many particles as the probability of accurately estimating values across all the dimensions is lower. Correctly guessing a number between 0 and 100 is easier than correctly guessing two numbers between 0 and 100. Unfortunately, increasing the number of particles also increases the computational complexity. This is a problem for real-time applications.

The resampling step is necessary to leverage from all the particles as much as possible. As described in [6], without the resampling, all but a few particles would have negligible weights after a while. Resampling can, for example, be performed every fifth iteration of the particle filter. Another method can be to randomly resample some of the particles every iteration. Unfortunately, the resampling erases the information held by the weights and thereby increases uncertainty. Therefore, we would like to perform resampling only when needed. When it is needed can be suggested by the effective number of samples [13],

$$\hat{N}_{eff} = \frac{1}{\sum_i (w_k^i)^2}, \tag{4.17}$$

where w_k^i is the weight of particle i at time step k. If a single particle has a weight of one and all the other particles have weights of zero, we obtain $\hat{N}_{eff} = 1$. Contrary, if all particles share the same weight of $\frac{1}{N}$, we have $\hat{N}_{eff} = N$. We can set a threshold, N_{th} such that if $\hat{N}_{eff} < N_{th}$, we should perform resampling. This ensures that resampling is performed only when needed.

Optimally, the sensor model should be able to score a particle directly according to how close it is to the correct state. As our sensor model relies on counting pixels, we have a discrete input to the particle filter. One of the problems with this is that a particle can be assigned a score of zero, but in reality it is never zero probability of a particular sensor reading no matter the real state of the system. To cope with this we can assume that the likelihood scores of the measurements are exponentially distributed, as in [3].

$$L(\vec{x}_{k+1}^i) = P(y_{k+1}|\vec{x}_{k+1}^i) \propto exp(-\lambda(1 - \frac{s_{k+1}^i - s_{min,k+1}}{s_{max,k+1} - s_{min,k+1}})) \tag{4.18}$$

In the above equation, s^i_{k+1} is the score we obtain directly from the measurement. With our method of counting the number of dark pixels in a 21 by 21 pixel square, this value will be between 0 and 441. $s_{min,k+1}$, the minimum value, will here be zero and $s_{max,k+1}$, the maximum value, will be 441. λ is a parameter that should be tuned according to if we want a high probability of a high scoring particle, relative to a low scoring particle, or not. The higher the λ, the larger the difference in probability of a low scoring and a high scoring measurement. With this method, a measurement of zero dark pixels will no longer lead to a score of zero, but still a lower score than a measurement of several dark pixels.

Implementations of the particle filter in both *MATLAB* and C++ are found in attachments and is explained in the appendix.

Chapter 5.

CUDA and Particle Filter Acceleration

5.1. The GPU and parallelization

GPU is an abbreviation of graphics processing unit. Originally GPUs were built to handle the stages required for 3D graphics generation, typically for video-games. Today, GPUs are increasingly being utilized in scientific contexts. This is because they are suited for parallel computations. If a task is both computationally expensive and possible to parallelize, utilizing a GPU can be beneficial. An example of a task that can be parallelized is the addition of two vectors, element by element. If we imagine a single person having to add two vectors of length 16, he or she would have to do 16 additions. If we instead imagine four persons that can co-operate on the same task, person one can add element one, two, three and four. Person two can add element five, six, seven and eight, and so on. Assuming that negligible time is spent deciding who should perform what and that they do not disturb each other, we have achieved a speed-up by a factor of four.

GPUs are suited for parallel calculations because they consist of many cores. While a modern day CPU seldom has more than eight cores, GPUs can have over a thousand cores. Typically, a single GPU core runs slower than a CPU core, but the total processing power of a GPU across all its cores will outperform a CPU. In this project report, a GPU of the type Nvidia GeForce GTX Titan was used for parallel computations. This GPU consists of 2688 cores grouped into parts of 192. Each part is called a streaming multiprocessor (SM), for this specific GPU we have $2688 \div 192 = 14$ SMs.

Particle filters are parallelizable to a great extent. This is why an effort is made in this project to implement parts of the algorithm on a GPU. The best example of this is the motion model where the particles are propagated in time. In this step, all the particles are completely independent. If we have as many cores as we have particles, a single core only has to propagate a single particle. Not all the steps of a particle filter are parallelizable, but these parts can be computed on the CPU.

5.2. Programming a Nvidia GPU

Programming a Nvidia GPU is typically performed using *CUDA*. This is a platform for parallel computing with a syntax much like C++. In *CUDA*, a *kernel* is a function that can be executed on a single core of the GPU. By executing a kernel on several cores at a time, we can do parallelized computations. Every core executes the same steps, but with different input. In the example where two vectors were added, each core is like a person and the kernel is the instruction to add the elements. When a kernel is running on a core it is called a *thread*. The threads are organized into *blocks* and the blocks are organized in a *grid*. Fortunately, for simple usage, we do not have to think about how this organization relates to which cores in which SMs that perform the computations.

In the code snippet 5.1, an example of a simple function is shown, **cpu_add()**. The function takes in three vectors and an integer telling it the length of the vectors. For a position, **i**, the i-th element in **z** is set equal to the sum of the i-th elements in **x** and **y**. A little further down in the snippet 5.1, a kernel called **gpu_add()"** is implemented. If we compare it with **cpu_add()**, we note that instead of looping through every single element of the vectors, we start at **index**, skip **stride** number elements before adding the next two numbers. A single one of the for-loops will do the calculations for $\frac{n}{stride}$ positions in the vectors. To calculate the sum for all positions, we need to run the for-loop **stride** number of times, each starting at a different position.

To run the kernel, we call **gpu_add<<<a, b>>>(n, x, y, z)**. **b** will be **blockDim.x** that we find in the kernel. This is the number of threads (instances of a kernel) that we want in a block. **a** will be the number of block in the grid, **gridDim.x**. This is the way that the programmer can decide how to organize the threads into blocks. If we call **gpu_add()** with four threads per block, three blocks and vectors of length 36, the kernel will be executed as shown in figure 5.2. A single thread will have to add numbers at three positions. The first thread

```
void cpu_add(int n, int* x, int* y, int* z) {
    for (int i = 0; i < n; i += 1) {
        z[i] = x[i] + y[i];
    }
}

cpu_add(n, x, y, z);

__global__
void gpu_add(int n, int* x, int* y, int* z) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < n; i += stride) {
        z[i] = x[i] + y[i];
    }
}

int threadsPerBlock = 4;
int numBlocks = 3;
gpu_add<<<numBlocks, threadsPerBlock>>>(n, x, y, z);

__global__
void gpu_add2(int n, int* x, int* y, int* z) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < n) {
        z[index] = x[index] + y[index];
    }
}

gpu_add2<<<9, 4>>>(int n, int* x, int* y, int* z);
```

Figure 5.1.: A simple kernel.

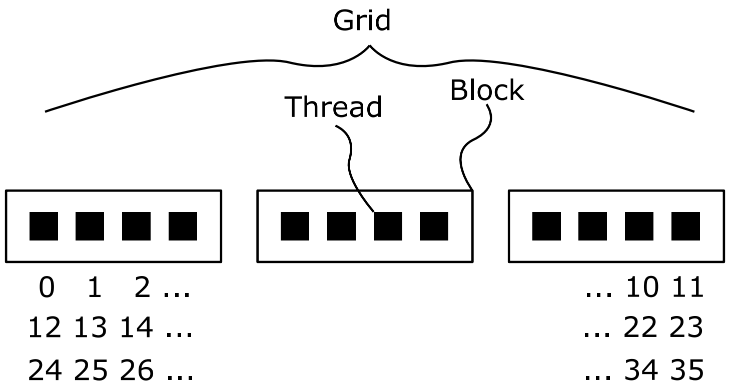

Figure 5.2.: Organization of threads and blocks in a grid

will work on position 0, 12 and 24. The second one at 1, 13 and 25, and so on. The stride will in this example be $3 \cdot 4 = 12$. The indices will be from 0 to 11. As an example, thread number 6 will use an index of 5 as it is the second thread in the second block, $1 \cdot 3 + 2 = 5$. Often, kernels are implemented even simpler. By initializing the same number of threads as there are elements in one of the vectors, the for-loop is avoided. A thread will now only process a single position, as shown in **gpu_add2()**, calling it with $9 \cdot 4 = 36$ threads. Note that if more threads are initialized than there are elements in a vector, we will get an "index out of range"-error. As shown in the code 5.1, a control of this should be included inside the kernel.

More on *CUDA* programming can be found in a series of blog posts from NVIDIA[1].

5.3. Memory transfers

Not all computations can be easily parallelized. For these operations, GPUs are not well suited. The resampling step of the particle filter is an example of this. In this part of the algorithm, the particles are not independent of each other. Firstly, a cumulative weight list is calculated from all of the particles. Secondly, particles are "cloned" and their states are copied over to other particles. Efforts can be made to partially parallelize these operations, but a serial implementation is more intuitive. As the clock speed of a CPU core is typically higher than that of a GPU core, a non-parallelized resampling runs faster on the CPU. To not overly complicate the particle filter implementation, a solution where some parts

[1] https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/

are run on the CPU and others on the GPU was opted for. The CPU acts as the "master" and the GPU as the "slave" that, when possible, does the heavy-lifting. This setup calls for memory transfers between the two.

The memory transfers are interesting to analyze because they take up significant time. Even though the calculations might run faster on the GPU than the CPU, in some cases it can be more efficient to not use the GPU because with memory transfers, an all-CPU implementation can be faster. First off, the transition model is analyzed.

Calculating the transition requires the state of all particles. With the 2D pendulum, the state-space is two-dimensional and consists of the angle and the angular velocity. With 32-bit (4 byte) floating point numbers describing the state and a memory transfer to the GPU and back to the CPU (four transfers in total) and $2^{16} = 65536$ particles, we obtain,

$$4 \cdot 65536 \cdot 4 \text{ byte} = 1 \text{ MB}.$$

This memory transfer takes place over the PCIe. For PCIe 2.0, the speed limit is 8 GB/s[2]. A memory transfer of 1 MB can then be assumed to require,

$$\frac{1}{8000} = 0.125 \text{ ms}.$$

The sensor model is more interesting. To run it on the GPU, the image must be transferred. A 1920 by 1080 grayscale image consisting of 8 bit (1 byte) pixels requires $1920 \cdot 1080 \cdot 1$ byte ≈ 2 MB. Furthermore, the angles and weights are required, but only the weights need to be transferred back to the CPU, resulting in a total of three transfers. Once again assuming floating point and 65536 particles,

$$3 \cdot 65536 \cdot 4 \text{ byte} + 2 \text{ MB} \approx 2.8 \text{ MB}.$$

The time usage becomes,

$$\frac{2.8}{8000} = 0.350 \text{ ms}.$$

For the necessary data transfers there are several techniques that can be used to achieve a speed-up. Different arrays can be combined into one array that is transferred. This eliminates some of the overhead cost of allocating different arrays on the GPU. Further, one can take advantage of a concept called *streams*. A stream is a sequence of operations that are executed in order on the GPU. By

[2]https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

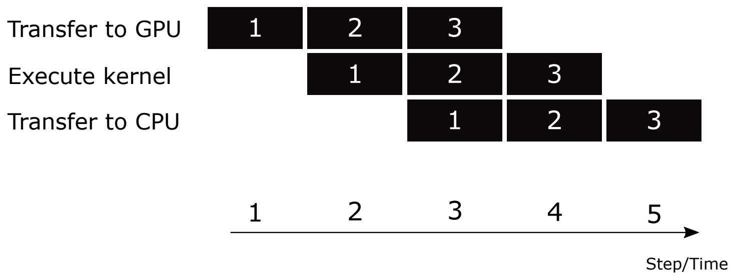

Figure 5.3.: Three streams executed on the GPU.

default, a single stream is defined "behind the scenes". However, several streams can be defined by the programmer and operations across different streams can be interleaved. The point here is that a GPU can have data transferred to it, compute a kernel, and transfer data back to the CPU, all at the same time. As an example, we can think of the motion model of the particle filter. The array of angles can be split up into several sub-arrays. Each of these sub-arrays can be processed in separate streams. For three streams, we have the situation of figure 5.3. At step three, the GPU utilization is at a max. The last set of angles (3) are transferred to the GPU, the angles in the middle (2) are updated in the kernel and the first angles (1) are already updated and are transferred back to the CPU. This asynchronous method is clearly more efficient than first transferring all the angles to the GPU, then computing the kernels and lastly transferring back as all parts of the GPU can be utilized at once.

5.4. Computational complexity

For the particle filter to run fast enough, efficient implementations of the algorithms are crucial. With an increasing number of particles, an increased amount of computing is needed. As discussed, we would like to be able to run the particle filter with as many particles as possible to increase accuracy. $2^{16} = 65536$ particles or even more can be necessary in some circumstances.

In the transition model, all the particles are independent and are processed individually. Calculating the transition of one particle requires the same effort no matter the number of total particles. This results in an algorithm complexity of $\Theta(n)$, where n is the number of particles. The exact same holds for the sensor model.

The resampling step of the particle filter poses some challenges in its implementation. The particles are no longer independent of each other since particles change state depending on other particles. With the proposed method including a cumulative weight array, this array is calculated in linear time, $\Theta(n)$. However, for all particles, which other particle they will become a clone of has to be determined. This is performed by looking up where a random number between 0 and 1 fits in the cumulative weight array. Naively looking through the array from the start until the right position is found results in a complexity of $O(n^2)$. For all particles, on average, we must look through half of the array before the right position is found. With 65536 particles, this technique is not feasible. 65536^2 is above four billion and for an algorithm that must be run several times per second, this will not work. Instead a binary search can be performed for every particle. This implementation lowers the complexity to $\Theta(n \log_2(n))$ which also becomes the complexity of one whole iteration of the particle filter.

Chapter 6.

The Kalman Filter

The Kalman filter is a popular filter. If the system has Gaussian distributions on time propagation uncertainty, sensor noise and state, the Kalman filter is proved to deliver the minimum variance and maximum likelihood estimate [6]. However, compared to the particle filter, the Kalman filter has some limitations. Firstly, it requires a linear state space model. Secondly, it can only estimate a single state. This means that it is not suited for tracking multiple objects. In the following, it will be shown that for our objective of tracking a single 2D pendulum, these limitations can be overcome. The benefit of a Kalman filter implementation over a particle filter one is a lower computational complexity.

6.1. The state space model

Just like the particle filter, the Kalman filter consists of a time propagation and a sensor update. Equations telling us how the system should evolve can predict a new state (a priori) of the system and input from sensors can update this prediction (posterior). For a particle filter, the belief in a state is represented by how close this state is to the weighted mean of the particles' states. The Kalman filter uses a Gaussian distribution to represent the belief. Our best guess of the real state will be the mean of this distribution. However, we are not only interested in the mean, but also the covariance of the distribution. Together, the mean vector and the covariance matrix can tell us how likely it is that the real state is within some arbitrary region. Applying this in the pendulum example, we could for example estimate the probability that the current state is within $\theta \in [-\frac{\pi}{6}, -\frac{\pi}{7}]$ and $\omega \in [0.1, 0.5]$.

The linear state space model can be presented as in [6], starting with the transition

$$\vec{x}_{k+1} = F_k\vec{x}_k + G_{u,k}\vec{u}_k + G_{v,k}\vec{v}_k.$$

F_k is the prediction matrix that implements the time propagation. $G_{u,k}$ and $\vec{u}_k$ is the control matrix and the control vector. In our case, we do not have any control input, the pendulum is swinging freely with the pivot point fixed, and this element can be neglected. $G_{v,k}$ and $\vec{v}_k$ represents the uncertainty of the transition. Assuming time invariant uncertainty, we can replace this by random Gaussian noise, $\vec{v}$. Finally, we assume a time-invariant prediction matrix, F. The simplified system equation becomes

$$\vec{x}_{k+1} = F\vec{x}_k + \vec{v}. \tag{6.1}$$

For the sensor update, a mapping between a state and the corresponding expected sensor input is needed.

$$\vec{y}_k = H_k\vec{x}_k + D_k\vec{u}_k + \vec{e}_k \tag{6.2}$$

The sensor mapping is assumed time-invariant and becomes H. Again, we have no control input and $D_k\vec{u}_k$ can be removed. $\vec{e}_k$ is the random Gaussian sensor noise and, when time-invariant, it can be written as $\vec{e}$. With all simplifications, we arrive at

$$\vec{y}_k = H\vec{x}_k + \vec{e}. \tag{6.3}$$

As mentioned, the mean and the covariance is used to represent the belief, $E(\vec{x}_k) = \vec{\mu}_k$ and $Cov(\vec{x}_k) = \Sigma_k$. Furthermore, the covariance of the noise is denoted as $Cov(\vec{v}) = Q$ and $Cov(\vec{e}) = R$.

6.2. The transition model

In the case of the 2D pendulum, the transition model looks a lot like in the case of the particle filter, equation (4.3). Strictly spoken, the 2D pendulum is not a linear system as the update of the angular velocity depends on the sine of the angle, $\omega_{t+1} = \omega_t - \frac{g sin(\theta_t)}{L}\Delta t$. However, for small angles, we have that $sin(\theta) \approx \theta$.

With a linear approximation of the state transition, the prediction matrix can be written as

$$F = \begin{bmatrix} 1 & \Delta t \\ -\frac{g}{L}\Delta t & 1 \end{bmatrix}. \tag{6.4}$$

6.3. The sensor model

In the particle filter, the sensor model took the state of a particle, projected it onto the image plane and analyzed the image in the area suggested by the particle. This approach will not work for the Kalman filter. Now, the sensor input should be a single suggestion of a state that can be directly compared to the state suggested by the transition model. For the pendulum case study, we must be able to find where in the image we believe the pendulum is and project this to the object space. Then, we can compare the suggestion of the sensor input with the suggestion of the transition model.

There are several methods available for finding where in an image an object is located. Here, an easy and general approach was implemented, inspired by [4]. The idea is that if you keep the camera fixed and only the object you want to track is moving, it is possible to find out, more or less, what the background looks like by comparing several consecutive frames. If the full picture of the background is known, the moving object can be found by simply subtracting the background from the frame. Ideally, the result should be an image with all zeros except where the object is.

Mathematically, the estimation of the background can be found by

$$B_{i+1} = K \cdot B_i + (1 - K) \cdot I_{i+1} \tag{6.5}$$

where B is the representation of the background, I is the newly obtained frame and K is a constant that determines how much the new frame should contribute in the update of the background. B can be initialized as a uniform dark image and should over time depict the frame as if the moving object was not there at all. Further, the object can be segmented out by $I_{diff,i} = abs(I_i - B_i)$. Lastly a threshold is applied, $I_{object,i} = I_{diff,i} > T$. This is done to filter out noise resulting from the background changing a little and to produce a binary image with a value of 1 at the position where the object is believed to be located. The process is illustrated in figure 6.1.

With the object segmented out, we can find the centroid in the image as

$$x_{c,i} = \frac{1}{total} \sum_{x=1}^{w} \sum_{y=1}^{h} x \cdot I_{object,i}(y, x) \tag{6.6}$$

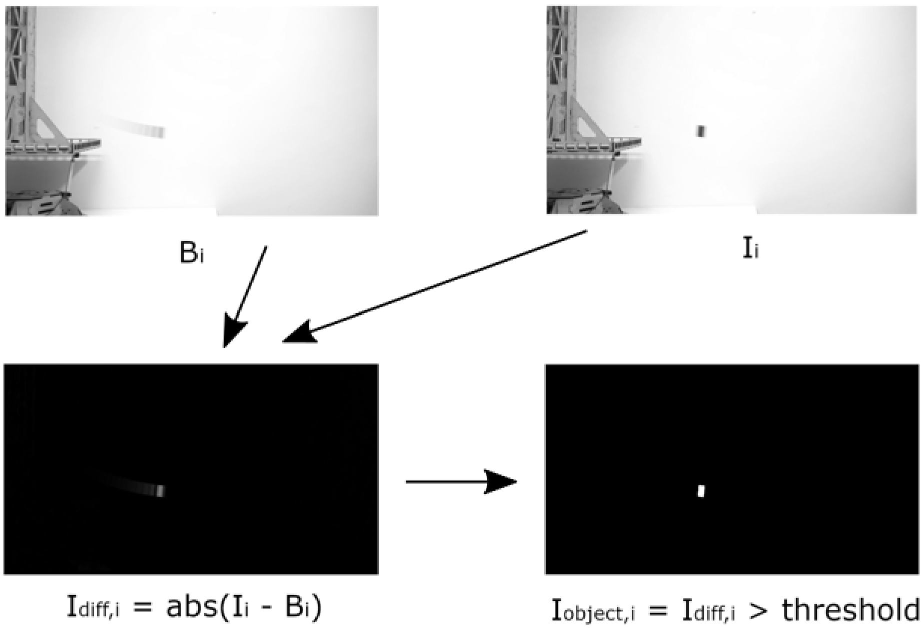

Figure 6.1.: Segmentation process.

and

$$y_{c,i} = \frac{1}{total} \sum_{y=1}^{h} \sum_{x=1}^{w} y \cdot I_{object,i}(y, x) \tag{6.7}$$

where $I_{object,i}(y, x) = 1$ if the pixel is one of the segmented pixels and zero if not. h is the height of the image and w is the width. *total* is the total number of segmented pixels, $\sum_{y=1}^{h} \sum_{x=1}^{w} I_{object,i}(y, x)$.

Lastly, the θ suggested by the sensor input can be found by projecting the point to the object frame and converting to polar coordinates.

$$\tilde{\vec{s}} = K^{-1}\tilde{\vec{p}} = K^{-1} \begin{bmatrix} x_{c,i} \\ y_{c,i} \\ 1 \end{bmatrix}$$

$$\vec{r}^{c}_{cp} = z\tilde{\vec{s}}$$

$$\tilde{\vec{r}}^{o}_{op} = T^{c\,-1}_{o}\tilde{\vec{r}}^{c}_{cp} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\theta = atan2(y, x) + \frac{\pi}{2}$$

With θ determined, the H-matrix of the sensor model can be specified. The relationship between the real angle and the angle suggested by the sensor input should be one to one. Unfortunately, we are not able to extract information about the angular velocity from the image. All in all, the mapping from the physical state to the expected sensor input, the H matrix, turns out to be simply

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix}.$$

$$\vec{z} = \begin{bmatrix} \theta \end{bmatrix}.$$

6.4. The Kalman filter algorithm

The Kalman filter algorithm consits of a prediction by the transition model followed by an update of what is called the Kalman gain and lastly an update from the sensor model. In the following, the steps will be explained for a time-invariant system without external influence. More details can be found in [1].

A prediction step is carried out as

$$\vec{\mu}_{k+1} = F\vec{\mu}_k \tag{6.8}$$

and the update of the covariance

$$\Sigma_{k+1} = Cov(F\vec{\mu}_k) + Q = FCov(\vec{\mu}_k)F^T + Q = F\Sigma_k F^T + Q. \tag{6.9}$$

Σ_{k+1} will have greater values than Σ_k as the uncertainty in the prediction is added with Q.

After the transition, the Kalman gain is updated with the following equation,

$$K' = \Sigma_{k+1}H^T(H\Sigma_{k+1}H^T + R)^{-1}. \tag{6.10}$$

The sensor input updates the prediction,

$$\vec{\mu}'_{k+1} = \vec{\mu}_{k+1} + K'(\vec{z}_{k+1} - H\vec{\mu}_{k+1}). \tag{6.11}$$

$\vec{z}_{k+1}$ is the received sensor input and K' is the Kalman gain. K' describes how

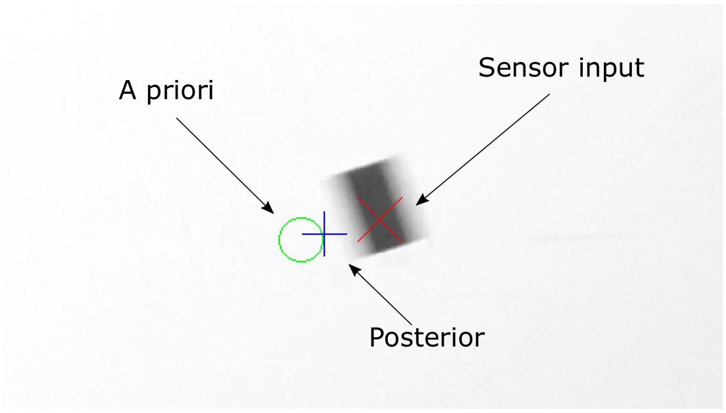

Figure 6.2.: Sensor input and prediction combined into the output of the Kalman algorithm (posterior).

much we trust the information from the sensor. The higher the values of K', the more the sensor information is able to influence the update of the state prediction, $\vec{\mu}'$. In figure 6.2 $\vec{\mu}_{k+1}$, $\vec{\mu}'_{k+1}$ and $\vec{z}_{k+1}$ is illustrated from left to right. Here, the output of the Kalman filter is clearly closer to the prediction than the sensor input. This is because the Kalman gain at this moment is

$$K' = \begin{bmatrix} 0.2894 \\ 0.3467 \end{bmatrix}. \tag{6.12}$$

The angle estimate of the Kalman filter is only shifted 0.2894 of the way from the prediction towards the sensor input.

Like the mean, the covariance also has to be updated as the sensor input is received. The following equation performs the necessary update,

$$\Sigma'_{k+1} = \Sigma_{k+1} - K'H\Sigma_{k+1}. \tag{6.13}$$

In figure 6.3, the prediction mean and uncertainty is shown for the same moment of time as in the image of figure 6.2. The mean and uncertainty after the prediction is at the top (a priori) and at the bottom the updated distribution is shown (posterior). One can note that this distribution is in a large degree similar to the distribution before the sensor update, but the uncertainty is lower and the mean is drawn towards the mean of the sensor.

Summed up, an iteration of the Kalman filter algorithm for a system without no

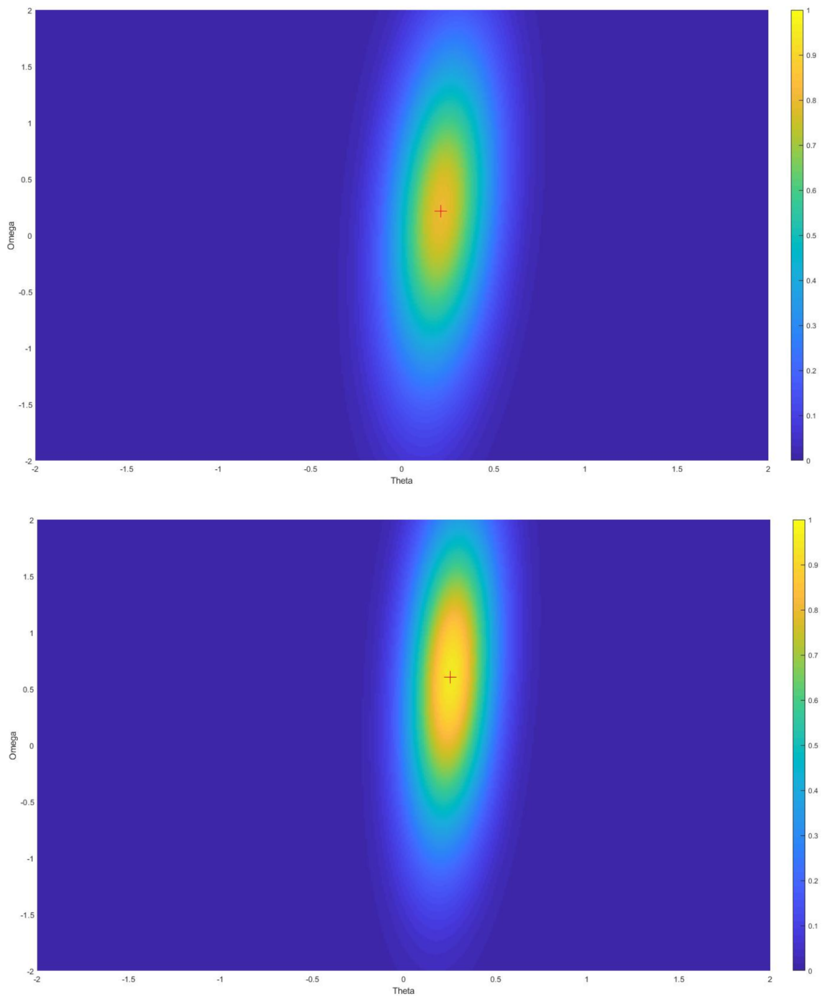

Figure 6.3.: The a priori distribution on top is updated and becomes the posterior distribution on the bottom.

control input and time-invariant time update and sensor mapping can be summed up in three steps. First, the prediction with equation (6.8) and (6.9). Then, the update of the Kalman gain with equation (6.10). Lastly, the sensor update with equation (6.11) and (6.13).

6.5. Determine covariance matrix of prediction uncertainty and sensor noise

The Kalman filter algorithm requires the prediction matrix, F, the sensor mapping, H, the covariance of the prediction uncertainty, Q, and the covariance of the sensor noise, R. F is given by the nature of the system and H is determined by how the state relates to the form of the sensor input. Q and R is somewhat more challenging to set.

Q determines how the uncertainty of the state changes during the prediction. $Q = Cov(\vec{v})$ and the matrix can be set such that the Wiener process originating from $\vec{v}$ of equation (6.1) evolves as desired. For example, we can require the angle of the pendulum to drift from standstill to 5 degrees after one second and adjust Q such that this is satisfied.

R is a property of the sensor. If you have a system with sensor input from a laser or similar, this property is typically specified. If not given, a method for determining R is to find the variance of the error of the sensor reading. While keeping the pendulum at an angle of zero, measurements can be made and the variance of these measurements will be the variance of the sensor noise, R.

6.6. Computational complexity

The main motivation to chose a Kalman filter implementation over a particle filter is limited computational power. The only variables that the algorithm needs to keep track of are stored in the mean state vector, the covariance of the mean and the Kalman gain. A particle filter, on the other hand, needs to store the state of every particle. Typically, more than 1000 particles are needed and for every iteration all particles must be updated by both the transition model and the sensor model. Historically, these facts have favoured the Kalman filter for real-time applications, but as mentioned, GPU accelerations of the particle filter can make it competitive also in real-time.

6.7. Implementation issues

The covariance matrix, P, should be positive definite and symmetric. Due to inaccuracies in the computations, we might end up with a P that does not fulfill this. If the matrix turns out not to be symmetric after an iteration of the Kalman filter, the matrix can be replaced with $\frac{1}{2}(P + P^T)$. Controlling if the matrix is positive definite and mitigating this is harder. A solution can be a *square root implementation* as in [6]. This implementation will at the same time make sure that P is always symmetric.

Outliers received from the sensor should be handled. The mean state is updated according to the difference between the received input and the expected input, equation (6.11). If the Kalman filter has converged and a significant outlier is received, this can move the mean state away from convergence. When an outlier is detected, one can do an iteration of the Kalman filter where only the prediction is performed and not the sensor update.

A implementation of the Kalman filter in *MATLAB* is found in the attachments and is explained in the appendix.

Chapter 7.

Tuning Parameters

7.1. Sensor

To compare the particle filter and the Kalman filter the same sensor input will be used for both. This will be the version that builds up an image of the background and subtracts this from the current frame, as explained in section 6.3. The first parameter, K, is apparent in equation (6.5) and controls to what degree the current frame contributes to the updated background image. The second one, T, is the parameter that determines how great the difference in pixel intensity has to be between the frame and the background before a region is labelled as foreground.

The root mean square error (RMSE) between the ground truth angle and the sensor estimate over a sequence of 250 frames was used as a performance measure.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{\theta}_i - \theta_i)}$$

θ_i is the ground truth, $\hat{\theta}_i$ is the estimate and N is the number of frames. For every combination of $K = [0.1, 0.2, ..., 0.9]$ and $T = [0.1, 0.2, ..., 0.9]$, the RMSE was calculated. The best result of 0.0034 rad was achieved with K set to 0.9 and T set to 0.3.

numParticles	resampleThresh	lambda	roughFraction	roughStd
64	0.5	1	0.3	0.15
256	0.65	5	0.5	0.3
1024	0.85	9	0.8	0.45

Table 7.1.: The parameters tested for the particle filter.

7.2. Particle filter

For the particle filter, five parameters were identified. These parameters were tuned independently from the sensor parameters, keeping the sensor parameters fixed to $K = 0.9$ and $T = 0.3$. To find reasonable regions for the particle filter parameters, some initial tests were run. Then, the particle filter was executed on the sample video with all combinations of the parameters seen in table 7.1. *numParticles* is the number of particles. *resampleThresh* is the threshold controlling when to resample and is related to equation (4.17). *lambda* is the parameter for the exponential distribution of the measurement, equation (4.18). *roughFraction* is the fraction of the particles that will have noise added after a resampling. *roughStd* is the standard deviation of the Gaussian noise that is added to a particle's angular velocity when roughening.

In total, $3^5 = 243$ runs were performed. The error between the particle filter estimate and the ground truth was captured for every frame of a 250 frame long sequence where the filter had converged. RMSE was once again used as a measure of accuracy. The set of parameters obtaining the lowest RMSE of $2.6549 \cdot 10^{-3}$ rad was the following.

- *numParticles*: 1024
- *resampleThresh*: 0.8
- *lambda*: 9
- *roughFraction*: 0.8
- *roughStd*: 0.3

Additionally, a gradient descent optimization approach was initialized around these parameters for further refinement, but no significant improvements were made from this. Note however that many different sets of parameters performed almost as good as this particular set. No particular set was significantly better than all the others.

7.3. Kalman filter

Q and R was set according to the material presented in section 6.5. The diagonal elements of Q were set to $\sigma_\theta^2 = 0.0001$ and $\sigma_\omega^2 = 0.0003$. Over 1000 runs of one second each this gave a root mean square drift of 0.06 radians. R was found to be approximately 0.000011.

Chapter 8.

Filtering Results

8.1. Accuracy of the particle filter

Like in the last section, the RMSE was calculated over a sequence of 250 frames were the filter had converged. This was done 100 times and the mean RMSE was $2.8 \cdot 10^{-3}$ rad.

In figure 8.1, an execution of the particle filter is shown. Here, the estimate is displayed together with the standard deviation. The weighed mean is calculated as earlier, $\sum_{i=1}^{N} \theta_i w_i$. The weighted standard deviation is calculated as

$$\sqrt{\frac{N}{N-1}\sum_{i=1}^{N} w_i(\theta_i - \mu)^2}.$$

N is here the number of particles, θ_i and w_i are respectively the angle and weight of particle number i. The estimation error of the same execution is shown in figure 8.2.

8.2. Speed of the particle filter

The speed tests were performed on the Ubuntu 18.04.1 OS with a Intel i7-3820 CPU (4 cores at 3.6 GHz). Two Corsair 8 GB 1333 MHz served as memory and the computer had two Nvidia GeForece GTX Titan (2688 cores at 0.837 GHz) available. Despite having two GPUs available, only one was utilized.

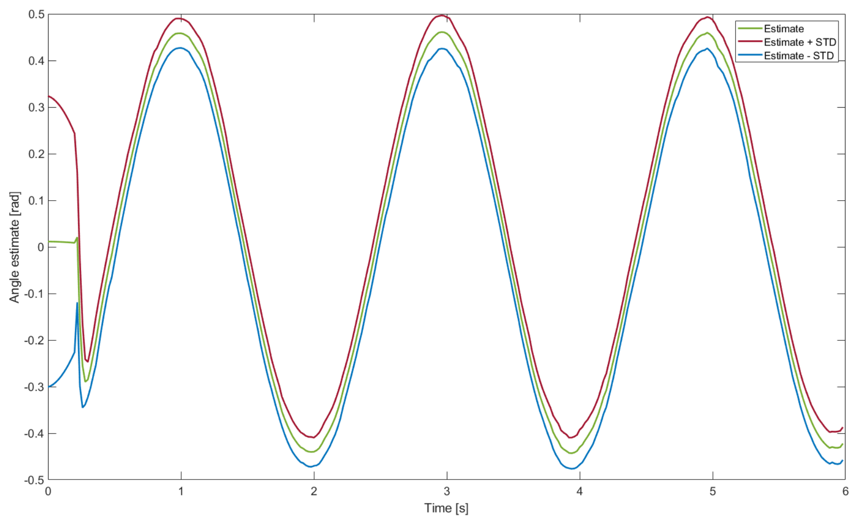

Figure 8.1.: Particle filter angle estimates with standard deviation.

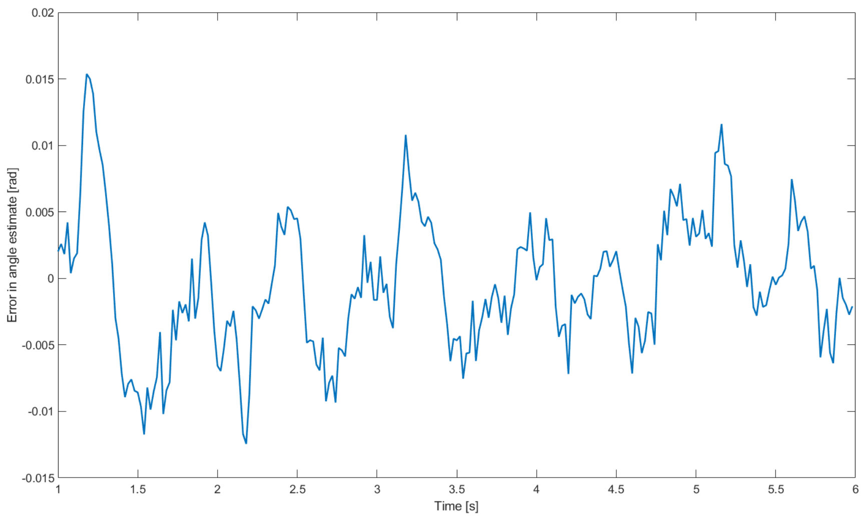

Figure 8.2.: Error in particle filter estimate.

Algorithm	Only CPU [ms]	GPU accelerated [ms]
Reading image	6.035	5.914
Image processing	39.323	39.666
Transition	0.216	3.931
Sensor	11.219	8.625
Resample	0.052	0.049
Finding mean	0.003	0.003
Total	56.847	58.187

Table 8.1.: Average processing time per frame with 1024 particles.

Algorithm	Only CPU [ms]	GPU accelerated [ms]
Reading image	5.983	6.233
Image processing	39.378	39.764
Transition	8.035	5.965
Sensor	13.859	11.585
Resample	22.942	23.268
Finding mean	0.637	0.651
Total	90.833	87.467

Table 8.2.: Average processing time per frame with 262144 particles.

For 1024 particles, the processing time per frame for the different algorithms is shown in table 8.1. The results are taken as an average over all the 347 frames in the recorded video. An increase in the number of particles to 262144 yields the results seen in table 8.2. The algorithms that were implemented with GPU acceleration are the ones for the transition and for the sensor update. The operation of reading in the frames is performed on the CPU by the OpenCV function **cv::read()**. Image processing is also performed by OpenCV CPU functions and includes conversion to grayscale 32-bit floats, addition, scalar multiplication, absolute value and thresholding. Functionality for resampling of the particles and finding the mean was written from the bottom and up for the CPU.

8.3. Accuracy of Kalman filter

With the same sensor input as the particle filter over the same sequence of frames, the Kalman filter achieved a RMSE of $3.3 \cdot 10^{-3}$ rad. The estimate is shown in figure 8.3 and the error can be seen in figure 8.4. The standard deviation for each

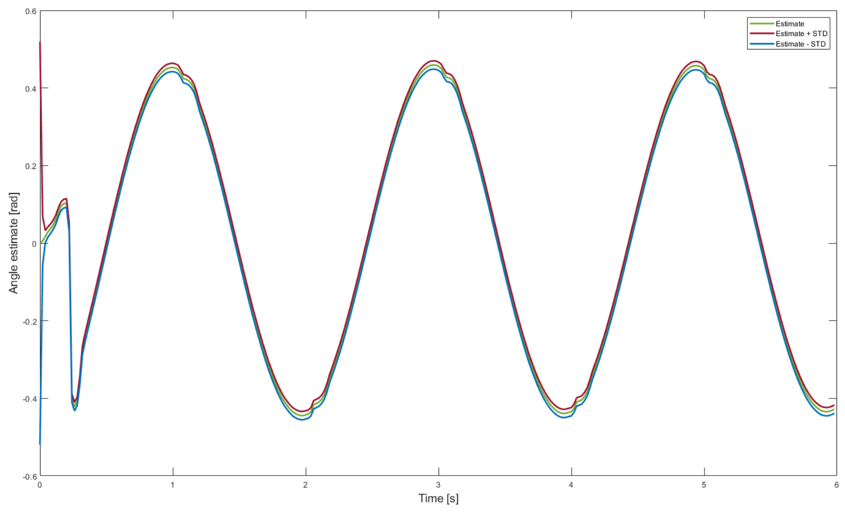

Figure 8.3.: Kalman filter angle estimates with standard deviation.

iteration is found in the covariance matrix, P. Analysis of the execution time of the Kalman filter is not provided here as it is much less computational complex than the particle filter.

8.4. Without filtering

The error of the sensor input without filtering is displayed in figure 8.5. As already stated, the RMSE is $3.4 \cdot 10^{-3}$ rad in this case.

8.5. With artificial noise

To simulate a more noisy sensor output and demonstrate the benefits of filtering, an artificial Gaussian noise with a standard deviation of 0.02 radians and a mean of zero was added to the sensor output. This caused the RMSE of the particle filter to increase to $9.0 \cdot 10^{-3}$ rad. The parameters of the particle filter were not adjusted to this particular case. For the Kalman filter, the RMSE grew to $12.6 \cdot 10^{-3}$ rad. However, the R in the Kalman filter was set to $0.02^2 = 0.0004$ to match the additional sensor noise. The results of the filtering with artificial

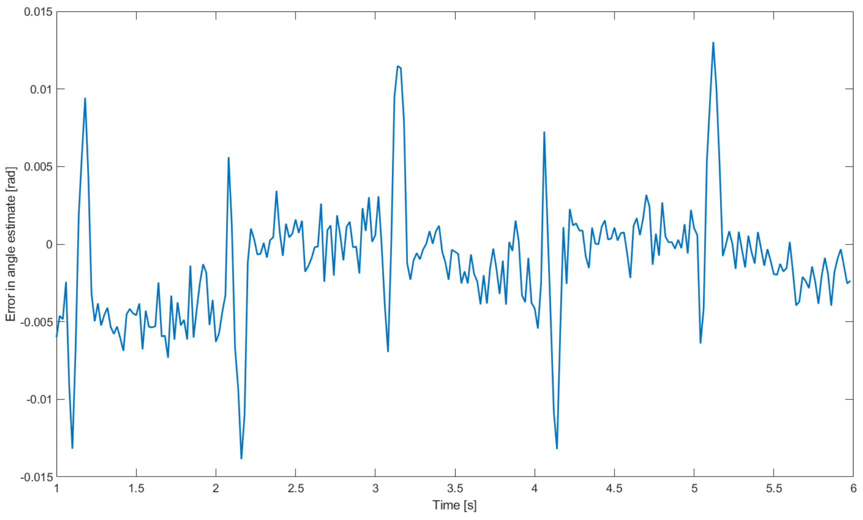

Figure 8.4.: Error in Kalman filter estimate.

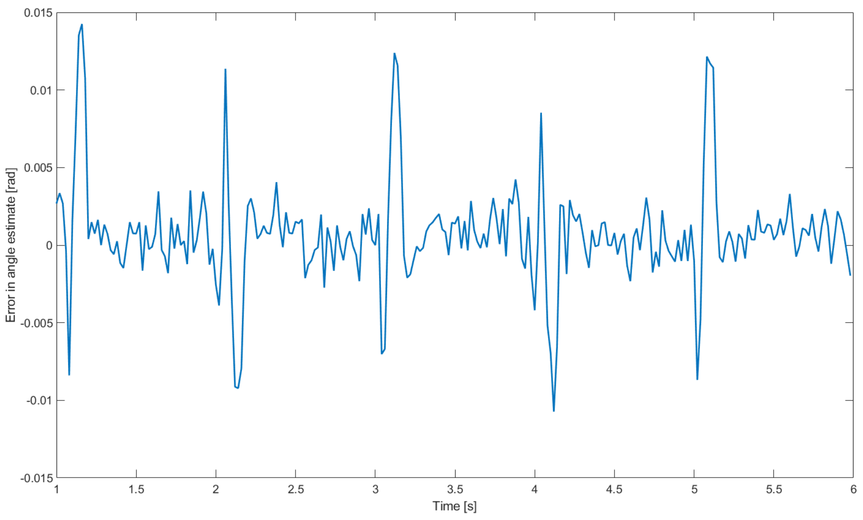

Figure 8.5.: Error in angle estimate when only applying the sensor model.

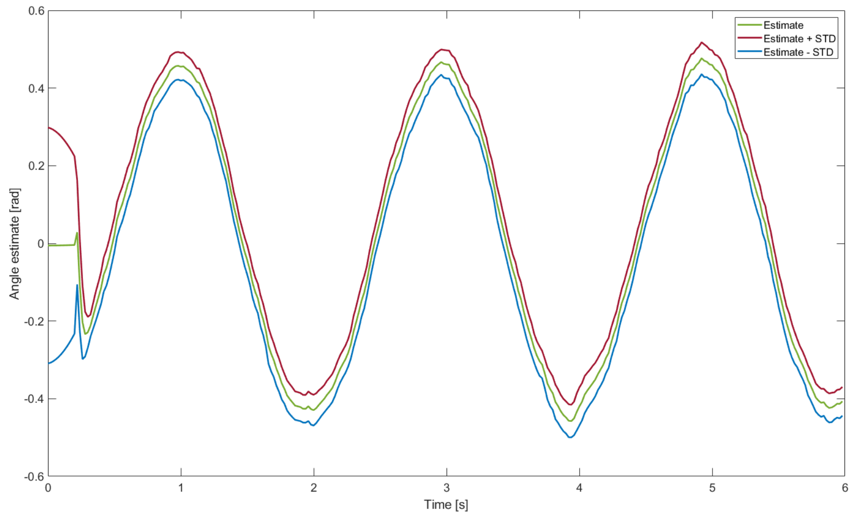

Figure 8.6.: Angle estimate of the particle filter with added sensor noise.

sensor noise can be seen in figure 8.6 and 8.7.

8.6. Alternative sensor model

The performance of the proposed sensor model of section 4.3 is presented in the following. For an execution with 1024 particles, the processing time is shown in table 8.3. Table 8.4 displays the results when using 262144 particles. Compared to section 8.2, the difference lies in the image processing and the sensor model. In this case, the image processing only consists of a conversion to grayscale performed on the CPU.

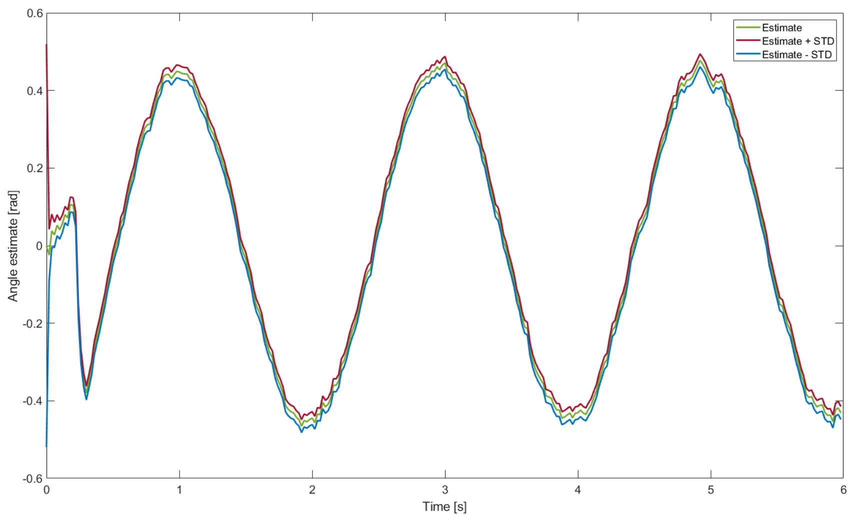

Figure 8.7.: Angle estimate of the Kalman filter with added sensor noise.

Algorithm	Only CPU [ms]	GPU accelerated [ms]
Reading image	6.801	9.536
Image processing	6.669	7.052
Transition	0.219	4.499
Sensor	14.847	4.268
Resample	0.061	0.066
Finding mean	0.003	0.003
Total	28.599	25.424

Table 8.3.: Average processing time per frame with 1024 particles.

Algorithm	Only CPU [ms]	GPU accelerated [ms]
Reading image	6.049	6.049
Image processing	6.378	6.455
Transition	12.533	8.553
Sensor	1312.464	21.530
Resample	17.646	17.954
Finding mean	0.637	0.643
Total	1355.706	61.184

Table 8.4.: Average processing time per frame with 262144 particles.

Chapter 9.

Discussion

Both the Kalman filter and the particle filter implementation performs well in terms of accuracy. With an RMSE of $2.8 \cdot 10^{-3}$ rad, the particle filter outperforms the Kalman filter by a small margin, but the Kalman filter's achieved RMSE of $3.3 \cdot 10^{-3}$ rad is still a solid result. For a rope length of 0.9611 m, the RMSE of the Kalman filter can be approximated as $3.3 \cdot 10^{-3}$ rad $\cdot$ 0.9611 m $\approx$ 3.2 mm. However, it should be noted that this result is partially due to an accurate sensor model. On its own, the sensor model is able to achieve a RMSE of $3.4 \cdot 10^{-3}$ rad which is, in practice, indistinguishable from that of the Kalman filter. This signals that in the test video, there is not a substantial need for filtering because the sensor noise is low. Furthermore, it is possible that the accuracy of the sensor input could be improved by a more thorough calibration procedure.

Myhre [17] reports that he was able to load parts with 5 mm tolerance without collisions. In other words, he experienced an accuracy in the same order of magnitude as the results seen in this report. However, the loading operation he demonstrates requires estimation of both position and orientation, adding up to three variables in total. In addition, he applies the method in a more cluttered environment than what has been demonstrated in this report. Yet another difference between the two experiments is the amplitude of the oscillation. Myhre's conveyor was swinging with an amplitude of about 2 degrees while here the tracking of a pendulum was demonstrated with an amplitude of 26 degrees. Hystad [8] experienced a RMSE of $5.2 \cdot 10^{-3}$ rad, $7.5 \cdot 10^{-3}$ rad and $34.3 \cdot 10^{-3}$ rad in each of the three angular variables when applying his particle filter on a simulated video. Once again, the accuracy is close to what was found in this report.

It can be argued that the test setup in this project was too simple. A dark pendulum on a more or less completely white and static background is easily segmented out. In a more realistic setting, the need for filtering would have been greater. To simulate a more complex environment with a noisy sensor input, artificial noise was added as described in section 8.5. In this setting, the particle filter once again outscored the Kalman filter with a small margin, $9.0 \cdot 10^{-3}$ rad against the Kalman filter's $12.6 \cdot 10^{-3}$ rad. Both of these are significantly better than the RMSE of the noisy sensor input. With a standard deviation of at least 0.02 rad, the RMSE of the non-filtered sensor input would be above $20.0 \cdot 10^{-3}$ rad in this setting. If we compare figure 8.6 with 8.7, it can be seen that the particle filter converges faster. After one third of a second, the particle filter has converged while it takes the Kalman filter about 1.5 seconds to converge. This is the greatest benefit of the particle filter over the Kalman filter that was found in this experiment.

Tracking difficulties in the region around the amplitude can be seen in figure 8.2 and 8.4. The estimation error when the pendulum is in the region around the maximum displacement is several times greater than elsewhere. Inspecting figure 8.5, it is seen that the error can be traced back to the sensor input. The problem seems to be that when the pendulum is moving slowly, the method of segmenting the foreground fails. When the pendulum occupies the same pixels over a period of several frames the algorithm includes it as part of the background. This means that for applications with small amplitudes and hence low speeds, the sensor model will perform poorly. In this setting, the sensor model proposed in section 4.3 will perform better.

The processing power can become a limiting factor when applying a particle filter. Typically, the number of particles required is above 1000 and as the computational complexity scales about linearly with the number of particles, powerful hardware is required. Table 8.1 displays the execution time for the different parts of the particle filter with 1024 particles. Note how the transition algorithm runs considerably faster on the CPU. This is because of the overhead of copying memory back and forth to the GPU. For a final version of a particle filter, all the algorithms should be implemented on the GPU to cut down on the overheads. However, this will make the code harder to read and less flexible. Thus, this was not carried out in this project. Increasing the number of particles to 262144, table 8.2, the execution time increases as expected. Note how the execution time of the transition and sensor algorithm not even doubles when the number of particles increase by a factor of 256. Once again a proof of how much of the execution time that is consumed by the overhead.

The bottleneck of the particle filter implementation is the image processing part. Operations for adding images together, thresholding and more from the OpenCV library seem to be too slow for real-time usage. A custom implementation of this on the GPU would probably reveal great speed-ups. Going back to 1024 particles, 60 ms per frame equates to an update frequency of about 17 Hz. If it was not for the slow image processing it would have been close to 50 Hz. Despite this, the robot controller would need a higher update rate and the solution would be to perform prediction to fill in the gaps between every image. To investigate if the alternative sensor model presented in section 4.3 would perform better, the execution time with this implementation is shown in table 8.3 and 8.4. Because this implementation requires less image processing, it performs better in terms of overall speed. Also, note how the GPU proves valuable for the sensor algorithm with 262144 particles. A lot more computation is required per particle in the sensor update than in the other algorithm and the GPU shows of its strengths.

Summing up, the particle filter and the Kalman filter provide estimates with similar accuracy for the purpose of visual tracking of a 2D pendulum. Despite the non-linearities of the system, a basic implementation of the Kalman filter proves to deliver satisfying results. The traditional argument against using a particle filter is the computational complexity it involves. But, with a modern GPU, this project and the work of Myhre [17] and Hystad [8] all show that real-time applications are feasible. Actually, for the simple application in this report, a single core implementation on the CPU seems to deliver a satisfying update rate. Some advantages of the particle filter over the Kalman filter are due to the Kalman filter's requirement of a Gaussian sensor model. The particle filter is more flexible in this regard and allows for easy combination of different sensor models. Often an exponential distribution is assumed in image processing and also the popular *Beam Based Sensor Model* that is typically used for LIDARs and similar does not follow a Gaussian distribution. In addition, it can be argued that the particle filter is easier to implement and understand. Dealing with the covariance matrices of the Kalman filter can be challenging.

To bridge the gap between the work presented in this report and a successful loading of a roof-mounted conveyor, a great effort remains. First and foremost, the estimation must be extended to 3D and include orientation of the conveyor. For this purpose, stereo vision or a 3D camera will be required. Further, the difference in frame rate of the camera and the update speed required by the robot controller must be accounted for. Next, the changing center of gravity and the occlusions during loading is a challenge. Most of these challenges have already been solved by Myhre [17], but a more robust and general sensor model is lacking. With all the recent advances in machine learning it could be interesting to look into training a convolutional neural network. An idea is to let the neural network do the rough localization and then use some other method for a more accurate pose estimation within the area proposed by the neural network.

Chapter 10.

Conclusion

This project report has presented methods for visual tracking of a swinging pendulum. The focus has been on estimating the state in a simple pendulum motion. Despite this, the hope is that by taking the reader all the way from the underlying mathematical equations to an example of a working implementation, the foundation to solve other interesting visual tracking challenges is laid.

The particle filter and Kalman filter both proved to deliver high accuracy. No significant issues regarding convergence or similar was encountered in the experiment. Yet, it must be noted that a real-world production process will typically be more complex. Robustness against changing illumination and a dynamic background are among the challenges that must be solved before a system can be used in the industry.

References

[1] Tim Babb. *How a Kalman filter works, in pictures*. 2015. URL: https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/ (visited on 11/08/2018).

[2] Harry Butler and Matthew Lambert. *Nvidia GeForce GTX Titan Review*. 2013. URL: https://bit-tech.net/reviews/tech/nvidia-geforce-gtx-titan-6gb-review/1/ (visited on 12/11/2018).

[3] C. Choi and H. I. Christensen. "RGB-D object tracking: A particle filter approach on GPU". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 1084–1091. DOI: 10.1109/IROS.2013.6696485.

[4] Zhaoxia Fu and Yan Han. "Centroid weighted Kalman filter for visual object tracking". eng. In: *Measurement* 45.4 (2012), pp. 650–655. ISSN: 0263-2241.

[5] N Gordon, D Salmond, and A Smith. "Novel approach to nonlinear/non-Gaussian Bayesian state estimation." eng. In: *IEE Proceedings F. Radar and Signal Processing* 140.2 (1993), P.107–P.113. ISSN: 0956-375X. URL: http://search.proquest.com/docview/26116978/.

[6] Fredrik Gustafsson. *Statistical sensor fusion*. eng. Lund, 2012.

[7] David Held, Sebastian Thrun, and Silvio Savarese. "Learning to Track at 100 FPS with Deep Regression Networks". In: (2016).

[8] Håkon Hystad. *Tracking a Moving Object with an Industrial Manipulator using a Particle Filter*. eng. 2018. URL: http://hdl.handle.net/11250/2561320.

[9] Michael Isard and Andrew Blake. "CONDENSATION—Conditional Density Propagation for Visual Tracking". eng. In: *International Journal of Computer Vision* 29.1 (1998), pp. 5–28. ISSN: 0920-5691.

[10] Keiji Kanazawa, Daphne Koller, and Stuart Russell. "Stochastic Simulation Algorithms for Dynamic Probabilistic Networks". In: (2013).

[11] A.A Kassim, T Tan, and K.H Tan. “A comparative study of efficient generalised Hough transform techniques”. eng. In: *Image and Vision Computing* 17.10 (1999), pp. 737–748. ISSN: 0262-8856.

[12] Genshiro Kitagawa. “Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models”. In: *Journal of Computational and Graphical Statistics* 5.1 (1996), pp. 1–25. ISSN: 1061-8600.

[13] Augustine Kong, Jun S. Liu, and Wing Hung Wong. “Sequential Imputations and Bayesian Missing Data Problems”. In: *Journal of the American Statistical Association* 89.425 (1994), pp. 278–288. ISSN: 0162-1459.

[14] D Kragic and H Christensen. “Robust visual servoing”. eng. In: *International Journal of Robotics Research* 22.10/11 (2003), pp. 923–939. ISSN: 02783649. URL: http://search.proquest.com/docview/230034583/.

[15] Morten Lind. *Open real-time control and emulation of robots and production systems*. eng. Trondheim, 2012.

[16] Michael Montemerlo. *FastSLAM : A Scalable Method for the Simultaneous Localization and Mapping Problem in Robotics*. eng. 2007.

[17] Torstein Anderssen Myhre. *Vision-based control of a robot interacting with moving and flexible objects*. eng. Trondheim, 2016.

[18] Torstein Anderssen Myhre and Olav Egeland. “Tracking a Swinging Target with a Robot Manipulator using Visual Sensing”. eng. In: (2016). ISSN: 0332-7353. URL: http://hdl.handle.net/11250/2474248.

[19] Olivier Roulet-Dubonnet. *Distributed control of flexible manufacturing systems : implementation of a specialized multi-agent middleware and applications of holonic concepts*. eng. Trondheim, 2011.

[20] Dan Simon. *Optimal state estimation : Kalman, H [infinity] and nonlinear approaches*. eng. Hoboken, N.J., 2006.

[21] Paul Viola and Michael Jones. “Robust Real-Time Face Detection”. eng. In: *International Journal of Computer Vision* 57.2 (2004), pp. 137–154. ISSN: 0920-5691.

[22] N. Xu, A. Liu, Y. Wong, Y. Zhang, W. Nie, Y. Su, and M. Kankanhalli. “Dual-Stream Recurrent Neural Network for Video Captioning”. In: *IEEE Transactions on Circuits and Systems for Video Technology* (2018). ISSN: 10518215.

Appendix A.

Implementation details

A.1. OpenCV and *CUDA* on Linux

Linux will be needed to control the industrial manipulators because Windows cannot provide a sufficient update rate to the robot controller. Therefore, the final script for tracking should be written to run on Linux. The newest long-term support version of Ubuntu was chosen as the operating system, Ubuntu 18.04 LTS. The toolkit for *CUDA 10.0*[1] was installed following the instructions[2]. Also, OpenCV 3.4.3 for C++[3] was installed[4] to help with image processing tasks. Make sure that "Include Nvidia Cuda Runtime support" and "Use GTK version 2" is set to "ON" in the CMakeLists.txt-file, before running cmake.

A.2. Source Code

The code necessary to run the implementations presented in this report is given in the attachments. To make it easily readable, the code is divided into several files. Comments are added in the code, but also a short description of how the most essential files work is presented in the following sections. Boldface refers to files and functions in the filters.

[1]https://developer.nvidia.com/cuda-toolkit-archive
[2]https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html
[3]https://opencv.org/releases.html
[4]https://docs.opencv.org/2.4/doc/tutorials/introduction/linux_install/linux_install.html

A.3. Particle filter, *MATLAB* implementation

The particle filter is started by running **particle_filter.m**.

To implement a particle filter is relatively straight forward. If one is familiar with object-oriented programming, each particle can be represented as an object. Every particle object will keep track of its own weight and state. However, to make the code easily understandable also for readers with no experience in object-oriented programming, the filter is implemented without the use of this technique. Instead of a single array of objects, an array is used to hold the weights, and every dimension of the state-space has its own array. If we need all the properties of particle number i, we have to look at the i-th position in the weight array, the array of angles and the array of angular velocities.

First, the parameters of the particle filter, physical constants, parameters of the input video and more must be set (**set_globals()**). Before the iterations of the particle filter can begin, the particles have to be initialized (**initialize()**). This is achieved by assigning every element of the weight array with the same value, all angular velocities are set to zero and the angles are assigned randomly within zero plus minus a maximum angle. We are now ready to begin the iterations of the particle filter.

The first step of an iteration is the image processing (**image_processing()**) which implements the method for finding the pixels where there is motion as in equation (6.5). Then, prediction and equation (4.3) is applied for every particle (**motion()**). This gives us the a priori particle distribution. Next, the weights are updated with the sensor input (**sensor()**). The input is analyzed as in section 4.3, the same way as for the Kalman filter. A likelihood is assigned with equation (4.18). The old weight of the particle is multiplied with the likelihood and the weights are normalized. By this point we have the posterior distribution. Further, equation (4.17) is used to control if a resampling is needed (**should_resample()**). If $N_{eff} < N_{th}$, a resampling is performed (**resample()**) as described in section 4.9. Lastly, the filter's estimate of the angle is found as the weighted mean of the individual particles estimate of the angle.

A.4. Particle filter, C++/*CUDA* implementation

MATLAB is not an ideal programming language to use when working on particle filters. For larger numbers of particles the filter is computational demanding and requires a higher performing programming language. Therefore, the *MATLAB* code was re-written in C++ on Linux. OpenCV is used to read the frames of the video and to display the results. Other than this, no libraries outside the standard library is used.

With C++, we are able to pass on variables as references. This significantly cuts down on the required memory operations. As we have a lot of variables in the particle filter, this is one of the greatest benefits of C++ over *MATLAB* for our use. To speed up the calculations further, we can use the GPU as an accelerator. C++ allows for detailed control of the GPU and this topic is discussed in chapter 5.

The implementation is a lot similar to the one in *MATLAB*. However, the *MATLAB* code includes more comments and should be read if something is unclear. A difference from the *MATLAB* implementation is that random numbers are generated before the filtering begins and are found in the **NormRand** and **UniRand** classes. Also, a **Timer** class is included to analyze the performance of the different algorithms. **ParticleFilter.h** is responsible for setting up all the parameters and constants. In addition, this file specifies if the sensor and motion model should be run on the GPU or the CPU. Files with the prefix "d_" are the ones that are partially executed on the device, the GPU. The code is compiled and executed (starting from the root folder of the project) with the following commands.

```
$ cd build
$ cmake ..
$ make
$ cd ParticleFilter
$ ./ParticleFilter
```

To see the GPU utilization, the following command can be used.

```
$ nvidia-smi -l 1
```

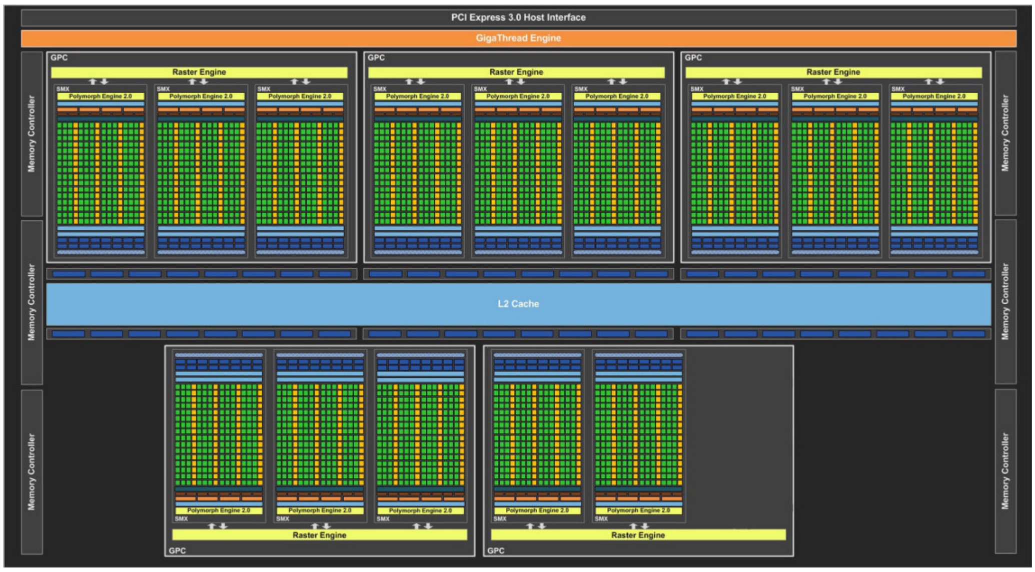

Figure A.1.: The architecture of a Nvidia GeForce GTX Titan GPU [2]
.

A.5. Important *CUDA* functions

- **___global___** Function decorator that signals that the function can be called from the CPU and the GPU.
- **___device___** Function decorator that signals that the function can only be called from the GPU (from inside a kernel).
- **cudaMalloc()** Allocate memory on the GPU.
- **cudaMemcpy()** Copy data from the CPU to the GPU or opposite.

A.6. Kalman filter, *MATLAB* implementation

The Kalman filter is started by running **kalman_filter.m**.

An implementation of the Kalman filter in *MATLAB* is simple once you have developed your equations for the system. All steps of the filter are implemented as matrix multiplications, operations which *MATLAB* supports by default. Starting of, the parameters of the Kalman filter, physical constants, parameters of the input video and more are set (**set_globals()**). Then, the filter is initialized with a mean state vector, $\vec{\mu}$, and the associated covariance matrix, Σ. For every iteration of the filter, equation (6.8) and (6.9) calculates the a priori distribution (**motion()**). The gain is updated with equation (6.10) (**update_gain()**). Then, the posterior distribution is found by reading in a frame from the video and applying equation (6.11) and (6.13) (**sensor()**). The image processing is the same as in the particle filter.