Streaming Video Completion using a Tensor-Networked Kalman Filter

S.J.S. de Rooij

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Streaming Video Completion using a Tensor-Networked Kalman Filter

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

S.J.S. de Rooij

July 13, 2020

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Copyright © Delft Center for Systems and Control (DCSC)
All rights reserved.

Abstract

In streaming video completion one aims to fill in missing pixels in streaming video data. This is a problem that naturally arises in the context of surveillance videos. Since these are streaming videos, they must be completed online and in real-time. This makes the streaming video completion problem significantly more difficult than the related video completion problem. State-of-the-art streaming video completion methods based on adaptive matrix completion, do not work well when the number of missing pixels is high ($\sim 95\%$). Therefore, in this report a new streaming video completion method will be introduced based on a tensor-networked Kalman filter. The results in this report will show that this Kalman filter method performs better than the state-of-the-art methods when the percentage of missing pixels is high ($\sim 95\%$).

Table of Contents

List of Figures

List of Tables

Preface & Acknowledgements

This document is a part of my Master of Science (in Systems and Control) graduation thesis. The subject of this thesis was proposed by my supervisor, as it has previously been the subject of a bachelor end project (BEP) [24]. In this thesis I will expand upon the research that has already been done. All the code that was used in order to produce the results of this thesis, can be found the following Git repository: https://gitlab.com/seline/thesis. Lastly, I cannot end this preface without thanking my supervisor dr.ir. K. Batselier for all his assistance during the writing of this thesis.

Delft, University of Technology
July 13, 2020

S.J.S. de Rooij

Chapter 1

Introduction

In *streaming video completion* one aims to fill in missing or removed pixels in streaming video data. These missing pixels, therefore, need to be restored in real-time. Furthermore, this completion can only be done based on past frames and the measurement of the current *corrupted* frame (Figure 1-1). This makes the streaming video completion more difficult than the related video completion problem. Where, in general, the full (offline) video is used to perform the completion [16, 37].

The corruption of streaming videos can occur for a variety of reasons. The focus of this thesis will be on the surveillance video application. In this case, camera footage can be suddenly obstructed. This could be due to hardware failure (e.g. dead pixels) or someone trying to cover-up the camera. In both cases it is important to still be able to see what is going on, preferably in real-time.

Because this report focuses on the video surveillance application, the following assumptions will be made in order to solve this problem. One is that we assume that we have access to uncorrupted video data prior to the corruption 'event'. So looking at Figure 1-1 there are $t_c - 1$ frames that can be used as prior information. Thus, it is possible to use information from past frames to perform the recovery of current frames. Another assumption is that the video cameras are fixed, which is often the case in video surveillance. This assumption makes it possible to subtract the background from the frame, which leads to a sparser representation of the data (Section 4-1-3). This makes it possible to represent the video data more efficiently (by using tensor-trains with smaller ranks, Section 4-2-2); as well as providing a better estimate of the video frame, since only the unknown 'foreground' of the frame is estimated.

Many state-of-the-art video completion methods are batch methods and use the full video to perform the completion [2, 14, 25, 46]. These methods either use patch-based completion techniques [17], often in combination with matrix completion, or they use tensor completion techniques on a tensor of the entire video [2, 46]. While this can give, especially in the case of tensor completion, an accurate reconstruction of the original video, these methods are not applicable to streaming videos. A possible way to solve this is to use a frame-by-frame approach, thus reducing the problem to an image completion problem. However, this

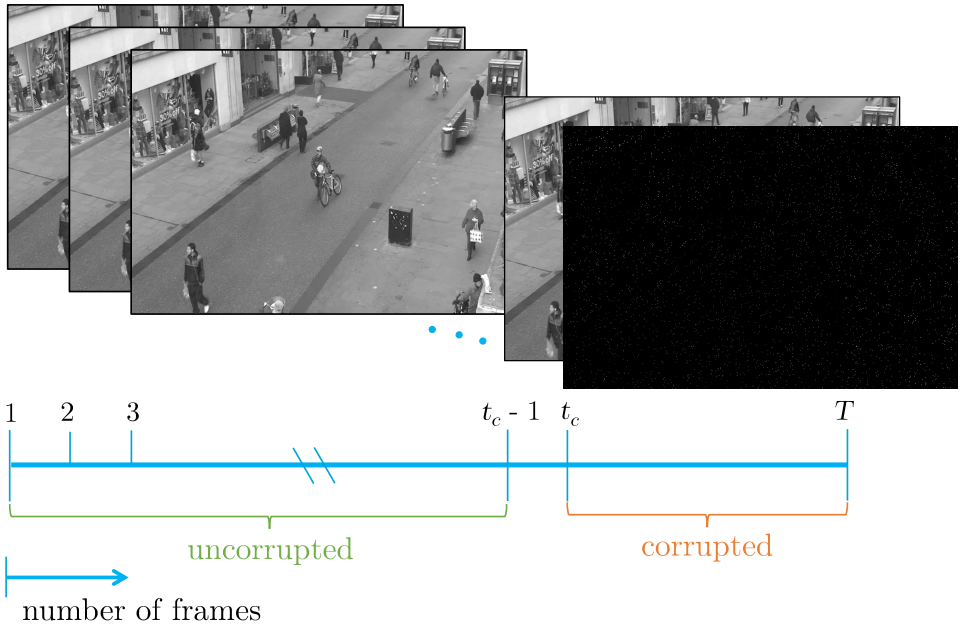


Figure 1-1: Illustration streaming video completion [1].

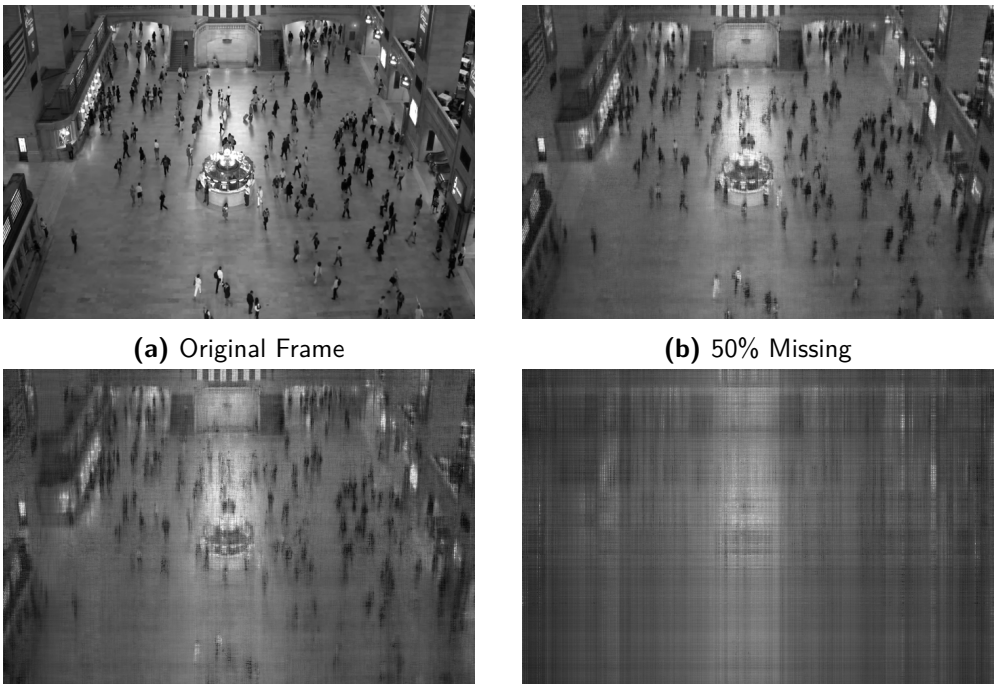
approach is rather naive, as it completely ignores the additional information provided in the temporal domain and could lead to a lack of temporal coherency.

Currently, there are a couple of methods that can be applied specifically to the streaming video completion. One is based on matrix completion and one is based on tensor completion: *adaptive matrix completion* [40] and *streaming tensor completion* [26]. How these methods work will be explained briefly.

Adaptive Matrix Completion In matrix completion one tries to recover a matrix from a small subset of its entries. In general this problem is ill-posed, it has an infinite number of solutions. However, when the underlying matrix is assumed to be of low-rank, the matrix completion problem is reduced to a rank minimization problem [3]. Because the rank function is non-convex in general and the minimization problem is NP-hard to solve, the rank function is often replaced by the nuclear norm: $\|\mathbf{X}\|_\star = \sum_i \sigma_i(\mathbf{X})$, where $\sigma(\mathbf{X})$ denotes the singular values of $\mathbf{X}$. The nuclear norm is the tightest convex bound of the matrix rank function [35]. Thus, the optimization problem becomes,

$$\begin{aligned} &\min_{\mathbf{X}} \|\mathbf{X}\|_\star \\ &\text{s. t. } \mathbf{M}(i,j) = \mathbf{X}(i,j) \quad (i,j) \in \Omega. \end{aligned} \tag{1-1}$$

where $\mathbf{M} \in \mathbb{R}^{M \times N}$ is a low-rank matrix that is observed over a subset Ω of its entries ($(i,j) \in \Omega$ if $\mathbf{M}(i,j)$ is observed), and $\mathbf{X}$ represents the completed low-rank matrix.



(a) Original Frame **(b)** 50% Missing

(c) 75% Missing **(d)** 95% Missing

Figure 1-2: Reconstruction of streaming video using the PLMS algorithm [1, 40]. **(a)** shows the original frame, **(b)**, **(c)** and **(d)** the reconstructed frames when 50%, 75% and 95% of the pixels are missing, respectively.

Adaptive matrix completion aims to complete a sequence of low-rank matrices:

$$\{\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \ldots, \mathbf{X}^{(T)}\}.$$

This method can be applied to the streaming video completion by setting the frames of the streaming video as the to be completed matrices (see Section 4-1-1 for how frames can be represented as matrices). The methods proposed in [40] are variations on the least mean square (LMS) algorithm [7], adding the nuclear norm as a regularizer to the cost function to ensure that the matrices are low-rank.

In Figure 1-2 one can see the result of using the proximal LMS (PLMS) algorithm from [40] applied to the streaming video completion problem ($\mu = 1$ and $\lambda = 0.8$). Clearly, when only 50% of the pixels is missing (Figure 1-2b) the reconstruction matches the original frame fairly well. However, when the percentage of missing pixels becomes much higher, the algorithm does not work as well. In the case that 95% of the pixels are missing the reconstructed frame does not resemble the original frame at all.

Streaming Tensor Completion Tensor completion is similar to matrix completion, however, instead of trying to recover a matrix, tensor completion aims to recover the missing entries of a tensor. In streaming tensor completion a new slice of the to-be-completed tensor becomes available at each time-step [26] (Figure 1-3). In case of a streaming video these slices would be equal to the frames of the video.

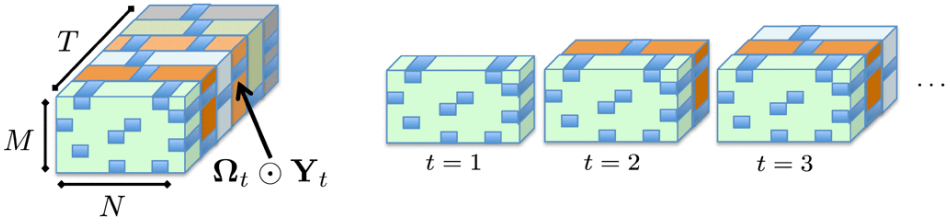


Figure 1-3: Illustration of streaming tensor completion [26]. $\mathcal{Y} \in \mathbb{R}^{M \times N \times T}$ is the to-be-completed tensor, $\mathbf{Y}_t \in \mathbb{R}^{M \times N}$ is the t-th slice of that tensor and $\mathbf{\Omega}_t \in \mathbb{R}^{M \times N}$ is a matrix containing the observations of $\mathbf{Y}_t$ (i.e. $\mathbf{\Omega}_t(m,n) = 1$ if $\mathbf{Y}_t(m,n)$ is observed).

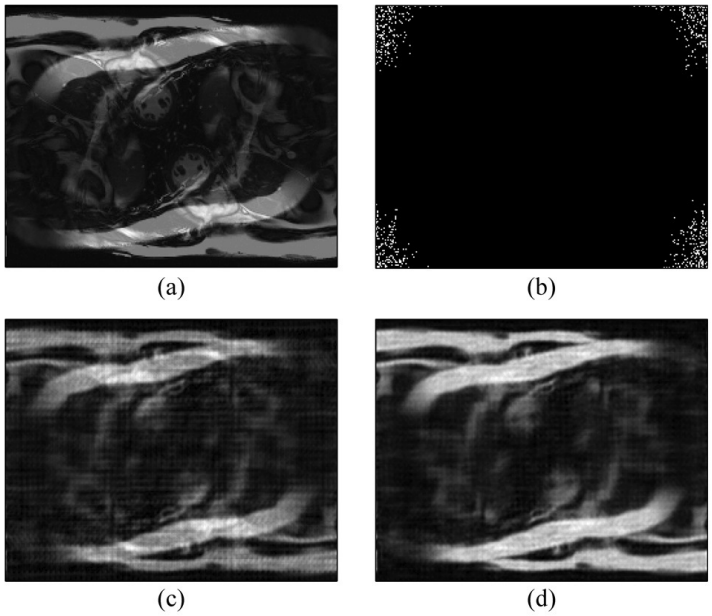


Figure 1-4: Dynamic MRI inpainting using TeOSGD [26]. **(a)** Original frame, **(b)** data undersampled by a factor of 4, reconstructed frame with **(c)** 25% available data and **(d)** 40% available data [26].

In [26] an algorithm was proposed to solve the streaming tensor completion problem, based on an online stochastic gradient descent (SGD) algorithm. This algorithm will be referred to as the online SGD algorithm for tensor decomposition and imputation (TeOSGD). In numerical tests TeOSGD was shown to be effective in the inpainting of dynamic cardiac MRI images (Figure 1-4), a problem related to the video completion problem. It can, however, take up to 100 seconds for the algorithm to learn the tensor subspace, so it can take 100 seconds before an accurate reconstruction can be made. For streaming video completion this is simply not fast enough.

Unfortunately the authors of [26] could not be reached for the code to implement the TeOSGD algorithm. Since it would be very time-consuming to implement this algorithm myself due to its complexity, and I simply did not have the time to do so, it was not possible to determine whether this method works if more than 75% of the pixels are missing.

Contribution of this Thesis

Because both of the above mentioned methods have their limitations, in this thesis a new method will be proposed to solve the streaming video completion problem, using a tensor-networked Kalman filter. The main idea behind using a Kalman filter for the video completion problem, is to model the frames of the video as the state-vectors ($\mathbf{x}[k]$) of a state-space system (1-2). The state-vectors are thus vectorizations of the video frames and the state-space system describes the changes in the video frames over time. The estimation of the state-vector is performed by the Kalman filter using the measurements of the uncorrupted pixels ($\mathbf{y}[k]$). Due to the size of the corresponding state-space system it is necessary to use tensor-networks to keep the calculations tractable.

$$\begin{aligned} \mathbf{x}[k+1] &= \mathbf{x}[k] + \mathbf{w}[k], \qquad \mathbf{w}[k] \sim \mathcal{N}(0, \mathbf{W}[k]) \\ \mathbf{y}[k] &= \mathbf{C}\,\mathbf{x}[k] \end{aligned} \tag{1-2}$$

This method was first proposed in [24] as part of a bachelor's thesis. Here, this method will be further improved and tested. Furthermore, it will be shown that the Kalman filter method outperforms the state-of-the-art methods based on adaptive matrix completion when the number of missing pixels is very high ($\sim 95\%$).

This thesis is structured as follows. First, some background information will be provided on tensors-networks (Chapter 2) and the Kalman filter (Chapter 3). Then the tensor-networked Kalman filter will be introduced in Chapter 4. Followed by the discussion of the results (Chapter 5), and some conclusions and recommendations for future research (Chapter 6).

1-1 Notation

The following notation conventions will be used. Vectors, matrices and tensors are denoted by boldface lowercase letters (e.g. $\mathbf{a}$), boldface uppercase letters (e.g. $\mathbf{A}$) and boldface Euler script letters (e.g. $\mathcal{A}$), respectively. Scalars are denoted by italic letters, these can be both lower- and uppercase (e.g. a or A). Usually, lowercase letters are used for indices and uppercase letters for the dimensions (or *modes*) of a tensor. For instance, an element indexed by $(i_1, i_2, \ldots, i_d)$ of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is denoted as $\mathcal{A}(i_1, i_2, \ldots, i_d)$.

Lower case italic letters can also be used as a more compact way to denote an element (or subset of elements) of a vector, matrix or tensor. In this case the indices are indicated in the subscript, e.g. $\mathcal{A}(i_1, i_2, \ldots, i_d) = a_{i_1, i_2, \ldots, i_d}$. Colons are used to indicate all, or a range of, elements of a dimension. The n-th row of a matrix is thus represented as $\mathbf{A}(n, :) = a_{n,:}$.

$\mathbf{A}^T$ and $\mathbf{A}^{-1}$ are used to denote the transpose and the inverse of a matrix $\mathbf{A}$, respectively. The singular values of a matrix $\mathbf{A}$ are denoted as $\sigma(\mathbf{A})$, and the i-th singular value (in descending order) is denoted by $\sigma_i(\mathbf{A})$. The Frobenius norm is denoted by $\|\cdot\|_F$ and the nuclear norm by $\|\cdot\|_\star$. $\odot$ is used to represent the Hadamard product and $\otimes$ to represent the Kronecker product. Table 1-1 gives an overview of all these notation conventions.

Notation	Definition
$\mathcal{A}$	Tensor
$\mathbf{A}$	Matrix
$\mathbf{a}$	Vector
a or A	Scalar
$\mathbf{a}(i) = a_i$	i-th entry of a vector $\mathbf{a}$
$\mathbf{A}(i,j) = a_{i,j}$	Element (i,j) of a matrix $\mathbf{A}$
$\mathcal{A}(i_1, i_2, \ldots, i_d) = a_{i_1,i_2,\ldots,i_d}$	Element $(i_1, i_2, \ldots, i_d)$ of a d-dimensional tensor $\mathcal{A}$
$\mathbf{A}^{(n)}$	n-th matrix in a sequence of matrices ($\{\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\}$)
$\mathbf{A}^{-1}$	Matrix inverse
$\mathbf{A}^T$	Matrix transpose
$\|\cdot\|_F$	Frobenius norm
$\|\cdot\|_\star$	Nuclear norm
$\odot$	Hadamard product
$\otimes$	Kronecker product

Table 1-1: Notation conventions.

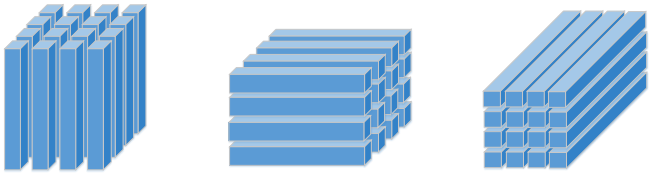
Chapter 2

Tensor-Networks

In this thesis a tensor-networked Kalman filter will be proposed to solve the streaming video completion problem. In order to understand what is meant by a *tensor-networked* Kalman filter, it is important to first understand what tensor-networks are. Therefore, in this chapter an introduction will be given into the subject of tensors (i.e. multi-linear algebra) (Section 2-1) and tensor-networks, specifically the tensor-train (TT) decomposition that will be used in this report (Section 2-2). This introduction is by no means exhaustive, as only those concepts will be introduced that are relevant to the current application. For a more exhaustive overview of tensors and tensor-networks the reader is advised to consult [8].

2-1 Tensors

Tensors are multidimensional arrays. A d-way or d-dimensional tensor has d indices. A matrix, therefore, is a 2-way tensor and a vector a 1-way tensor. The dimensions of tensors are often referred to as *modes*. A *fiber* of a tensor is defined by fixing every index of a tensor but one. For a matrix, a mode-1 fiber is a column and a mode-2 fiber is a row. Figure 2-1 gives an illustration of the fibers of a 3-way tenosor. A *slice* is a two-dimensional section of a tensor (Figure 2-2). In this case all indices except two are fixed. [23]



(a) Mode-1 fibers **(b)** Mode-2 fibers **(c)** Mode-3 fibers

Figure 2-1: Fibers of a 3-way tensor.

(a) Horizontal slices **(b)** Lateral slices **(c)** Frontal slices

Figure 2-2: Slices of a 3-way tensor.

The *matricization* (a.k.a. *unfolding* or *flattening*) of a tensor is the process of reshaping a d-way tensor into a matrix. Two commonly used matricization methods are the mode-n matricization and the mode-$\{n\}$ canonical matricization. In the definitions of these matricization methods, the multi-index notation will be used.

Definition 2.1 (multi-index) [8, p. 274]. A multi-index $i = \overline{i_1 i_2 \dots i_d}$ is an index which takes all possible combinations of values of the indices $i_1, i_2, \dots, i_d$ in a specific order. In this thesis the following (little-endian) ordering convention will be used to be consistent with MATLAB [27]:

$$\overline{i_1 i_2 \dots i_d} = i_1 + (i_2 - 1)I_1 + (i_3 - 1)I_1 I_2 + \cdots + (i_d - 1)I_1 \cdots I_{d-1}. \tag{2-1}$$

Where, $I_1, I_2, \dots, I_d$ represent the size of the modes of the tensor and $i_1, i_2, \dots, i_d$ the corresponding index.

Definition 2.2 (mode-n matricization) [8, p. 275]. The mode-n matricization of a d-way tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ arranges the mode-n fibers to be the columns of a resulting matrix,

$$\mathbf{A}_{(n)} \in \mathbb{R}^{I_n \times (I_1 \cdots I_{n-1} I_{n+1} \cdots I_d)}. \tag{2-2}$$

This matrix has I_n rows and $I_1 I_2 \cdots I_{n-1} I_{n+1} \cdots I_d$ columns with entries,

$$\mathbf{A}_{(n)}(i_n, \overline{i_1 i_2 \dots i_{n-1} i_{n+1} \dots i_d}) = \mathcal{A}(i_1, i_2, \dots, i_d).$$

Another way matrices can be unfolded is by matricizing along n modes, the so-called mode-$\{n\}$ canonical matricization.

Definition 2.3 (mode-$\{n\}$ canonical matricization) [8, p. 275]. In mode-$\{n\}$ canonical matricization a d-way tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is unfolded along n modes, this results in the matrix,

$$\mathbf{A}_{<n>} \in \mathbb{R}^{I_1 I_2 \cdots I_n \times I_{n+1} \cdots I_d}, \tag{2-3}$$

with $I_1 I_2 \cdots I_n$ rows and $I_{n+1} \cdots I_d$ columns, and entries,

$$\mathbf{A}_{<n>}(\overline{i_1 i_2 \dots i_n}, \overline{i_{n+1} i_{n+2} \dots i_d}) = \mathcal{A}(i_1, i_2, \dots, i_d).$$

Using MATLAB syntax the mode-$\{n\}$ canonical matricization can easily be calculated with a single call to the `reshape` function: $\mathbf{A}_{<n>} = \texttt{reshape}\left(\mathcal{A}, \prod_{k=1}^{n} I_k, \prod_{k=n+1}^{d} I_k\right)$.

Tensors can be multiplied with matrices. One way to do this is by multiplying the tensor in mode n: the *n-mode product*.

Definition 2.4 (n-mode product) [23, p. 460–461]. For a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ and a matrix $\mathbf{B} \in \mathbb{R}^{J \times I_n}$ the n-mode product is defined as,

$$\begin{aligned} &\mathcal{C} = \mathcal{A} \times_n \mathbf{B} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_d} \\ \text{where} \quad &\mathcal{C}(i_1, \ldots, i_{n-1}, j, i_{n+1}, \ldots, i_d) = \sum_{i_n=1}^{I_n} \mathcal{A}(i_1, i_2, \ldots, i_d)\, \mathbf{B}(j, i_n) \end{aligned} \tag{2-4}$$

This can also be expressed in terms of unfolded tensors: $\mathbf{C}_{(n)} = \mathbf{B}\,\mathbf{A}_{(n)}$.

The norm of a tensor is similarly defined as the Frobenius norm of matrices and vectors.

Definition 2.5 (Tensor norm) [23, p. 457]. The Frobenius norm of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is defined as the square root of the sum of squares of the elements of the tensor,

$$\|\mathcal{A}\|_F = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_d=1}^{I_d} a_{i_1, i_2, \ldots, i_d}^2}. \tag{2-5}$$

As for matrices the Kronecker product of two tensors can be computed. A distinction can be made between the right and left Kronecker product. The right Kronecker product of two matrices $\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{P \times Q}$ is defined as,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & \cdots & a_{1,N}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{M,1}\mathbf{B} & \cdots & a_{M,N}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{MP \times NQ}.$$

And the left Kronecker product as,

$$\mathbf{A} \otimes_L \mathbf{B} = \begin{bmatrix} \mathbf{A}\,b_{1,1} & \cdots & \mathbf{A}\,b_{1,P} \\ \vdots & \ddots & \vdots \\ \mathbf{A}\,b_{Q,1} & \cdots & \mathbf{A}\,b_{Q,P} \end{bmatrix} \in \mathbb{R}^{MP \times NQ}.$$

It is easy to observe that $\mathbf{A} \otimes \mathbf{B} = \mathbf{B} \otimes_L \mathbf{A}$ [8]. Therefore, in this report only the right Kronecker product will be used as this is the one most generally used.

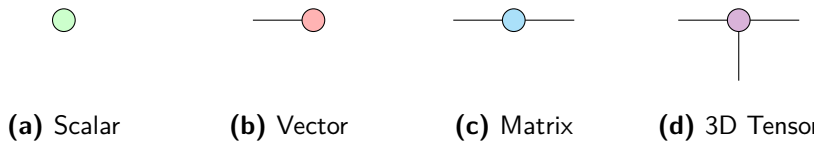
Definition 2.6 (Kronecker product of tensors) [8, p. 277–279] The (right) Kronecker of two tensors $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ and $\mathcal{B} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_d}$ results in a tensor $\mathcal{C} \in \mathbb{R}^{I_1 J_1 \times I_2 J_2 \times \cdots \times I_d J_d}$. The elements of this tensor are equal to,

$$\mathcal{C}(\overline{j_1 i_1}, \ldots, \overline{j_d i_d}) = \mathcal{A}(i_1, \ldots, i_d)\, \mathcal{B}(j_1, \ldots, j_d).$$

2-1-1 Tensor Diagrams

Notation of tensors can quickly become very cumbersome due to the large amount of indices. That is why Penrose [31] developed a graphical way to illustrate tensors and tensor products. These graphical representations of tensors are called *tensor diagrams*.

In tensor diagrams each tensor is illustrated by a node. The nodes can take on a variety of shapes. In this report circles will be used. Each node can have attached to it a number of *branches*. Each branch represents a specific mode of the tensor. The size of the mode can be noted next to the branch. In Figure 2-3 some examples are given of tensors represented by tensor diagrams.

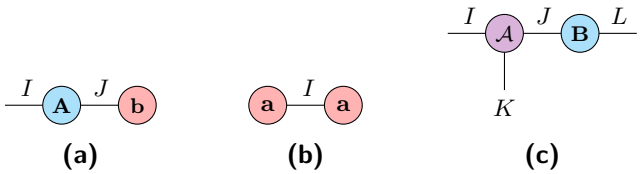


(a) Scalar **(b)** Vector **(c)** Matrix **(d)** 3D Tensor

Figure 2-3: Tensor diagrams of tensors of different size.

Tensors can also be interconnected by linking two (or more) branches. Each interconnection represents a contraction over that mode, i.e. a summation of products over that mode [8]. This way tensor products can be represented by tensor diagrams. Figure 2-4 shows examples of three simple products being represented by tensor diagrams, the matrix-vector product, the (vector) dot product and the n-mode product.

In Figure 2-4a a matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$ is multiplied with a vector $\mathbf{b} \in \mathbb{R}^{J}$. The link between the nodes represents the contraction over the mode of size J, which is equal to the matrix-vector product: $\mathbf{A}\,\mathbf{b} = \sum_{j=1}^{J} \mathbf{A}(:, j)\mathbf{b}(j)$. Similarly, the dot product of two vectors can be expressed as summation of the products of the elements of the vectors: $\langle \mathbf{a}, \mathbf{a} \rangle = \sum_{i=1}^{I} \mathbf{a}(i)\mathbf{a}(i)$. This is denoted in the tensor diagram by a connection along the single mode of the vectors (Figure 2-4b). Lastly, Figure 2-4c shows an example of the 2-mode product of 3-way tensor $\mathcal{A} \in \mathbb{R}^{I \times J \times K}$ and a matrix $\mathbf{B} \in \mathbb{R}^{L \times J}$. In this case the tensor and the matrix are contracted over the mode of size J (the second mode of the tensor): $\mathcal{A} \times_2 \mathbf{B} = \sum_{j=1}^{J} \mathcal{A}(:, j, :)\,\mathbf{B}(l, j)$.

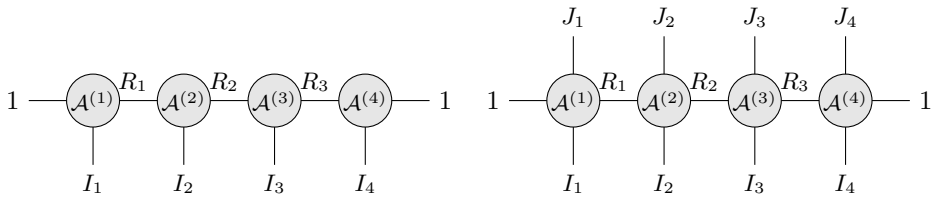


(a) **(b)** **(c)**

Figure 2-4: Tensor diagrams of tensor products. **(a)** matrix-vector product, **(b)** vector dot product, **(c)** n-mode product.

2-2 Tensor-Train Decomposition

The number of entries in a tensor grows exponentially in d (for a d-way tensor), therefore high dimensional problems cannot be handled efficiently by standard numerical methods, as operations and memory usage grow exponentially as well (*curse of dimensionality*) [29].



(a) TT **(b)** TTm

Figure 2-5: TT **(a)** and TTm **(b)** decomposition of a tensor $\mathcal{A}$ with cores $\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(4)}$ and ranks $R_1, \ldots, R_3$. The size of the modes of the TT(m)-cores are denoted next to the branches of the tensor diagram.

Thus, an efficient representation of a tensor is needed to work with these problems. There are three commonly used tensor decompositions: the CANDECOMP/PARAFAC (CP) [4, 12], Tucker [41] and TT decomposition [29]. In this report the tensor-train (TT) decomposition will be used, as this decomposition can efficiently store low-rank data while still being able to perform matrix and vector computations [28]. This section will give an overview of this type of tensor-network.

2-2-1 Tensor-Train Definition

In the TT decomposition a d-way tensor is decomposed into d auxiliary three-dimensional tensors (*cores*).

Definition 2.7 (Tensor-Train) [29, p. 2296]. A d-dimensional tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is decomposed in the tensor-train format if the following holds

$$\begin{aligned} \mathcal{A}(i_1, i_2, \ldots, i_d) &= \langle\!\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\!\rangle \\ &= \sum_{r_0, \ldots, r_d} \mathcal{A}^{(1)}(r_0, i_1, r_1) \mathcal{A}^{(2)}(r_1, i_2, r_2) \cdots \mathcal{A}^{(d)}(r_{d-1}, i_d, r_d), \end{aligned} \tag{2-6}$$

where $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ (for $n = 1, 2, \ldots, d$) are the so-called *TT-cores*. The parameters R_n are called the *TT-ranks* and by definition $R_0 = R_d = 1$.

Figure 2-5a shows a graphical representation of a TT with four cores using tensor diagrams. Each link represents one of the indices of the cores. The linkage of two cores represents a contraction over this index.

Definition 2.8 (TT-rank) [29, p. 2297]. The TT-rank of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with ranks $R_1, R_2 \ldots, R_{d-1}$ is defined as:

$$\operatorname{rank}_{\mathrm{TT}}(\mathcal{A}) = [R_1, R_2, \ldots, R_{d-1}]\,. \tag{2-7}$$

It has been determined that the TT-ranks are equivalent to the ranks of the canonical mode-$\{n\}$ unfolded matrices, so $R_n = \operatorname{rank}(\mathbf{A}_{<n>})$ [8].

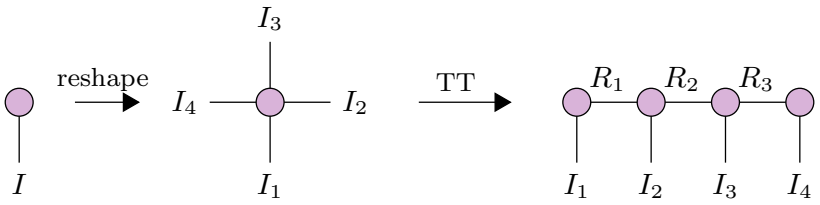


Figure 2-6: Tensor-train representation of a vector. A vector of length I is reshaped into a tensor of size $I_1 \times I_2 \times I_3 \times I_4$. This tensor is then converted into a TT.

2-2-2 Vectors and Matrices as TT(m)'s

Vectors and matrices can be represented by TT's. For a vector this can simply be done by treating a vector of length $I = I_1 I_2 \cdots I_d$ as a d-dimensional tensor with modes of size I_k, where $k = 1, 2, \ldots, d$, and representing this tensor in the TT-format (Figure 2-6). A large matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$ can be represented by a $2d$-dimensional tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \cdots \times I_d \times J_d}$, where $I = I_1 I_2 \cdots I_d$ and $J = J_1 J_2 \cdots J_d$. This tensor can then be decomposed in the tensor-train matrix (TTm) format (Definition 2.9) [8], which is similar to the TT decomposition, but instead of 3-dimensional cores, the TTm decomposition has 4-dimensional cores (Figure 2-5b).

Definition 2.9 (Tensor-Train Matrix) [29, p. 2312]
A $(2d)$-dimensional tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \cdots \times I_d \times J_d}$ is decomposed in the tensor-train matrix format if the following holds,

$$\begin{aligned} \mathcal{A}(i_1, \ldots, i_d, j_1, \ldots, j_d) &= \langle\!\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\!\rangle \\ &= \sum_{r_0, \ldots, r_d} \mathcal{A}^{(1)}(r_0, i_1, j_1, r_1) \mathcal{A}^{(2)}(r_1, i_2, j_2, r_2) \cdots \mathcal{A}^{(d)}(r_{d-1}, i_d, j_d, r_d), \end{aligned} \tag{2-8}$$

where $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ (for $n = 1, 2, \ldots, d$) are the so-called *TTm-cores*. The parameters R_n are called the *TTm-ranks* and by definition $R_0 = R_d = 1$.

2-2-3 Quantized Tensor-Train

The process of transforming lower-dimensional data like vectors and matrices into higher-dimensional tensors, is called *tensorization*. When the size of the modes of the resulting tensor are small (typically of size 2, 3 or 4) this is called *quantization* [19]. Using quantization high compression ratios can be obtained with a low-rank TT approximation [28]. A tensor-train constructed of a quantized tensor is called a quantized tensor-train (QTT). To showcase the high compression ratio that can be achieved using QTT's, an example is given below.

Example Given a vector $x \in \mathbb{R}^I$, where $I = 2^d$. This vector can be quantized by reshaping it into the d-way tensor $\mathcal{X} \in \mathbb{R}^{2 \times 2 \times \cdots \times 2}$. The TT-decomposition of this tensor has cores of size $(R_{n-1} \times 2 \times R_n)$. The ranks of the cores of such a QTT are typically small and almost independent of data size [8].
Suppose $R = \max_n(R_n) = 1$ (the ideal case), this would lead to a reduction in storage

requirements from $\mathcal{O}(I)$ to $\mathcal{O}(\log_2(I))$. For $I = 1000$ this leads to a compression ratio of 100, so the QTT is 100 times smaller than the original vector.

The concept described above can be generalized for arbitrary low-dimensional tensors. A tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ has a storage complexity of $\mathcal{O}\left(I_{\max}^d\right)$, where $I_{\max} = \max\{I_1, I_2, \ldots, I_d\}$. Suppose it can be quantized using the quantization parameter q, such that $I_n = q^k$. Storing this quantized version of the tensor in TT-format leads to a logarithmic (sub-linear) reduction in storage complexity: from $\mathcal{O}\left(I_{\max}^d\right)$ to $\mathcal{O}\left(dR^2 \log_q(I_{\max})\right)$, $R = \max\{R_1, R_2, \ldots, R_{d-1}\}$ [8]. This type of compression is generally referred to as *super-compression* [19]. In Table 2-1 the storage complexities of the TT and QTT decomposition are compared to the storage complexity of a tensor in full format.

Format	Storage Complexity
Full	$\mathcal{O}\left(I_{\max}^d\right)$
TT	$\mathcal{O}\left(dR^2 I_{\max}\right)$
QTT	$\mathcal{O}\left(dR^2 \log_q(I_{\max})\right)$

Table 2-1: Storage complexities of tensor decompositions for an arbitrary tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ that can be quantized by a parameter q. $I_{\max} = \max\{I_1, I_2, \ldots, I_d\}$ and $R = \max\{R_1, R_2, \ldots, R_{d-1}\}$.

2-2-4 Transforming a Tensor into TT-format

Transforming tensors into their TT decomposition can be done using the TT-SVD algorithm [29]. This algorithm uses a truncated singular value decomposition (SVD) of the unfolding matrices. The values at which these SVD's are truncated are the TT-ranks. A graphical illustration of this algorithm is given by Figure 2-7.

Determining at which rank the SVD should be truncated can be done by one of two ways. One possibility is to simply set a maximum rank constraint for each core, and truncate the SVD at this value. Another way makes use of the property that the Frobenius norm of a matrix is equal to the square root of the sum of the squared singular values of that matrix. Using this property it can be determined that the total (absolute) approximation error is less than or equal to the following sum of the 'discarded' singular values:

$$\|\mathcal{A} - \mathcal{A}_{TT}\|_F^2 \leq \sum_{n=1}^{d-1} \sum_{i=R_{n+1}}^{I_n} \sigma_i^2(\mathbf{A}_{<n>}). \tag{2-9}$$

Where $\sigma_i(\mathbf{A}_{<n>})$ denotes the i-th largest singular value of the unfolding matrix $\mathbf{A}_{<n>}$.

Thus, it is possible to prescribe an accuracy ε such that $\|\mathcal{A} - \mathcal{A}_{TT}\|_F \leq \varepsilon \|\mathcal{A}\|_F$, by truncating the ranks of each core n at R_n such that,

$$\sqrt{\sum_{i=R_{n+1}}^{I_n} \sigma_i^2(\mathbf{A}_{<n>})} \leq \frac{\varepsilon}{\sqrt{d-1}}.$$

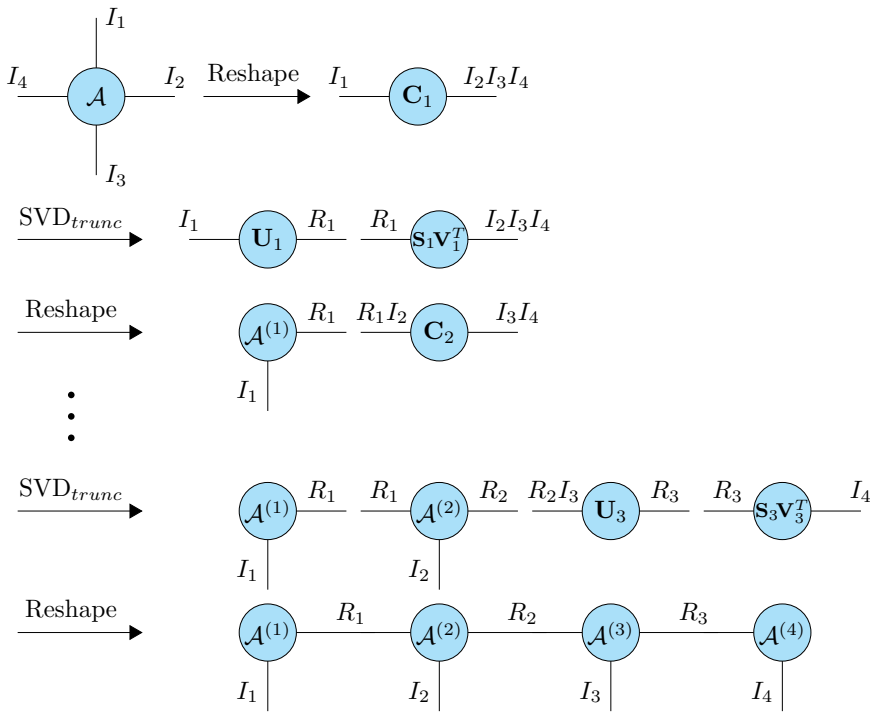


Figure 2-7: Graphical illustration of the TT-SVD algorithm (inspired by Figure 4.13 of [8]).

Both of these truncation methods have their advantages and disadvantages. When setting a maximum rank constraint, one has full control over the size of the resulting tensor-train. However, the downside is that the resulting TT can be very inaccurate and there is no control over the approximation error. Using a prescribed accuracy on the other hand does ensure the relative error stay below a certain value. However, the resulting TT could be of such a large size that performing computations with it becomes too expensive. Generally a trade-off needs to be made between the accuracy of the estimate and the computational performance. Algorithm 1 shows the full TT-SVD algorithm.

Tensor-Train Matrix While this section has shown how a tensor can be transformed into TT-format. It should be noted that transforming a TT into a TTm can very easily be done by a reshape of the cores. Therefore, the methods described above are also valid for TTm's.

To illustrate this let's take a matrix $\mathbf{X} \in \mathbb{R}^{I \times J}$ that can be transformed into a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times J_1 \times \cdots \times I_d \times J_d}$. This $2d$-dimensional tensor can be reshaped into a d-dimensional tensor $\mathcal{Y} \in \mathbb{R}^{I_1 J_1 \times \cdots \times I_d J_d}$. Using the TT-SVD algorithm this tensor can be decomposed as a tensor-train $\mathcal{Y} = \langle\!\langle \mathcal{Y}^{(1)}, \ldots, \mathcal{Y}^{(d)} \rangle\!\rangle$, with cores $\mathcal{Y}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n J_n \times R_n}$. Obtaining the tensor-train matrix representation of $\mathcal{X}$ is now a simple matter of reshaping the cores: $\mathcal{X}^{(n)} = \texttt{reshape}(\mathcal{Y}^{(n)}, [R_{n-1}, I_n, J_n, R_n])$.

Algorithm 1: TT-SVD[1] (p. 2301 of [29])

Data: $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$, desired TT-ranks $[R_1, \ldots, R_{d-1}]$ or accuracy ε.
Result: Tensor-Train $\mathcal{B} = \langle\!\langle \mathcal{B}^{(1)}, \mathcal{B}^{(2)}, \ldots, \mathcal{B}^{(d)} \rangle\!\rangle$.
Compute truncation parameter: $\delta = \frac{\varepsilon}{\sqrt{d-1}} \|\mathcal{A}\|_F$;
Temporary variable: $\mathbf{C} = \mathcal{A}$;
Set $R_0 = 1$;
for $n = 1, 2, \ldots, d-1$ **do**
  $\mathbf{C} = \texttt{reshape}(\mathbf{C}, [R_{n-1} I_n, \frac{\texttt{numel}(\mathcal{C})}{R_{n-1} I_n}])$
  Truncated SVD: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{SVD}_{trunc}(\mathbf{C})$, where $\mathbf{C} = \mathbf{U}\,\mathbf{S}\,\mathbf{V}^T + \mathbf{E}$, $\|\mathbf{E}\|_F \leq \delta$.
  New rank: $R_n = \texttt{size}(\mathbf{U}, 2)$
  $\mathcal{B}^{(n)} = \texttt{reshape}(\mathbf{U}, [R_{n-1}, I_n, R_n])$
  $\mathbf{C} = \mathbf{S}\,\mathbf{V}^T$
end
$\mathcal{B}^{(d)} = \mathbf{C}$

2-3 Calculating with Tensor-Trains

This section will introduce some algorithms that make it possible to perform mathematical operations using tensor-trains. In Section 2-3-1 basic operations like addition and scalar multiplication are introduced. Matrix and vector products are included in Section 2-3-2. Finally, sections 2-3-3 and 2-3-4 introduce the orthogonalization and rounding algorithms that help keep the calculations tractable.

2-3-1 Basic Operations in TT-format

Addition The addition of two tensors in TT-format,

$$\mathcal{C} = \mathcal{A} + \mathcal{B} = \langle\!\langle \mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(d)} \rangle\!\rangle + \langle\!\langle \mathcal{B}^{(1)}, \ldots, \mathcal{B}^{(d)} \rangle\!\rangle = \langle\!\langle \mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(d)} \rangle\!\rangle$$

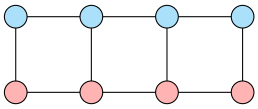
can be done by merging the cores in the following way[2]

$$\begin{aligned} \mathcal{C}^{(n)}(i_n) &= \begin{bmatrix} \mathcal{A}^{(n)}(i_n) & 0 \\ 0 & \mathcal{B}^{(n)}(i_n) \end{bmatrix}, \quad n = 2, 3, \ldots, d-1 \\ \mathcal{C}^{(1)}(i_n) &= \begin{bmatrix} \mathcal{A}^{(1)}(i_n) & \mathcal{B}^{(1)}(i_n) \end{bmatrix}, \quad \mathcal{C}^{(d)}(i_n) = \begin{bmatrix} \mathcal{A}^{(d)}(i_n) \\ \mathcal{B}^{(d)}(i_n) \end{bmatrix}. \end{aligned} \tag{2-10}$$

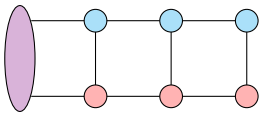
The resulting TT has rank $\mathbf{r}_\mathcal{C} = \mathbf{r}_\mathcal{A} + \mathbf{r}_\mathcal{B}$ (where $\mathbf{r}_\mathcal{A}$ represents the ranks of tensor $\mathcal{A}$: $\mathbf{r}_\mathcal{A} = [R_1, R_2 \ldots, R_{d-1}]$) [29].

[1] code: https://gitlab.com/seline/thesis/-/blob/a907866468cef6f74655d84bb3255b06f7ff5958/Matlab_Code/TT_Functions/TT_SVD_eps.m, https://gitlab.com/seline/thesis/-/blob/a907866468cef6f74655d84bb3255b06f7ff5958/Matlab_Code/TT_Functions/TT_SVD_rank.m

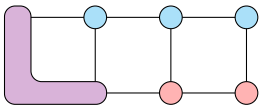
[2] Indices are omitted for sake of legibility: $\mathcal{A}^{(n)}(i_n) = \mathcal{A}^{(n)}(:, i_n, :)$ etc.



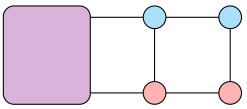
(a) Step 1: connecting the cores.



(b) Step 2: contracting the mode-1 cores.



(c) Step 3: contraction of mode-1 and mode-2 cores.



(d) Step 4: contracting the mode-2 cores.

Figure 2-8: Illustration of first steps of the TT dot product using tensor diagrams.

Multiplication by a scalar The multiplication of a TT with a scalar can be accomplished by simply multiplying one of the cores with this value. If the TT is orthogonalized in mode n (see Section 2-3-3) the orthogonality of the TT can be preserved by multiplying the scalar with the n-th core.

Transpose The transpose of a matrix in TTm-format can simply be calculated by permuting the indices. A matrix $\mathbf{A}$ represented by a tensor-train matrix $\mathcal{A}$ with cores $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ is thus transformed into its transpose $\mathbf{A}^T$ by a TTm $\mathcal{A}^T$ with cores $(\mathcal{A}^T)^{(n)} \in \mathbb{R}^{R_{n-1} \times J_n \times I_n \times R_n}$.

Extracting Values Values of a tensor in TT-format can easily be extracted while in TT-format. Extracting the value $\mathcal{A}(i_1, i_2, \ldots, i_d)$ can be done by selecting the values in each of the cores $\mathcal{A}^{(n)}(:, i_n, :)$ (for $n = 1, 2, \ldots, d$), and then contracting what is left of the cores.

If a vector is decomposed in QTT-format, the same principle applies, only now the linear (vector) index should be converted in a multidimensional index. This can be achieved in MATLAB using the `ind2sub` function. For matrices in TTm-format it is also possible to extract a row or a column, by selecting only one of the index values in each of the cores. To extract a column, for instance, that has an index $j \rightarrow (j_1, j_2, \ldots, j_d)$, it is simply a matter of setting the cores of a new tensor-train as $\mathcal{B}^{(n)} = \mathcal{A}^{(n)}(:, :, j_n, :)$ (for $n = 1, 2, \ldots, d$).

2-3-2 Matrix and vector products

Vector and matrix products of large vectors and matrices that are transformed into d-way TT's or TTm's, can be efficiently computed using matricizations of the cores. All algorithms corresponding to the products referenced below can be found in Appendix A-1. The computational complexity of the algorithms is denoted in Table 2-2.

Dot product

The dot (or *inner*) product of two vectors in TT-format ($c = \langle \mathcal{A}, \mathcal{B} \rangle$) can be calculated using a contraction of the cores. The easiest way to explain how this can be done is by using tensor diagrams. In Figure 2-8 the first steps of the dot product calculation in TT-format is

Product	Complexity	Cores
Matrix-Vector	$\mathcal{O}\left(d\,IJ(PR)^2\right)$	$\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1}\times I_n\times J_n\times R_n}$, $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1}\times J_n\times P_n}$
Matrix-Matrix	$\mathcal{O}\left(d\,IJM(PR)^2\right)$	$\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1}\times I_n\times J_n\times R_n}$, $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1}\times J_n\times M_n\times P_n}$
Outer product	$\mathcal{O}\left(d\,IM(PR)^2\right)$	$\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1}\times I_n\times R_n}$, $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1}\times M_n\times P_n}$
Dot Product	$\mathcal{O}\left(d\,I(R^2P+RP^2)\right)$	$\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1}\times I_n\times R_n}$, $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1}\times M_n\times P_n}$

Table 2-2: Complexity of matrix and vector products in TT(m)-format [8]. The size of the cores of the corresponding TT(m)'s is denoted in the third column. Each TT(m) has d cores, where $n = 1, 2, \ldots, d$; and $I = \max(I_n)$, $J = \max(J_n)$, $R = \max(R_n)$ etc.

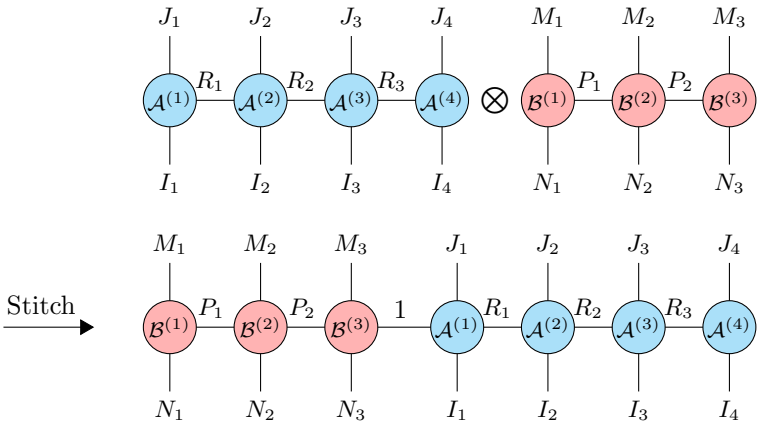


Figure 2-9: Kronecker product tensor diagram.

illustrated. As one can see, the dot product is calculated using successive contractions of the cores. If all the cores of the tensor-trains are contracted this leads to a scalar, since the ranks at the edges of the tensor-trains, R_0 and R_d, are equal to one.

The order in which these contractions are performed decides the complexity of the operation. In this report the contractions are performed from left to right. This leads to the complexity noted in Table 2-2.

Kronecker product

In TT(m)-format the Kronecker product of two variables can be calculated by 'stitching' the ends of the TT(m)'s with a branch of rank 1 [8]. In Figure 2-9 the Kronecker product of two tensor-train matrices is illustrated. Due to MATLAB's little-endian ordering (2-1) the TTm's need to be in reversed order when connecting to perform the (right) Kronecker product. Because the Kronecker product is simply a case of 'connecting' the two TT(m)'s, no products are computed. Thus, there is no (theoretical) computational complexity.

Matrix-vector and matrix-matrix product

The matrix-vector and matrix-matrix products of vectors and matrices in TT and TTm format, respectively, can be calculated using contractions of the cores. This method is referred

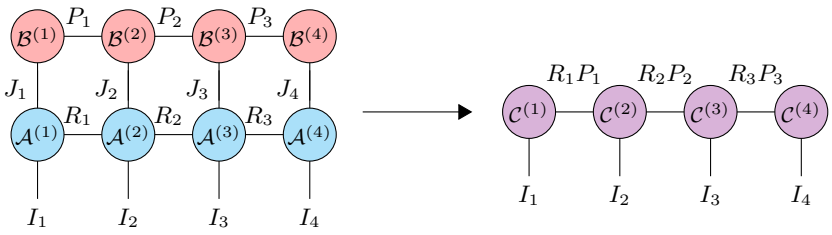


Figure 2-10: Matrix-vector product tensor diagram.

to as the 'zip-up' method, and was first introduced in [39]. The name 'zip-up' referes to the way each core is (sequentially) contracted.

Figure 2-10 shows a tensor diagram of the matrix-vector product. Here, $\mathcal{B} = \langle\langle \mathcal{B}^{(1)}, \dots, \mathcal{B}^{(4)} \rangle\rangle$ is a 4-dimensional TT with cores of size $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1} \times J_n \times P_n}$, and $\mathcal{A}$ is a TTm with cores of size $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$. The resulting product is a tensor-train, $\mathcal{C}$, with cores of size $\mathcal{C}^{(n)} \in \mathbb{R}^{R_{n-1}P_{n-1} \times I_n \times R_n P_n}$.

In Figure 2-11 the same is shown for the matrix-matrix product. Two TTm's $\mathcal{A}$ and $\mathcal{B}$ representing matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{J \times M}$ are multiplied to form a TTm $\mathcal{C}$ that represents the product of $\mathbf{A}$ and $\mathbf{B}$: $\mathbf{C} \in \mathbb{R}^{I \times M}$. The cores of the original TTm's, $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ and $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1} \times J_n \times M_n \times P_n}$, are contracted to form the cores of $\mathcal{C}$: $\mathcal{C}^{(n)} \in \mathbb{R}^{R_{n-1}P_{n-1} \times I_n \times M_n \times R_n P_n}$.

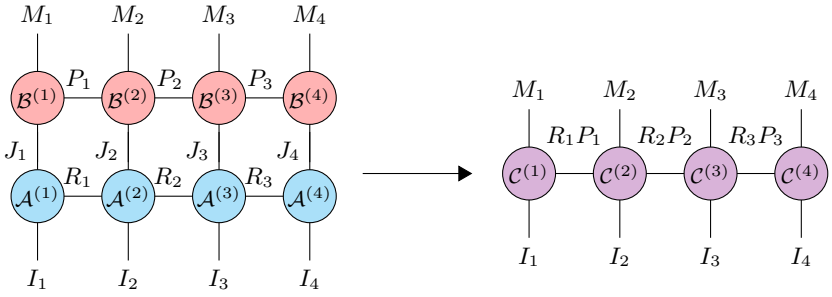


Figure 2-11: Matrix-matrix product tensor diagram.

Vector outer product

Like the dot product also the vector outer product ($\mathbf{C} = \mathbf{a} \circ \mathbf{b}$) can be calculated in TT-format. Calculation of this product in TT-format is similar to the matrix-matrix product (Figure 2-11), only now the dimensions J_n are all equal to one. This way two vectors in TT-format, $\mathcal{A}$ with cores $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ and $\mathcal{B}$ with cores $\mathcal{B}^{(n)} \in \mathbb{R}^{P_{n-1} \times M_n \times P_n}$ can be multiplied to form a d-dimensional TTm $\mathcal{C}$ with cores $\mathcal{C}^{(n)} \in \mathbb{R}^{R_{n-1}P_{n-1} \times I_n \times M_n \times R_n P_n}$.

2-3-3 Orthogonalization

The orthogonalization of tensor-trains is an essential part of the rounding algorithm that will be discussed in the next section. Therefore, this section will introduce the concept of

n-orthogonality of tensor-trains. If a tensor-train is n-orthogonal, the unfolded cores of the tensor-train, except for the n-th core, are orthogonal matrices. Definition 2.10 gives the formal definition of this n-orthogonality.

Definition 2.10 (n-orthogonality) [8, p. 399–400] A d-dimensional tensor in TT-format $\mathcal{A} = \langle\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\rangle$ is called n-orthogonal if the following holds for the unfolded cores,

$$(\mathbf{A}^{(m)}_{<2>})^T \, \mathbf{A}^{(m)}_{<2>} = \mathbf{I}, \quad m = 1, 2, \ldots, n-1 \tag{2-11}$$

$$\mathbf{A}^{(m)}_{<1>} (\mathbf{A}^{(m)}_{<1>})^T = \mathbf{I}, \quad m = n+1, n+2, \ldots, d. \tag{2-12}$$

Where $\mathbf{A}^{(m)}_{<2>}$ is the mode-$\{2\}$ canonical matricization of core m, which is sometimes referred to as the left unfolding of the core. And $\mathbf{A}^{(m)}_{<1>}$ is the mode-$\{1\}$ canonical matricization of core m, the right unfolding of the core. When $n = d$ the TT is called left-orthogonal, and when $n = 1$ the TT is called right-orthogonal.

A TT can be orthogonalized using recursive QR decompositions of the unfolded cores (Algorithm 2). In this algorithm the QR decomposition of the unfolded cores is computed. The $\mathbf{Q}$-matrix of this decomposition is then set as the current core; whereas the $\mathbf{R}$-matrix is 'absorbed' by the next core. This continues until the n-th core is reached. Since $\mathbf{Q}$ is an orthogonal matrix, this results in the orthogonalization of the unfolded cores (except the n-th core). Because of this orthogonality, the Frobenius norm of a tensor in n-orthogonal TT-format is equal to the norm of the n-th core: $\|\mathcal{A}\|_F = \left\|\mathcal{A}^{(n)}\right\|_F$.

Algorithm 2: Orthogonalization[3] [8, p. 399–400]

Data: $\mathcal{A} = \langle\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with ranks $[R_0, R_1, \ldots, R_d]$.
Result: n-orthogonal $\mathcal{A}$.
{Left-to-right orthogonalization}
for $k = 1, 2, \ldots, n-1$ **do**
  $\mathbf{A}_L = \texttt{reshape}(\mathcal{A}^{(k)}, R_{k-1} I_k, R_k)$
  $[\mathbf{Q}, \mathbf{R}] = \texttt{qr}(\mathbf{A}_L)$
  $R_k = \texttt{size}(\mathbf{Q}, 2)$
  $\mathcal{A}^{(k)} = \texttt{reshape}(\mathbf{Q}, R_{k-1}, I_k, R_k)$
  $\mathcal{A}^{(k+1)} = \mathcal{A}^{(k+1)} \times_1 \mathbf{R}$
end
{Right-to-left orthogonalization}
for $k = d, d-1, \ldots, n+1$ **do**
  $\mathbf{A}_R = \texttt{reshape}(\mathcal{A}^{(k)}, R_{k-1}, I_k R_k)$
  $[\mathbf{Q}, \mathbf{R}] = \texttt{qr}(\mathbf{A}_R^T)$
  $R_{k-1} = \texttt{size}(\mathbf{Q}, 2)$
  $\mathcal{A}^{(k)} = \texttt{reshape}(\mathbf{Q}^T, R_{k-1}, I_k, R_k)$
  $\mathcal{A}^{(k-1)} = \mathcal{A}^{(k+1)} \times_3 \mathbf{R}^{\mathbf{T}}$
end

[3] code: https://gitlab.com/seline/thesis/-/blob/dd9fdd34becb438d8c49366eafc1f9b2cd74003b/Matlab_Code/@TensorTrain/orthogonalize.m

2-3-4 Rounding

Performing operations on TT's can lead to the increase of the ranks of the TT's. The matrix-vector product, for example, leads to a multiplication of the TT(m)-ranks. To prevent that the TT's become so large in size that it is no longer possible to perform efficient calculations, the ranks need to be truncated to smaller values. The process by which this can be done is called *rounding* (or *TT-rounding*) [29].

Rounding consists of two steps, an orthogonalization and a compression step. First, the tensor-train is orthogonalized from right-to-left to obtain the left-orthogonal form of the TT. Once the tensor-train is left-orthogonal, the unfolded cores are compressed using a truncated SVD. Algorithm 3 shows the full TT-rounding algorithm. This rounding process is mathematically similar to the TT-SVD algorithm, but because the tensor is already in TT-format it is more efficient. Only SVD's of the relatively small unfolded cores need to be computed. Because of this the complexity of the rounding process is only $\mathcal{O}\left(dIR^3\right)$ ($I = \max\{I_1, \ldots, I_d\}$ and $R = \max\{R_1, \ldots, R_{d-1}\}$) [8].

Algorithm 3: TT-rounding[4][8, 29]

Data: Tensor-Train $\mathcal{A} = \langle\!\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with cores $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$, tolerance ε and/or maximum rank R_{max}.

Result: Tensor-Train $\mathcal{A}$ with reduced TT-ranks. Ranks are bounded by R_{max} and/or relative error is bounded by ε.

{Left orthogonalization}

$\mathcal{A} = \texttt{orthogonalize}(\mathcal{A}, 1)$ `// Algorithm 2`

{Compression of orthogonalized representation}

for $n = 1, 2, \ldots, d-1$ **do**

  $\mathbf{A}_n = \texttt{reshape}(\mathcal{A}^{(n)}, [R_{n-1} I_n, R_n])$

  Compute truncated SVD: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{SVD}_{trunc}(\mathbf{A}_n)$

  Set new rank: $R_n = \texttt{size}(\mathbf{U}, 2)$

  Replace cores: $\mathcal{A}^{(n)} = \texttt{reshape}(\mathbf{U}, R_n, I_n, R_{n+1})$, $\mathcal{A}^{(n+1)} = \mathcal{A}^{(n+1)} \times_1 (\mathbf{V}\,\mathbf{S})^T$

end

2-4 Summary

In this chapter all the necessary background information on tensors and tensor-trains for the tensor-networked Kalman filter has been introduced. This chapter started with an introduction of some important concepts in tensor algebra (Section 2-1). After this the tensor-train decomposition was introduced as a way to efficiently store high-dimensional tensors as well as low-dimensional data like vectors and matrices (Section 2-2); and, perhaps most importantly, how these tensor-trains can be used perform calculations with large vectors and matrices (Section 2-3).

The next chapter will provide background information on the Kalman filter, the other important part of the tensor-networked Kalman filter method.

[4] code: https://gitlab.com/seline/thesis/-/blob/dd9fdd34becb438d8c49366eafc1f9b2cd74003b/Matlab_Code/@TensorTrain/rounding.m

Chapter 3

Kalman Filter

This chapter provides some background information on Kalman filtering. Section 3-1 start with an introduction in state-space modeling. In Section 3-2 a general introduction is given into the Kalman filter. This section gives the general form of the Kalman filter and its properties. Section 3-3 introduces a version of the Kalman filter where the update step of the Kalman filter is divided into *partitions*, the partitioned update Kalman filter (PUKF).

3-1 State-Space Model

A Kalman filter is a model-based estimator. It uses a state-space model of a system to perform the estimation of the corresponding state-vector. Therefore, this section will introduce this type of modeling and provide the notation conventions that will be used throughout this thesis.

State-space models are often used in control engineering. It provides a mathematical representation of a physical system. The input, output and state variables are related by set of first order differential (for continuous time systems) or difference (for discrete time systems) equations. Because videos are inherently discrete, only discrete-time systems will be discussed.

Equation (3-1) gives an example of a noise-free discrete-time state-space system. Here, $\mathbf{x}[k] \in \mathbb{R}^I$ represents the state-vector at time-step k, $\mathbf{u}[k] \in \mathbb{R}^Q$ the input at time-step k and $\mathbf{y}[k] \in \mathbb{R}^J$ the output at time-step k. $\mathbf{A}[k]$ is the state-transition matrix, $\mathbf{B}[k]$ the control matrix, $\mathbf{C}[k]$ the measurement matrix and $\mathbf{D}[k]$ the direct feedthrough matrix. Time indices are indicated by square brackets: $[\cdot]$.

$$\begin{aligned} \mathbf{x}[k+1] &= \mathbf{A}[k]\mathbf{x}[k] + \mathbf{B}[k]\mathbf{u}[k] \\ \mathbf{y}[k] &= \mathbf{C}[k]\mathbf{x}[k] + \mathbf{D}[k]\mathbf{u}[k] \end{aligned} \tag{3-1}$$

The state-transition matrix includes the system dynamics, it represents how the state-vector changes over time. The control matrix indicates how an input signal effects the states of the system. The relation between the measurements and the states is provided by the measurement matrix. This matrix can be very sparse, as often a measurement is of a single state variable. The direct feedthrough matrix describes the *direct* relation between the input $\mathbf{u}[k]$ and the output $\mathbf{y}[k]$ (without time-delay). This matrix is often equal to the zero-matrix, as the input is generally applied to the system and rarely directly influences the output. If all system matrices are independent of time, the system is said to be linear time-invariant (LTI).

3-2 Kalman filter

The Kalman filter was first introduced by Rudolf E. Kàlmàn in 1960 [18]. Since then it has been applied to many different engineering problems. The most famous of which was perhaps its application in estimating the trajectory of a manned spacecraft for the Apollo program [10]. Other applications include control of dynamic systems [21], image processing [20] and finance [33].

3-2-1 Process to be Estimated

The Kalman filter is a member of a class of filter known as *observers*. Observers aim to estimate the state-vector $\mathbf{x}[k]$ of controlled process that can be represented by a state-space model. This estimation is performed using information of the previous state-vector and measurements of the system. This makes the Kalman filter a recursive estimator. Which enables it to be implemented in real-time.

One of the main differences that distinguishes the Kalman filter from other observers is that it takes noise disturbances into account. This makes it more suitable for real-world applications, as in practice systems are (almost) always disturbed by noise [5]. The (discrete-time) Kalman filter assumes the following underlying dynamical system [43],

$$\mathbf{x}[k+1] = \mathbf{A}[k]\mathbf{x}[k] + \mathbf{B}[k]\mathbf{u}[k] + \mathbf{w}[k] \tag{3-2}$$

$$\mathbf{y}[k] = \mathbf{C}[k]\mathbf{x}[k] + \mathbf{D}[k]\mathbf{u}[k] + \mathbf{v}[k] \tag{3-3}$$

where $\mathbf{w}[k]$ and $\mathbf{v}[k]$ are random variables representing the *process noise* and *measurement noise*, respectively.

The process and measurement noise are assumed to be independent, zero-mean Gaussian white noise sequences with probability distributions:

$$\mathbf{w}[k] \sim \mathcal{N}(0, \mathbf{W}[k]) \tag{3-4}$$

$$\mathbf{v}[k] \sim \mathcal{N}(0, \mathbf{V}[k]). \tag{3-5}$$

Furthermore, the prior distribution of the state-vector is also assumed to be Gaussian,

$$\mathbf{x}[0] \sim \mathcal{N}(\overline{\mathbf{x}}[0], \mathbf{P}[0]). \tag{3-6}$$

here, $\overline{x}[0]$ denotes the mean of the initial state-vector $\mathbf{x}[0]$ and $\mathbf{P}[0]$ the initial estimate of the covariance matrix $\mathbf{P}[0] = \mathrm{E}\left[(\mathbf{x}[0] - \overline{\mathbf{x}}[0])(\mathbf{x}[0] - \overline{\mathbf{x}}[0])^T\right]$ [36].

3-2-2 The Kalman Filter Algorithm

The Kalman filter algorithm can be divided into two stages: the prediction step (3-7)–(3-8) and the update step (3-9)–(3-12) [22]. For ease of notation the system and measurement covariance matrices are assumed to be time invariant (so $\mathbf{A}[k] = \mathbf{A}$, $\mathbf{B}[k] = \mathbf{B}$, $\mathbf{W}[k] = \mathbf{W}$ etc.), as for the time-varying case the equations remain the same (with some extra indices). The hat operator '$\hat{}$' indicates an estimate of a variable, and the superscripts '$-$' and '$+$' indicate the *a priori* (i.e. predicted) and the *a posteriori* (i.e. updated) estimates, respectively.

Prediction step:

$$\text{Predicted state estimate} \quad \hat{\mathbf{x}}[k]^- = \mathbf{A}\,\hat{\mathbf{x}}\,[k-1]^+ + \mathbf{B}\,\mathbf{u}[k-1] \tag{3-7}$$

$$\text{Predicted covariance} \quad \mathbf{P}[k]^- = \mathbf{A}\,\mathbf{P}[k-1]^+\,\mathbf{A}^T + \mathbf{W} \tag{3-8}$$

Update step:

$$\text{Measurement residual} \quad \mathbf{v}[k] = \mathbf{y}[k] - \mathbf{C}\,\hat{\mathbf{x}}[k]^- \tag{3-9}$$

$$\text{Residual covariance matrix} \quad \mathbf{S}[k] = \mathbf{C}\,\mathbf{P}[k]^-\,\mathbf{C}^T + \mathbf{V} \tag{3-10}$$

$$\text{Kalman gain} \quad \mathbf{K}[k] = \mathbf{P}[k]^-\,\mathbf{C}^T\,\mathbf{S}[k]^{-1} \tag{3-11}$$

$$\text{Updated state estimate} \quad \hat{\mathbf{x}}[k]^+ = \hat{\mathbf{x}}[k]^- + \mathbf{K}[k]\mathbf{v}[k] \tag{3-12}$$

$$\text{Updated covariance matrix} \quad \mathbf{P}[k]^+ = \mathbf{P}[k]^- - \mathbf{K}[k]\,\mathbf{S}[k]\,\mathbf{K}[k]^T \tag{3-13}$$

If a system can be described by Equations (3-2)–(3-6), the Kalman filter provides a statistically optimal estimate of the state-vector. This estimate is statistically optimal because the Kalman filter minimizes the variance of the state reconstruction error, the mean squared error. This optimality of the Kalman filter has been proven extensively [18, 36], and will therefore not be derived here.

3-2-3 Tuning of Filter Parameters

The tuning of the filter parameters is not a trivial part of implementing the Kalman filter. While the measurement noise covariance can generally be estimated ahead of time using measurement data, finding a proper expression for the process covariance often proves more difficult. This is due to the fact that the process is, in most cases, difficult to observe directly.

Finding a good expression for the process noise covariance is generally a matter of tuning. In case of a fairly poor process model, the Kalman filter can still produce decent results when enough uncertainty is 'injected' in the model through the process noise covariance [44]. In this case more emphasis is given to the measurements of the system than the system model, which means that the measurements should be sufficiently accurate.

3-3 Partitioned Update Kalman Filter

The inversion of $\mathbf{S}[k] \in \mathbb{R}^{J\times J}$ is one of the most time consuming operations of the Kalman filter, as it has a computational complexity of $\mathcal{O}\left(J^3\right)$ [13]. Thus, if the number of measurements J is large, this calculation can quickly become computationally expensive. To mitigate

this, there is a way to calculate the Kalman update one measurement at a time, using a partitioned update Kalman filter (PUKF) [34, 38]. This way, the calculation of $\mathbf{S}^{-1}$ is reduced to J scalar inversions. To be able to apply this version of the Kalman filter to the system (3-2)–(3-3), the measurement covariance matrix ($\mathbf{V}[k]$) must be diagonal [38].

The PUKF works by sequentially calculating the Kalman update for every measurement and using the partially updated state-vector as a prior for the next measurement. Suppose $\mathbf{y}[k] \in \mathbb{R}^J$, then the update steps (3-10)–(3-13) for each measurement ($y_j[k]$ for $j = 1, 2, \ldots, J$) become:

$$v_j[k] = y_j[k] - \mathbf{C}(j,:)\hat{\mathbf{x}}_{j-1}[k]^+ \tag{3-14}$$

$$s_j[k] = \mathbf{C}(j,:)\,\mathbf{P}_{j-1}[k]^+\,\mathbf{C}(j,:)^T + \mathbf{V}(j,j) \tag{3-15}$$

$$\mathbf{k}_j[k] = \frac{\mathbf{P}_{j-1}[k]^+\,\mathbf{C}(j,:)^T}{s_j[k]} \tag{3-16}$$

$$\hat{\mathbf{x}}_j[k]^+ = \hat{\mathbf{x}}_{j-1}[k]^+ + \mathbf{k}_j[k]v_j[k] \tag{3-17}$$

$$\mathbf{P}_j[k]^+ = \mathbf{P}_{j-1}[k]^+ - \mathbf{k}_j[k]s_j[k]\mathbf{k}_j[k]^T \tag{3-18}$$

The starting values of the update are the *a priori* estimates of the state-vector and state covariance: $\hat{\mathbf{x}}_1[k]^+ = \hat{\mathbf{x}}[k]^-$ and $\mathbf{P}_1[k]^+ = \mathbf{P}[k]^-$. The *a posteriori* estimates of the state-vector and state covariance are equal to the updates corresponding to the last measurement: $\mathbf{P}[k]^+ = \mathbf{P}_J[k]^+$ and $\hat{\mathbf{x}}[k]^+ = \hat{\mathbf{x}}_J[k]^+$

3-4 Summary

This chapter has provided a brief introduction into the Kalman filter. First the basic concept of state-space modeling was introduced (Section 3-1). Then, background information on the Kalman filter algorithm was provided (Section 3-2). And, lastly a way to circumvent the inversion of large scale matrices in the Kalman filter algorithm was proposed (Section 3-3).

Thus, after this chapter, background information has been provided on all the key ingredients that are needed for the tensor-networked Kalman filter. The next chapter will introduce this tensor-networked Kalman filter and explain how it can be used to perform streaming video completion.

Chapter 4

Kalman Filter for Video Completion

In this chapter a novel method for streaming video completion using a tensor-networked Kalman filter will be introduced. This method was first introduced by the authors of [24], this thesis will further expand on their research. The main idea behind using the Kalman filter to perform video completion, is to model the frames of the video as the state-vector (where each pixel represents one state) and to use the Kalman filter to estimate the 'unmeasured' states, which in this case are the corrupted pixels.

In Section 4-1 a state-space model of a video is derived. In the following section the tensor-networked Kalman filter is introduced (Section 4-2). This section includes an explanation on why tensor-networks are needed and what this means for the Kalman filter. Section 4-3 provides argumentation on how the initial values of the Kalman filter can be selected. Thereafter, Section 4-4 explains how the Kalman filter variables can be converted into TT-format.

4-1 Modeling a Video

To able to use a Kalman filter for streaming video completion, it is important to first establish how a video could be modeled as a state-space system. To find such a state-space representation more information is needed on how a video is represented. Therefore, we will first discuss how a video can be represented mathematically (Section 4-1-1), before deriving a state-space model of a video (Section 4-1-2). Furthermore, in Section 4-1-3 a case is made for subtracting the background of the video from the frames prior to the estimation.

4-1-1 Mathematical Representation Video Data

A video consists of a sequence of images, or *frames* as they are called in this context. The speed at which these frames arrive on a screen is decided by the *frame rate* of the video. The frame rate is usually expressed in terms of the number of frames per second (fps). For broadcast television this frame rate is set at 24 fps [45].

Figure 4-1: Example of how a (part of a) frame can be represented by a matrix [1].

An image consists of pixels. Each pixel represents a small part of the image. The intensity of a pixel decides how dark or light this part of the image is. The value the intensity of a pixel takes on, is represented by an 8-bit integer value. So, a pixel can be represented by a integer value between 0 and 255. If the intensity of a pixel is 0, the pixel is at its lowest intensity (black). And if the intensity is 255, the pixel has the highest possible intensity (white). The number of pixels per image decides the resolution of the image (or frame). In general, this resolution is expressed as the number of pixels per dimension. An HD video has a resolution of 1080×1920, which means that it has 1080 pixels along the height of the frame and 1920 along the width of the frame [32].

Images can be either in grayscale or in color. When an image is in grayscale, all pixels take on a value on a scale between black (0) and white (255). When an image is in color, colors are represented by three component intensities, such as red, green and blue (RGB) [32]. Thus, in a color image each pixel is represented by three different RGB intensities (Figure 4-2).

A grayscale image can easily be represented by a matrix. Each matrix element corresponds to one of the pixels in the image. The value of this element is simply equal to its intensity. In Figure 4-1 an example is shown of how a section of a frame can be represented by a matrix. Thus, a frame of size $M \times N$ can be represented by a matrix $\mathbf{X} \in \mathbb{R}^{M \times N}$. A grayscale video with T frames can therefore be expressed as a sequence of matrices: $\{\mathbf{X}[1], \mathbf{X}[2], \ldots, \mathbf{X}[T]\}$.

Each individual RGB component in a color image can be represented by a matrix in a similar way as for grayscale images. Instead of one single matrix to represent each frame, now three matrices represent one video frame. A color video can thus be represented by three different matrix sequences. In video completion each individual component color of an RGB video is often treated separately [40].

4-1-2 The State-Space Model

In order to use the Kalman filter to fill in the missing pixels in a video, it is necessary to find a state-space representation of a video. This state-space system models the changes in video frames over time. A video is, by nature, a discrete-time system. It consists of a discrete sequence of events (i.e. the frames). Thus, it needs to be modeled by a discrete-time

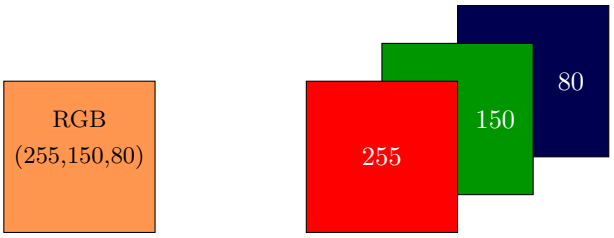


Figure 4-2: RGB pixel illustration.

state-space system. The structure of this model is assumed to be similar to (3-2)–(3-3), the underlying dynamics of the Kalman filter. Thus, we assume that the system is linear and that its noise is zero-mean Gaussian. Furthermore, because the system cannot have a control input, the input matrix of the state-space system is equal to zero.

In order to find a state-space model of a video, we must first define the states of the system. Each frame (of a grayscale video) that is represented by a matrix $\mathbf{X}[k] \in \mathbb{R}^{M \times N}$ can be transformed into a vector $\mathbf{x}[k] \in \mathbb{R}^{MN}$ by stacking the columns of the matrix (4-1). In MATLAB this conversion can be done by a single call to the `reshape` function: $\mathbf{x}[k] =$ `reshape` $(\mathbf{X}[k], MN, 1)$. Using this vectorization of the frames, the state-vector of the state-space system is defined as the frames of the video.

$$\mathbf{X}[k] = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,N} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M,1} & x_{M,2} & \cdots & x_{M,N} \end{bmatrix} \quad \rightarrow \quad \mathbf{x}[k] = \left[\begin{array}{c} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{M,1} \\ \hline x_{1,2} \\ x_{2,2} \\ \vdots \\ x_{M,2} \\ \hline \vdots \\ \hline x_{1,N} \\ x_{2,N} \\ \vdots \\ x_{M,N} \end{array}\right] \tag{4-1}$$

Determining an expression for the state-transition matrix is difficult. A video is a highly unpredictable process. Anything can happen. And what has happened in the past is not necessarily a good predictor of what will happen in the future. What can be said about video frames, however, is that two consecutive frames are highly similar. The difference between pixels at the same location between two consecutive frames is generally small. This is due to the high speed at which the frames are captured. To showcase this, Figure 4-3 shows a histogram of the difference between pixel values of two consecutive frames, using the Grand Central Station dataset [47]. Clearly, most values are around zero. Because of this high similarity between frames and the unpredictability of the process, the state-transition matrix is chosen to be identity.

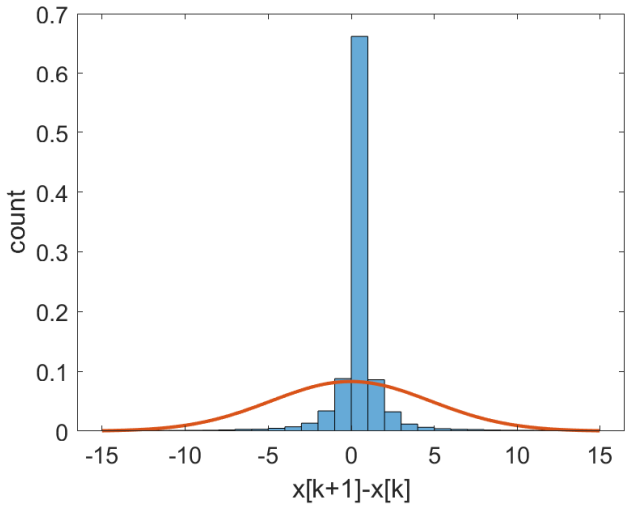


Figure 4-3: Histogram of the difference between pixel values of two consecutive frames ($\mathbf{x}[k+1] - \mathbf{x}[k]$). The count is normalized over the number of values in the bins. The red line shows the Gaussian distribution fitted with the mean and the variance of the data. The data used is from the Grand Central Station dataset [47].

Any difference between the frames must therefore be captured by the process noise. To use a Kalman filter on a system, the process noise should be zero-mean and (approximately) Gaussian. From Figure 4-3 it is clear that this process noise is indeed zero-mean when the state-transition matrix is equal to identity, since $\mathbf{w}[k] = \mathbf{x}[k+1] - \mathbf{x}[k]$. The shape of the histogram also provides some justification for the Gaussian distribution presumption. The histogram is approximately symmetric around zero, its mean. It has a peak at its mean and its values recede to zero the further away they are from the mean. These are all properties of a Gaussian distribution. Though, from Figure 4-3 one can also see that the histogram has a higher peak at zero than its corresponding Gaussian distribution (the red line).

The measurements of the system are simply the uncorrupted pixels of each frame. The location of these pixels are stored in the measurement matrix of the state-space system ($\mathbf{C}$). Each row of this matrix thus contains only one value equal to 1, the rest are all zeros. In this thesis it is presumed that the location of the uncorrupted pixels are known. Furthermore, it is assumed that these locations are always the same. Thus, the measurement matrix is time-invariant[1]. Because we have direct access to the uncorrupted pixels, the measurements are noise-free, so $\mathbf{v}[k] = 0$.

This leads to the following state-space system [24],

$$\mathbf{x}[k+1] = \mathbf{x}[k] + \mathbf{w}[k] \tag{4-2}$$

$$\mathbf{y}[k] = \mathbf{C}\,\mathbf{x}[k] \tag{4-3}$$

where $\mathbf{x}[k] \in \mathbb{R}^{MN}$ represents the video frame at time-step k and $\mathbf{y}[k] \in \mathbb{R}^{J}$ contains the uncorrupted pixels.

[1]Please note that this is just a presumption to make calculations more tractable. It is of course possible to adjust the measurement matrix at each time-step to account for changes in the corrupted pixels.

Figure 4-4: Mean of the Grand Central Station Video (1000 frames).

4-1-3 Subtracting the Background

In the previous section a state-space system was derived where the state-vectors of the system are the video frames and the state variables are the corresponding pixel values. Pixel values range between 0 and 255 and are thus always positive. This while the Kalman filter estimates can take on any real value. A mapping of the pixel values could be used to ensure that the states can also hold negative values and to make the distribution more even.

This mapping should be done while keeping the Kalman filter properties such as the covariances the same. One way to ensure this, is to subtract the mean from the states. In case of a fixed-camera video, which is often the case in surveillance footage, the mean is (approximately) equal to the background of the video as long as this background is static (Figure 4-4). A static background in this case is a background that, during the duration of the video completion, does not change over time. If, for instance, the camera is located somewhere where the light intensity changes continuously throughout the day, this background would not be static.

Subtracting the background from the video frames holds several advantages. One is that the states can hold negative values which leads to a more even distribution of values. Another advantage is that the resulting vector (or matrix) is approximately sparse. This is due to the fact that, in this new vector, only the values that correspond to pixels that are part of the foreground contain large values. To illustrate this in Figure 4-5 an image plot is shown of the frame when the background is subtracted ($\mathbf{X}_{foreground} = \mathbf{X}_{frame} - \mathbf{X}_{background}$). As can be seen most of this image plot is green, which corresponds to values of around zero.

The sparsity resulting from the background subtraction, also leads to advantageous properties for the tensor-network representation, as sparse vectors and matrices generally have lower TT(m)-ranks. More on this in Section 4-2-2.

The new state-vector ($\mathbf{x}_F$) resulting from the background subtraction is shown in (4-4). Here, $\mathbf{x}_F$ represents the 'foreground' of the frames, and $\mathbf{x}_B$ the background which is approximately equal to the mean of the frames $\overline{\mathbf{x}}[k]$ (in case of a static background).

$$\mathbf{x}_F[k] = \mathbf{x}[k] - \mathbf{x}_B \approx \mathbf{x}[k] - \overline{\mathbf{x}} \tag{4-4}$$

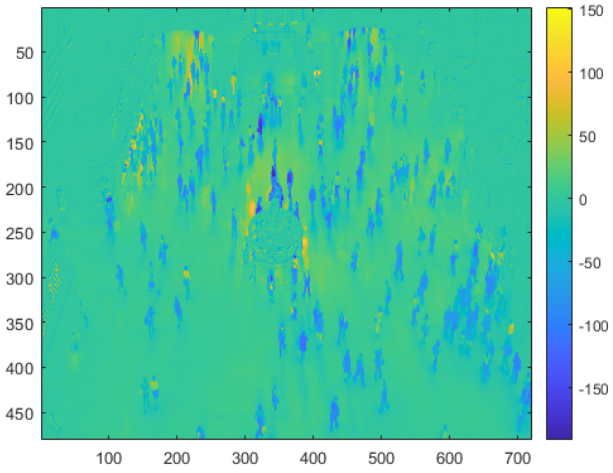


Figure 4-5: Image plot of frame with background subtracted ($\mathbf{X}_{foreground} = \mathbf{X}_{frame} - \mathbf{X}_{background}$).

Covariance For a fixed-camera view with a non-changing background, it can be determined that the covariance of the frames $\mathbf{x}[k]$ is equal to the covariance of the foreground $\mathbf{x}_F[k]$. A frame can be separated into a foreground and a background part (time indices are omitted for notational ease):

$$\mathbf{x} = \mathbf{x}_F + \mathbf{x}_B. \tag{4-5}$$

Suppose that it does not necessarily hold that $\overline{\mathbf{x}} = \mathbf{x}_B$, then the covariance of $\mathbf{x}$ equals:

$$\begin{aligned}\mathbf{P} &= E[(\mathbf{x} - \overline{\mathbf{x}})(\mathbf{x} - \overline{\mathbf{x}})^T] = E\left[(\mathbf{x}_F + \mathbf{x}_B - (\overline{\mathbf{x}}_F + \overline{\mathbf{x}}_B))\,(\mathbf{x}_F + \mathbf{x}_B - (\overline{\mathbf{x}}_F + \overline{\mathbf{x}}_B))^T\right] \\ &= E\left[(\mathbf{x}_F - \overline{\mathbf{x}}_F)(\mathbf{x}_F - \overline{\mathbf{x}}_F)^T\right] + E\left[(\mathbf{x}_B - \overline{\mathbf{x}}_B)(\mathbf{x}_B - \overline{\mathbf{x}}_B)^T\right] \\ &\quad + E\left[(\mathbf{x}_F - \overline{\mathbf{x}}_F)(\mathbf{x}_B - \overline{\mathbf{x}}_B)^T\right] + E\left[(\mathbf{x}_B - \overline{\mathbf{x}}_B)(\mathbf{x}_F - \overline{\mathbf{x}}_F)^T\right]\end{aligned}$$

Assuming the background remains stationary it follows that $\mathbf{x}_B = \overline{\mathbf{x}}_B$. And thus, the covariance matrix of the frames is equal to the covariance matrix of the foreground,

$$\mathbf{P} = E\left[(\mathbf{x}_F - \overline{\mathbf{x}}_F)(\mathbf{x}_F - \overline{\mathbf{x}}_F)^T\right]. \tag{4-6}$$

4-1-4 Color Videos

The state-space model derived above is only for frames with one component intensity, so for grayscale videos. There are two options to expand this model for color videos. One is to extend the dimensions of the state-space system to allow for all three RGB component intensities of a frame to be stacked in one vector:

$$\mathbf{x}[k] = \begin{bmatrix}\mathbf{x}_R[k] \\ \mathbf{x}_G[k] \\ \mathbf{x}_B[k]\end{bmatrix}.$$

Another way is to define three separate state-space systems, where the update for each component intensity is performed in parallel. Each frame is now defined not by one state-vector $\mathbf{x}[k]$, but by a tuple of three state-vectors that correspond to the three RGB components:

$$\{\mathbf{x}_R[k], \mathbf{x}_G[k], \mathbf{x}_B[k]\}.$$

The state-space system that is used for each component is the same as for the grayscale videos (4-2)–(4-3), though each component has different initial states and covariance matrices. Because this method allows for the completion of the three intensities to be computed in parallel and does not require any adjustments to the state-space model, this method was chosen to complete color videos.

4-2 Tensor-Networked Kalman Filter

In the streaming video completion problem, a streaming video gets suddenly corrupted. Because this is a streaming video, the completion should preferably be done online, in real-time. The Kalman filter is a recursive estimator, it only uses information of past events (states) to estimate the current state. This makes it suitable for application to the streaming video completion problem, because calculations can be done online. In this section the tensor-networked Kalman filter will therefore by introduced as a method to perform video completion.

4-2-1 Kalman Equations

In the previous section a state-space model was derived that describes the changes in the video frames over time. Applying a Kalman filter to this system, the Kalman filter equations (3-7)–(3-13) reduce to (4-7)–(4-13). The initial state-vector and covariance matrices can be determined using the uncorrupted frames that are observed prior to the corruption 'event'. How this can be done will be explained in Section 4-3-1.

Prediction step:

Predicted state estimate	$\hat{\mathbf{x}}[k]^- = \hat{\mathbf{x}}\,[k-1]^+$	(4-7)
Predicted covariance	$\mathbf{P}[k]^- = \mathbf{P}[k-1]^+ + \mathbf{W}[k]$	(4-8)

Update step:

Measurement residual	$\mathbf{v}[k] = \mathbf{y}[k] - \mathbf{C}\,\hat{\mathbf{x}}[k]^-$	(4-9)
Residual covariance matrix	$\mathbf{S}[k] = \mathbf{C}\,\mathbf{P}[k]^-\,\mathbf{C}^T$	(4-10)
Kalman gain	$\mathbf{K}[k] = \mathbf{P}[k]^-\,\mathbf{C}^T\,\mathbf{S}[k]^{-1}$	(4-11)
Updated state estimate	$\hat{\mathbf{x}}[k]^+ = \hat{\mathbf{x}}[k]^- + \mathbf{K}[k]\mathbf{v}[k]$	(4-12)
Updated covariance matrix	$\mathbf{P}[k]^+ = \mathbf{P}[k]^- - \mathbf{K}[k]\,\mathbf{S}[k]\,\mathbf{K}[k]^T$	(4-13)

4-2-2 Why use Tensor-Networks?

One major obstacle in using the Kalman filter for video completion, is that the dimensions of the state-space system become so large that the storage of the system parameters is prohibitively expensive. For a standard-definition (SD) 480×720 video, for instance, the state-vector becomes: $\mathbf{x}[k] \in \mathbb{R}^{345600}$, so its size is of $\mathcal{O}(10^5)$. This means that the covariance

matrix of $\mathbf{x}[k]$, $\mathbf{P}[k]$, is of size 345600×345600 ($\mathcal{O}(10^{11})$). If one were to store a matrix of this size on a computer, one would need about 20 GB of memory. For any regular computer, this is just not feasible. Therefore, a memory efficient way of expressing both $\mathbf{x}[k]$ and $\mathbf{P}[k]$ is necessary.

One way to do this is by using tensor-networks. From Section 2-2-3 it is clear that by using quantized tensor-trains to represent matrices and vectors, a high compression ratio can be achieved. For a video of size $M \times N$ the state-vector can be reduced from $\mathcal{O}(MN)$ to $\mathcal{O}\left(R^2 \log_q(MN)\right)$, where R is the maximum rank of the TT representing the state-vector and q is the quantization parameter ($MN = q^d$).

4-2-3 Tensor-Networked Kalman Filter Algorithm

Matrix and vector products can easily be computed in TT-format (see Section 2-3-1). Matrix inversions however are more difficult. While there is some research into efficient algorithms for matrix inversion in TT-format, it is not yet desirable to utilize these. This is due to their long computation times, the large TTm-ranks of the inverse (compared to the original TTm) and the fact that they have not yet been proven to converge [30]. To circumvent having to perform matrix inversions, the partitioned update Kalman filter (PUKF) can be used. This reduces the matrix inversion to a number of scalar inversions, while achieving the same result as a regular Kalman filter (Section 3-3). The PUKF can be used for the current system because the measurements are not correlated, since the measurement covariance is equal to zero.

Because the measurements are of distinct state variables (i.e. the uncorrupted pixels), the following multiplications of the PUKF, $\mathbf{C}(j,:)\mathbf{x}_j$, $\mathbf{C}(j,:)\,\mathbf{P}_{j-1}\,\mathbf{C}(j,:)^T$ and $\mathbf{P}_{j-1}\,\mathbf{C}(j,:)^T$, are reduced to a simple extraction of the vector and matrix values. If the location of the measurements are stored in a vector $\mathbf{c} \in \mathbb{R}^J$, then this reduces to,

$$\mathbf{C}(j,:)\mathbf{x}_j = \mathbf{x}_{j-1}(c_j) \tag{4-14}$$

$$\mathbf{C}(j,:)\,\mathbf{P}_{j-1}\,\mathbf{C}(j,:)^T = \mathbf{P}_{j-1}(c_j, c_j) \tag{4-15}$$

$$\mathbf{P}_{j-1}\,\mathbf{C}(j,:)^T = \mathbf{P}_{j-1}(:, c_j). \tag{4-16}$$

Algorithm 4 shows the tensor-networked Kalman filter with partitioned updates. Note that every operation in this algorithm can be performed using the algorithms stated in Section 2-3. Only multiplication and addition operations are computed (subtraction is equal to addition and multiplication with -1). After these operations the ranks of the tensor-trains can increase. If no rounding occurs after each update step, these ranks will quickly climb to such high values that the calculations are no longer tractable. Therefore, rounding should occur when the ranks of the tensor-trains become too high.

Choosing when to perform this rounding influences the computational speed of the Kalman filter update. If after every iteration (of the `for`-loop) the tensor-train (TT)(m)'s are rounded, this rounding can be performed relatively fast due to the small TT-ranks. However, because the rounding function is called every iteration, this does slow down the calculation of the Kalman update. Increasing the ranks at which rounding occurs (R_x^{max} and R_P^{max}), leads to longer computation times of the rounding function. However, because the rounding function

is called less often, it can lead to a faster computation of the overall Kalman update. Thus, in the selection of these values a trade-off needs to be found, that ensures that the rounding is still relatively fast, while reducing the number of calls to this function.

Algorithm 4: TN Kalman Filter[2]

Data: $\mathbf{x}[k-1] \in \mathbb{R}^I$, $\mathbf{P}[k-1] \in \mathbb{R}^{I \times I}$, $\mathbf{W}[k] \in \mathbb{R}^{I \times I}$, in their corresponding TT-format of dimension d. Maximum and desired rank of TT representation of $\mathbf{x}[k-1]$: R_x^{max} and R_x; maximum and desired rank of TTm representation of $\mathbf{P}[k-1]$: R_P^{max} and R_P. The measured values (not in TT-format) $\mathbf{y}[k] \in \mathbb{R}^J$. Locations of measured values $\mathbf{c} \in \mathbb{R}^J$.

Result: Updated state and covariance: $\mathbf{x}[k]$, $\mathbf{P}[k]$.

{Prediction}

$\mathbf{P}_0 = \mathbf{P}[k-1] + \mathbf{W}[k]$

$\mathbf{P}_0 = \texttt{rounding}(\mathbf{P}_0, R_P)$ `// Algorithm 3`

$\mathbf{x}_0 = \mathbf{x}[k]$

for $j = 1, 2, \ldots, J$ **do**

- {Update}
- $v_j = y_j[k] - \mathbf{x}_{j-1}(c_j)$
- $s_j = \mathbf{P}_{j-1}(c_j, c_j)$
- $\mathbf{k}_j = \mathbf{P}_{j-1}(:, c_j)/s_j$
- $\mathbf{x}_j = \mathbf{x}_{j-1} + \mathbf{k}_j v_j$
- $\mathbf{P}_j = \mathbf{P}_{j-1} - \mathbf{k}_j s_j \mathbf{k}_j^T$
- {TT-rounding}
- **if** $\max(\text{rank}(\mathbf{P}_j)) \leq R_P^{max}$ **then**
  - $\mathbf{P}_j = \texttt{rounding}(\mathbf{P}_j, R_P)$
- **end**
- **if** $\max(\text{rank}(\mathbf{x}_j)) \leq R_x^{max}$ **then**
  - $\mathbf{x}_j = \texttt{rounding}(\mathbf{x}_j, R_x)$
- **end**

end

Set: $\mathbf{x}[k] = \mathbf{x}_J$ and $\mathbf{P}[k] = \mathbf{P}_J$

[2] code: https://gitlab.com/seline/thesis/-/blob/a907866468cef6f74655d84bb3255b06f7ff5958/Matlab_Code/Kalman_Filter/Update/kalmanUpdateTTPixel.m

4-3 Initialization of the Kalman Filter

In this section the initial values of the Kalman filter and the process covariance matrix will be derived.

4-3-1 Initial State Mean and Covariance

The Kalman filter is a recursive estimator. Thus, it needs some information on the initial state-vector $\mathbf{x}[0]$ to perform the estimation. From Section 3-1 we know that the Kalman filter assumes the following Gaussian distribution for the initial state-vector,

$$\mathbf{x}[0] \sim \mathcal{N}(\overline{\mathbf{x}}[0], \mathbf{P}[0]). \tag{4-17}$$

where $\overline{x}[0]$ denotes the mean of the initial state-vector $\mathbf{x}[0]$ and $\mathbf{P}[0]$ the initial estimate of the covariance matrix $\mathbf{P}[0] = \mathrm{E}\left[(\mathbf{x}[0] - \overline{\mathbf{x}}[0])(\mathbf{x}[0] - \overline{\mathbf{x}}[0])^T\right]$.

In the streaming video completion problem the video gets corrupted at a certain point in time t_c (Figure 1-1). This means that all frames prior to the corruption are observable and can be used to form estimates for the mean of the initial state-vector and its covariance matrix. It is even possible to use the exact value of the initial state-vector, since this value corresponds to frame $[t_c - 1]$, which is is not corrupted.

Using the last uncorrupted frame as the initial value for the state-vector (with or without background subtraction), would mean that the covariance matrix should be very small. The covariance matrix cannot be equal to zero, as it should be symmetric positive definite. However, because the estimate is so accurate, its values should be low enough to reflect this. One way to initialize the state covariance matrix is to define it as a diagonal matrix with equal values on the diagonal (4-18).

$$\mathbf{P}[0] = \sigma_P^2 \mathbf{I} \tag{4-18}$$

The variable σ_P^2 represents the variance of each state, of each pixel. In the case that a video contains multiple frames with only background images, this value could be determined by calculating the (average) variance between these 'background frames'. This variance would then be somewhat equal to the variance of the measurement noise of the original video capture. This value should be relatively low and provides a decent estimate of the accuracy of the initial state-vector. If it is not possible to calculate this variance based on a number of background frames, because these frames are simply not available, choosing $\sigma_P^2 = 1$, so $\mathbf{P}[0] = \mathbf{I}$, generally provides a good result.

Choosing the initial state-vector as the last uncorrupted frame and using relatively low values for the covariance matrix can lead to a problem. It can cause the Kalman filter to place such importance on the initial value of the state-space system that it does not properly update the states of the system using the measurements. This can lead to a certain 'shadow effect', where a sort of shadow of the initial frame remains visible even after a number of Kalman filter steps. In Figure 4-6 an example can be seen of this shadow effect.

This shadow effect can occur due to the fact that the state-transition matrix of the state-space model is equal to identity and the Kalman filter therefore assumes the states remain the same.

Figure 4-6: Example of 'shadow effect'. On the left the initial frame is shown and on the right the Kalman filter estimate after a number of time-steps.

To prevent this effect from occurring enough uncertainty (in the state-space model) should be injected by the process covariance.

Sometimes even increasing the values of the process covariance cannot remove this shadow effect. This usually occurs when the amount of observed pixels is very low (less than 10%) and the object that is moving through the frame is relatively large (such as in the figure) . In this case the shadow effect can be prevented by choosing the background of the video as the initial state-vector. This would mean that a lot more emphasis is placed on the measurements of the system than the initial state. In this case the variance σ_P^2 should be calculated using the difference between the pixels of the background (the mean) and the last uncorrupted frame (4-19).

$$\sigma_P^2 = \frac{1}{I}\sum_{i=1}^{I}(x_i[t_c - 1] - \overline{x}_i)^2, \quad \mathbf{x} \in \mathbb{R}^{I=MN} \tag{4-19}$$

4-3-2 Process Covariance

Because the state-transition matrix is equal to identity, all changes between the frames must be captured by the process covariance (4-20). Therefore, this matrix should contain information on how the pixels of consecutive frames are related to each other. Finding an exact representation for the process covariance is difficult, due to the high unpredictability of the process. However, by using some video properties it is possible to find an estimate that performs well enough to perform the video completion.

$$\mathbf{W}[k] = \mathrm{E}\left[\mathbf{w}[k]\mathbf{w}[k]^T\right] = \mathrm{E}\left[(\mathbf{x}[k+1] - \mathbf{x}[k])(\mathbf{x}[k+1] - \mathbf{x}[k])^T\right] \tag{4-20}$$

One of those properties is the fact that neighboring pixels have a higher similarity than pixels that are further removed in the frames. This property can be verified by looking at the (average) correlation between pixels as a function of the distance between the pixels. The

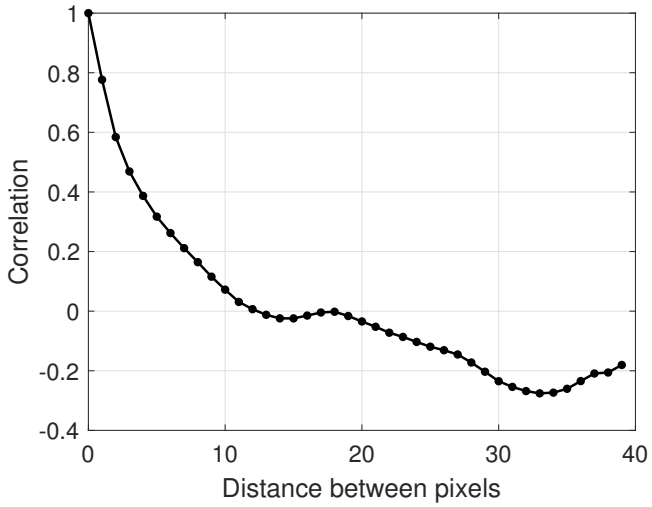


Figure 4-7: Correlation between pixels. The x-axis shows the distance between the pixels (in number of pixels) and the y-axis shows the correlation coefficient. Data is from the Grand Central Station dataset [47].

distance between the pixels can be defined as follows. If two pixels are next to, or above or below each other this distance is equal to one. For every 'next pixel' (i.e. every 'border' that is crossed) this value increases by one. In Figure 4-8 an example is provided to showcase how this distance can be determined.

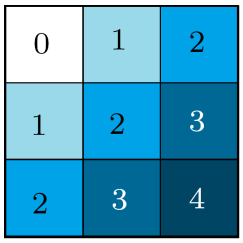


Figure 4-8: Example of the distance between pixels on a 3×3 grid, relative to the pixel in the top left corner. The distance between the pixel and the pixel in the top left corner is denoted at each pixel location.

In Figure 4-7 the correlation between pixels is plotted as a function of the distance between the pixels, using different patches of frames from the Grand Central Station dataset [47]. Clearly, when two pixels are near each other the correlation between those pixels is higher than when they are not.

A way to use this property is by defining a $\mathbf{W}[k]$ such that its values are dependent on the distance between the respective pixels; that the values in this matrix decrease the further away the corresponding pixels are from each other. Because the columns of the frames are stacked into a column vector (4-1), this matrix would have a block structure. This is due to the fact that neighboring pixels in different columns of the frame, are 'separated' due to this reshaping. The size of each block is equal size of the height of the frame (M) and the number of blocks is equal to the width of the frame (N).

To obtain this block structure the covariance matrix can be constructed using the Kronecker product of two matrices, where the size of these matrices correspond to the height (M) and the width (N) of the frames (4-21).

$$\begin{aligned} &\mathbf{W}[k] = \mathbf{W}_1[k] \otimes \mathbf{W}_2[k] \\ &\mathbf{W}_1[k] \in \mathbb{R}^{N \times N}, \quad \mathbf{W}_2[k] \in \mathbb{R}^{M \times M} \end{aligned} \tag{4-21}$$

To ensure that the values in $\mathbf{W}[k]$ decrease when the distance between the corresponding pixels becomes larger, the matrices $\mathbf{W}_1[k]$ and $\mathbf{W}_2[k]$ can be chosen as Toeplitz band matrices with values that are highest on the main diagonal and get gradually lower the further away they are from the main diagonal [24]g. In (4-22) an example of such a structure is provided. Here, $w(d)$ is a function of the distance between two pixels (d), it has a maximum at 0 and decreases for higher d, until $d = \alpha$ at which $w(\alpha) = 0$. The variable α is the *bandwidth* of the band matrix. In this thesis a linear interpolation between 1 and 0 will be used for $w(d)$ (4-23). The bandwidth of this matrix can be chosen by looking at the correlation between nearby pixels (Figure 4-7). In this case α should be around 10.

$$\mathbf{W}_n(\alpha) = \begin{bmatrix} w(0) & w(1) & \cdots & w(\alpha) & 0 & \cdots & 0 \\ w(1) & w(0) & w(1) & \dots & w(\alpha) & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & w(\alpha) & \dots & w(1) & w(0) \end{bmatrix}, \qquad n = \{1, 2\} \tag{4-22}$$

$$w(d) = \max\left(0, 1 - \frac{d}{\alpha + 1}\right) \tag{4-23}$$

Defining $\mathbf{W}_1$ and $\mathbf{W}_2$ by (4-22) and (4-23), results in a matrix $\mathbf{W}$ with ones on the main diagonal. This expression can be multiplied by a scalar value $\sigma^2_W[k]$ to ensure that the process covariance matrix is adaptable in magnitude (4-24). The values on the diagonal are now equal to $\sigma^2_W[k]$. This value is chosen to be time-dependent, since the difference between consecutive frames changes over time. To calculate σ^2_W the variance between the two previous frames can be used (4-25).

$$\mathbf{W}[k] = \sigma^2_W[k]\, \mathbf{W}_1 \otimes \mathbf{W}_2 \tag{4-24}$$

$$\sigma^2_W[k] = \frac{1}{I} \sum_{i=1}^{I} (x_i[k-1] - x_i[k-2])^2 \ , \qquad \mathbf{x} \in \mathbb{R}^{I=MN} \tag{4-25}$$

Positive Definiteness To verify that the matrix defined above is positive definite, all that is necessary is to show that both $\mathbf{W}_1$ and $\mathbf{W}_2$ are positive definite, since the Kronecker product of two positive definite matrices is positive definite [6]. The positive definiteness of $\mathbf{W}_1$ and $\mathbf{W}_2$ as defined by (4-22) and (4-23), can be verified numerically prior to the Kalman filter estimation.

Instead of having to verify it numerically, it is also possible to define $w(d)$ such that $\mathbf{W}_1$ and $\mathbf{W}_2$ (4-22) are diagonally dominant. This ensures that those matrices are always positive definite (see theorem 6.1.10 (b) of [13]).

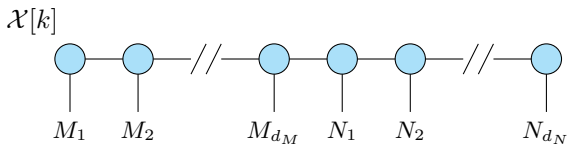


Figure 4-9: Tensor-Train of state-vector.

4-4 TT-Representation of Kalman Filter

In the tensor-networked Kalman filter TT-representations of the state-vector and covariance matrices will be used. Therefore, it is necessary to determine how these values can be accurately represented using tensor-trains and tensor-train matrices.

4-4-1 State-Vector

The state-vectors of the state-space system ($\mathbf{x}[k]$) correspond to the vectorized frames of the video. These vectors can be represented in TT-format using quantized tensor-trains. For a $M \times N$ video this means that the state-vector $\mathbf{x} \in \mathbb{R}^{MN}$ representing a video frame is first converted into a tensor

$$\mathcal{X} \in \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_{d_M} \times N_1 \times N_2 \times \cdots \times N_{d_N}},$$

where it holds that $M = M_1 M_2 \cdots M_{d_M}$ and $N = N_1 N_2 \cdots N_{d_N}$. Ideally these values ($M_i$ and N_i) should be as small as possible, as this can lead to larger compression ratio of the original vector.

The quantized state-vectors can be converted into tensor-trains using the TT-SVD algorithm (Algorithm 1). To ensure that calculations remain tractable the ranks of the tensor-trains should be truncated at a certain value. Selecting this value has consequences for the accuracy of the tensor-train approximation of the state-vector.

To verify that the frames can be approximated using a low-rank tensor-train, we can look at the singular values of a video frame. In Figure 4-10 these singular values are plotted, in the same figure also the singular values of the foreground (i.e. frame with background subtracted) are shown. From Figure 4-10 it is clear that a video frame is relatively low-rank, as its singular values rapidly decrease in size. Thus, a video frame can be approximated by a low-rank TT. Furthermore, the singular values of the foreground are smaller than those of the full frame.

Choosing the maximum ranks of the tensor-train representing the state-vector is a matter of tuning. Higher ranks could lead to a more accurate solution, but does lead to higher computation times. Therefore, it is recommended to choose low ranks when high accuracy is not that important, or perhaps not even possible in the case of a high percentage of missing pixels ($> 90\%$). When accuracy is important and computational cost is not, then the ranks could be chosen higher.

To verify if the chosen rank is sufficiently accurate one can look at the TT decomposition of the last frame, prior to the corruption. In Figure 4-13 examples are shown of TT approximations

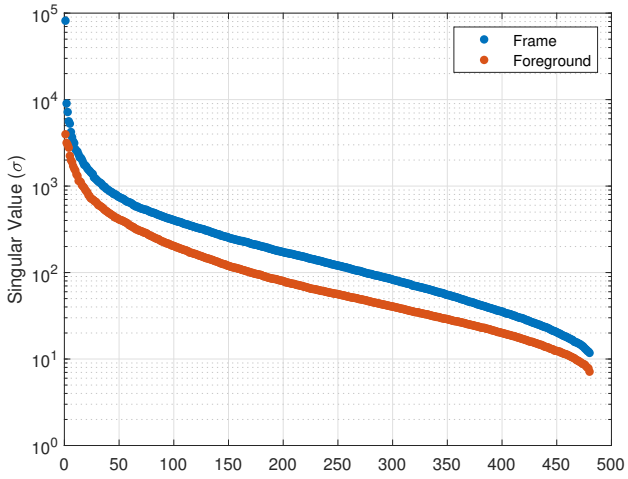


Figure 4-10: Singular values of a video frame. With (red) and without (blue) background subtraction. The video data that was used is from the Town Centre dataset [1].

of a video frame truncated at different ranks (original frame in Figure 4-12). From this figure it is clear that truncating the ranks at $R_x = 20$ or $R_x = 30$ leads to a poor approximation of the original frame. Whereas, when the ranks at truncated at $R_x = 50$ the tensor-train approximation of the video frame is significantly better. In Figure 4-14 the TT-representation of the video frames is shown when background subtraction is used. In this case even the lower-rank approximations ($R_x = 20$ and $R_x = 30$) lead to a fairly good representation of the original frame.

The relative errors corresponding to these tensor-train approximations are plotted in Figure 4-11. Clearly, when using background subtraction the ranks of the TT can be chosen lower. This coincides with the smaller singular values of the foreground (Figure 4-10). Therefore, it is recommended to always, if possible, use background subtraction for video completion using the tensor-networked Kalman filter.

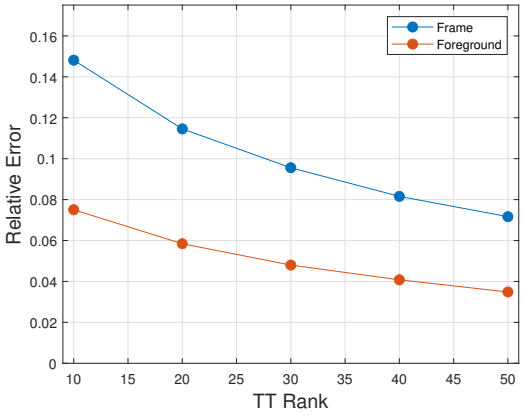
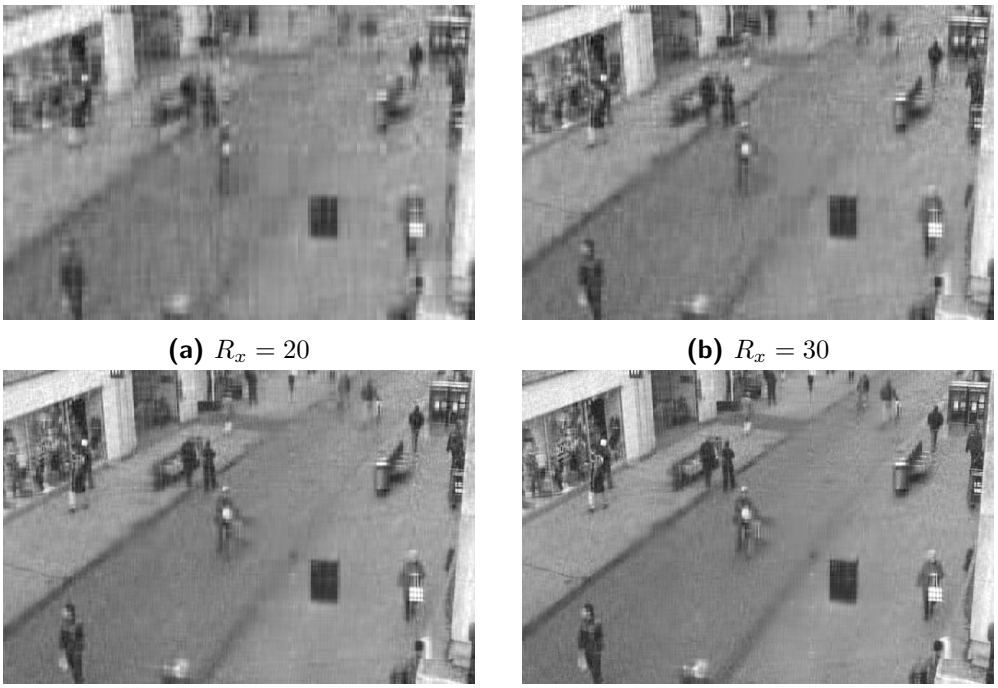


Figure 4-11: Relative error of TT-approximation of a video frame (blue) and foreground of a frame (red), as a function of the TT-rank.

Figure 4-12: Original frame [1].



(a) $R_x = 20$ **(b)** $R_x = 30$

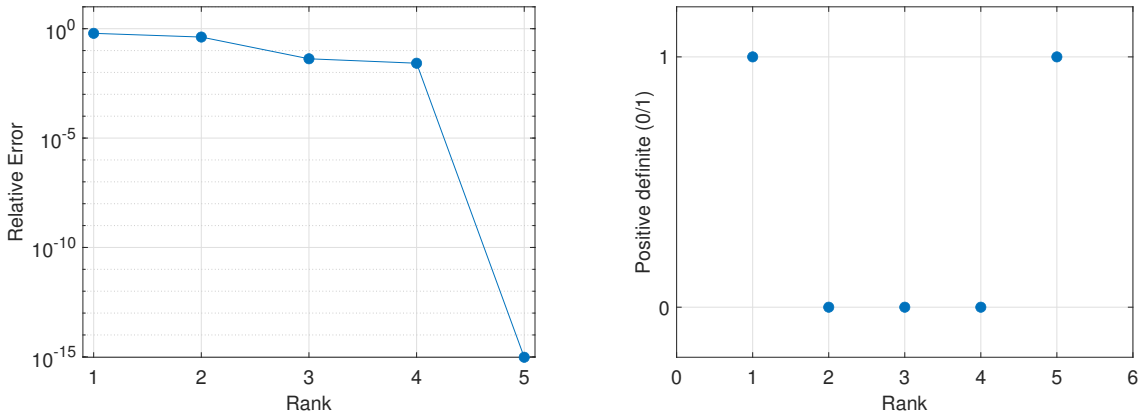
(c) $R_x = 40$ **(d)** $R_x = 50$

Figure 4-13: TT-approximation of a video frame, using different TT-ranks. Data from [1].

(a) $R_x = 20$ **(b)** $R_x = 30$

(c) $R_x = 40$ **(d)** $R_x = 50$

Figure 4-14: TT-approximation of the foreground (i.e. frame where background is subtracted prior to computing the TT decomposition), using different TT-ranks. Data from [1].



(a) Relative error **(b)** Positive definiteness

Figure 4-15: Relative error and positive definiteness of TTm approximation of $\mathbf{W}_1 \in \mathbb{R}^{480\times480}$ as a function of the maximum TTm-rank.

4-4-2 Process Covariance Matrix

The process covariance matrix is defined as the Kronecker product of two matrices, $\mathbf{W}_1 \in \mathbb{R}^{N\times N}$ and $\mathbf{W}_2 \in \mathbb{R}^{M\times M}$ (multiplied by a scalar $\sigma_W^2[k]$). Both of these matrices can be converted into tensor-train matrix (TTm)-format using the TT-SVD algorithm. The ranks of the resulting TTm's should be chosen in such a way that the positive definiteness of the matrices are preserved.

For a bandwidth of the band matrices ($\mathbf{W}_1$ and $\mathbf{W}_2$) equal to 10, it can be determined that setting the ranks of the TTm at 5 ensures that the TTm approximation is equal to the 'true' matrices. In Figure 4-15a the relative error of the TTm approximation of $\mathbf{W}_1 \in \mathbb{R}^{480\times480}$ is plotted as a function of the rank. And in Figure 4-15b the positive definiteness of these approximations are shown. This positive definiteness was determined using the Cholesky decomposition (`chol`) of the TTm's converted back to full matrix format.

Clearly, only for a rank of 1 or 5 the positive definiteness can be preserved. Generally, a rank 1 approximation preserves the positive definiteness as it simply equal to a number of Kronecker products. The rank 5 approximation is also positive definite, as in this case the TTm approximation is equal to the true matrix. Because the true covariance matrix can be exactly represented by a TTm with such low ranks, it is recommended to use this exact TTm decomposition.

4-4-3 State Covariance Matrix

Initial Covariance Matrix

The covariance matrix has dimensions $\mathbf{P} \in \mathbb{R}^{MN\times MN}$. In general, it is not possible to explicitly define this matrix, due to its large size. Therefore, the Kronecker product(s) of smaller matrices are used to set the initial state covariance matrix. When this matrix initial

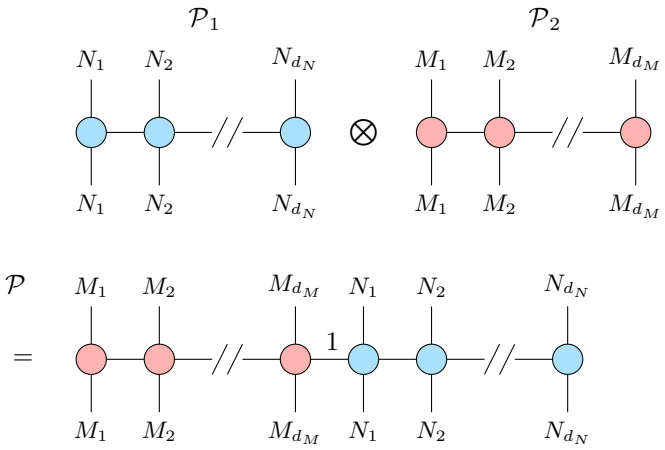


Figure 4-16: Tensor-Train matrix of the state covariance.

covariance matrix is chosen to be diagonal, the matrix can be decomposed as the Kronecker product of smaller diagonal matrices.

The covariance matrix should be quantized using similar quantization parameters as the state-vector and the process covariance matrix, to ensure that performing the Kalman filter matrix and vector products in TT(m)-format remain possible.

$$\mathcal{P} \in \mathbb{R}^{(M_1 \times M_1 \times \cdots \times M_{d_M} \times M_{d_M}) \times (N_1 \times N_1 \times \cdots \times N_{d_N} \times N_{d_N})}$$

One way to ensure this is the case, is by defining two matrices $\mathbf{P}_1 \in \mathbb{R}^{N \times N}$ and $\mathbf{P}_2 \in \mathbb{R}^{M \times M}$ for which it holds that $\mathbf{P} = \mathbf{P}_1 \otimes \mathbf{P}_2 \in \mathbb{R}^{MN \times MN}$. Matrix $\mathbf{P}_1$ can be quantized as a tensor $\mathcal{P}_1 \in \mathbb{R}^{N_1 \times N_1 \times \cdots \times N_{d_N} \times N_{d_N}}$ and matrix $\mathbf{P}_2$ as a tensor $\mathcal{P}_2 \in \mathbb{R}^{M_1 \times M_1 \times \cdots \times M_{d_M} \times M_{d_M}}$. If both of these tensors are then transformed into TTm-format, their Kronecker product can be computed in TTm-format which results in the TTm-decomposed version of $\mathbf{P}$. Figure 4-16 illustrates this.

Rank of P

In the previous section it was shown that a low-rank TTm approximation of a covariance matrix does not always remain positive definite. During simulation it was found that only when the rank of the TTm representing the state covariance matrix, is chosen as 1, the TTm approximation remains positive definite. In other cases there is a possibility that this matrix will become non positive definite, which causes the completion to become unstable, as the values of the state-vector and the covariance matrix go to ∞ and $-\infty$ (Figure 4-17). Therefore, the desired rank of the state covariance matrix (R_P) should always be chosen as 1.

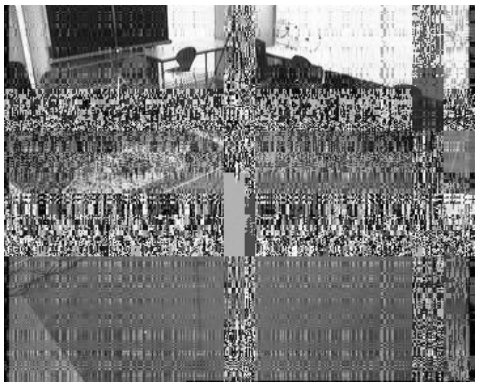


Figure 4-17: Example of estimation with non positive definite $\mathbf{P}[k]$ [9].

Chapter 5

Results & Discussion

In this chapter the tensor-networked Kalman filter as described in Chapter 4 will be applied to corrupted video data (Section 5-1). The influence of the TT-rank of the state-vector on the performance will be investigated (Section 5-2-2). Thereafter, the performance of the Kalman filter will be compared to that of a state-of-the-art adaptive matrix completion method, the proximal LMS (PLMS) [40] algorithm (Section 5-2-3).

5-1 Video Data

To be able to perform the video completion, first corrupted video data needs to be 'created'. In this thesis we will be considering the case where the missing pixels are randomly distributed throughout the frame. Furthermore, it will be assumed that the missing pixels are always at the same known locations.

5-1-1 Choosing Video Data

In this chapter data from three different videos will be used: the 4p-c0 (camera view 0) video from the EPFL data set [9], the Town Centre video [1] and the Grand Central Station data set [47] (Figure 5-1). Each of these videos have different properties. In the Grand Central Station, for instance, there are a lot of people walking through the frame, but all these people take up a relatively small part of the frame, due to the zoomed out view of the station. While in the 4p-c0 video there are much fewer people walking through the frame, but the camera is much closer to those people. The Town Centre video on the other hand, is a bit in between the other two videos, in the sense that it is not as crowded and zoomed out as the Grand Central Station video, but the camera is not as near to the people as in the 4p-c0 video.

The Town Centre video and the 4p-c0 video are both in color, whereas the Grand Central Station video is only available in grayscale. To be able to compare the results from the three videos, the Town Centre and 4p-c0 are cast as grayscale videos. This is done in MATLAB

(a) Grand Central Station **(b)** Town Centre **(c)** 4p-c0

Figure 5-1: Frames of the three different videos [1, 9, 47].

using the `rgb2gray` command. Most of the results discussed in this chapter will be for the grayscale videos. Only in Section 5-2-4, the color versions of the Town Centre and 4p-c0 video will be used, to showcase that the method also works for RGB color videos.

The Town Centre video has an original resolution of 1080×1920 (full HD). To speed up the calculations, this video was downcast to an SD resolution of 480×720. This also makes it easier to compare to the other videos, as those are already in SD resolution: 480×720 (Grand Central Station) and 288×360 (4p-c0). All the videos used have a fixed camera and a stationary background. Therefore, background subtraction will be used for all three videos prior to the application of the Kalman filter and the PLMS algorithm.

Quantization Parameters

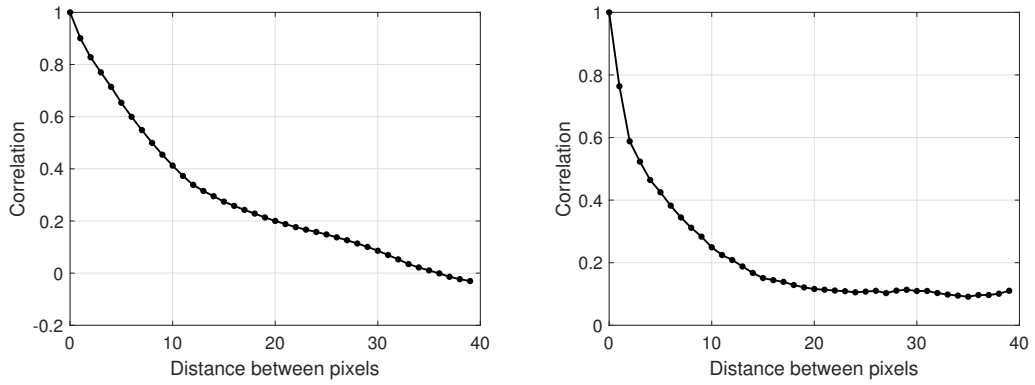
The state-vector and the covariance matrices need to be quantized along the dimensions of the videos. The dimensions of a $M \times N$ video can be quantized by $M = M_1 M_2 \cdots M_d$ and $N = N_1 N_2 \cdots N_d$. Ideally the values M_i and N_i are as small as possible, to achieve a maximal compression rate. The quantization parameters used for the video frames can be found in Table 5-1.

Video Size ($M \times N$)	Quantization Height (M)	Quantization Width (N)
480×720	$\{5, 3, 2, 2, 2, 2, 2\}$	$\{5, 3, 3, 2, 2, 2, 2\}$
288×360	$\{3, 3, 2, 2, 2, 2, 2\}$	$\{5, 3, 2, 2, 2, 2\}$

Table 5-1: Quantization parameters of the video frames.

Bandwidth

From Figure 4-7 in Section 4-3-2 a bandwidth $\alpha = 10$ for the process covariance of the Grand Central Station video can be chosen. For the other two videos similar plots can be made of the correlation between pixels (Figure 5-2). Using this figure, the bandwidth of the process covariance matrix of the 4p-c0 video was set to $\alpha = 20$. This results in a maximum TTm-rank for the TTm representation of the covariance matrix of 7, to ensure an exact representation is achieved. For the Town Centre video the bandwidth was set to $\alpha = 15$. The



(a) 4p-c0 **(b)** Town Centre

Figure 5-2: Correlation between pixels as a function of the distance.

process covariance matrix corresponding to this bandwidth, can be exactly deconstructed in a tensor-train matrix with a maximum TTm-rank of 3.

5-1-2 Creating Corrupted Data

To obtain the corrupted video data, first a matrix $\mathbf{J}$ is created. This matrix, which will be referred to as the *mask*, contains only 1's and 0's. At the location of each uncorrupted pixel $\mathbf{J}(m,n) = 1$, while at the locations of the corrupted pixels $\mathbf{J}$ has a value of 0. The frames of the corrupted or masked video ($\mathbf{X}_c[k]$) can be created by taking the Hadamard product of the frames ($\mathbf{X}[k]$) and the mask (5-1).

$$\mathbf{X}_c[k] = \mathbf{J} \odot \mathbf{X}[k] \tag{5-1}$$

The uncorrupted pixels can be collected into a vector $\mathbf{y}[k]$ by selecting the values of the frame $\mathbf{X}[k]$ for which $\mathbf{J}(m,n) = 1$. The (linear) indices of these pixels are used as input for the Kalman filter ($\mathbf{c}$). The amount of missing pixels will be expressed as a percentage, denoted by β.

5-2 Performance

In this section the performance of the tensor-networked Kalman filter will be evaluated. All computations were performed using MATLAB version R2019b [27]. The computer on which the computations were performed has a Intel Core i7-7700HQ CPU running at 2.80GHz with 8.00 GB of RAM.

5-2-1 Performance Measures

To test the performance of the Kalman filter two different performance measures will be used: the relative error and the peak signal-to-noise ratio (PSNR). Both of these measures are well established in the field of video and image processing [15, 40].

The relative error per frame ($\varepsilon[k]$) is calculated by (5-2). Here, $\mathbf{X}[k]$ denotes the frame at time k and $\widehat{\mathbf{X}}[k]$ the estimate of the frame at time k.

$$\varepsilon[k] = \frac{\left\|\mathbf{X}[k] - \widehat{\mathbf{X}}[k]\right\|_F}{\|\mathbf{X}[k]\|_F} \tag{5-2}$$

The peak signal-to-noise ratio is the ratio between the maximum power of a signal and the power of the corrupting noise [15]. Generally, the PSNR is expressed on a decibel scale. The PSNR is calculated using (5-3), here MSE represents the mean-squared error (MSE) between the original frame and the estimated frame (5-4). A better reconstruction leads to a higher PSNR, since the power of the noise ($\mathbf{X}[k] - \widehat{\mathbf{X}}[k]$) is smaller.

$$PSNR\,[k] = 10\log_{10}\left(\frac{255^2}{MSE}\right) \tag{5-3}$$

$$MSE\,[k] = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}\left(x_{m,n}[k] - \widehat{x}_{m,n}[k]\right)^2 \tag{5-4}$$

Color videos For color videos the calculation of the relative error and the PSNR are analogous. Only now the frames are 3-dimensional tensors instead of matrices: $\mathcal{X}[k] \in \mathbb{R}^{M\times N\times 3}$. This leads to the following expressions for the relative error and the PSNR,

$$\varepsilon[k] = \frac{\left\|\mathcal{X}[k] - \widehat{\mathcal{X}}[k]\right\|_F}{\|\mathcal{X}[k]\|_F} \tag{5-5}$$

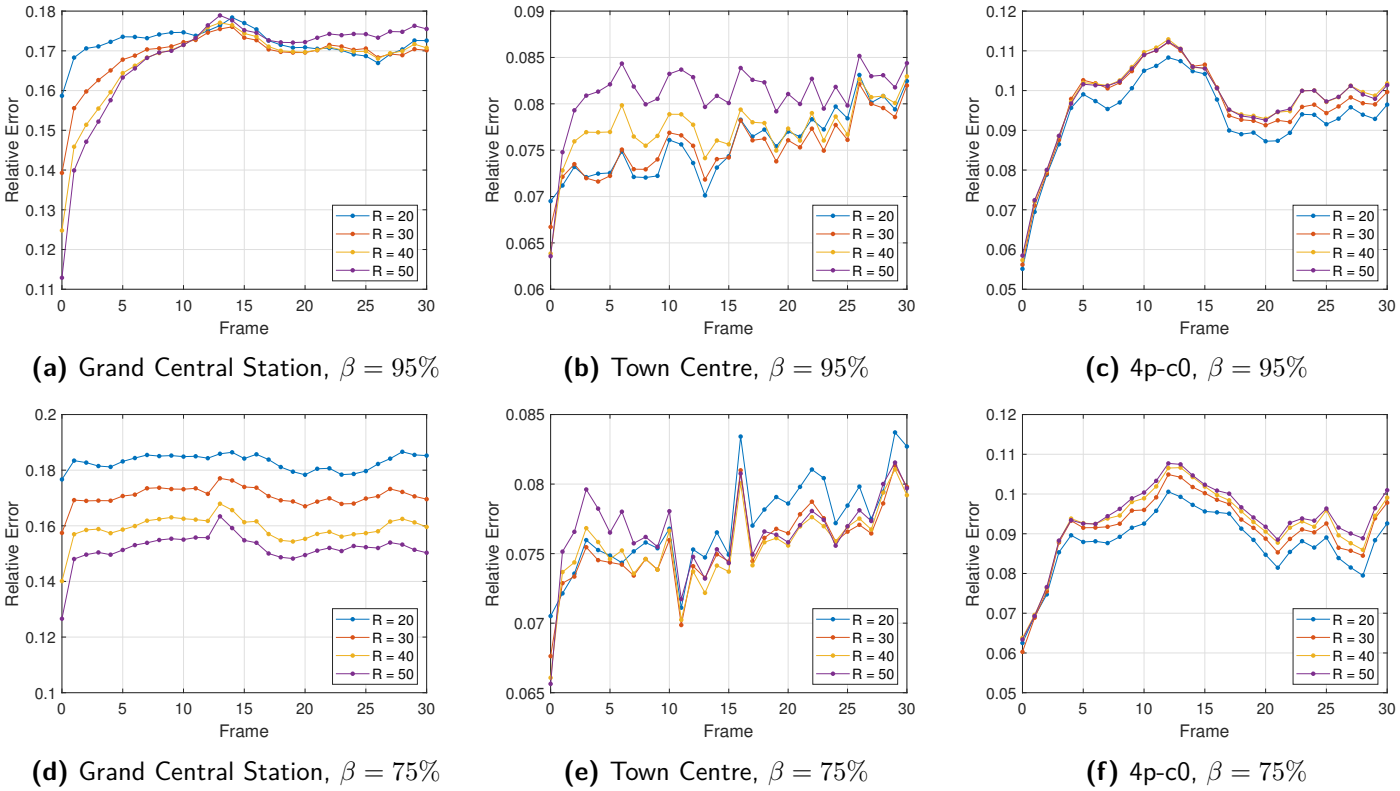
$$PSNR\,[k] = 10\log_{10}\left(\frac{255^2}{MSE}\right),\ MSE\,[k] = \frac{1}{3MN}\sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{i=1}^{3}\left(x_{m,n,i}[k] - \widehat{x}_{m,n,i}[k]\right)^2. \tag{5-6}$$

5-2-2 Influence Rank of $\mathbf{x}$

In this section the influence of the maximum TT-rank of the TT-representation of $\mathbf{x}[k]$ on the Kalman filter estimation will be tested. This is done by evaluating the performance of the Kalman filter for different ranks of $\mathbf{x}[k]$. In Figure 5-3 the relative error of the Kalman filter estimation for four different maximum TT-ranks can be found. And in Figure 5-4 the same can be found for the PSNR of the estimation. The computation times per frame are tabulated in Table 5-2.

What is interesting about these results is that higher ranks do not always lead to better results. Especially in the case that $\beta = 95\%$, the highest rank ($R_x = 50$) actually seems to perform worse over time than the lower ranks ($R_x = 20$ and $R_x = 30$). This result seems to indicate that the compression of the state-vector can actually lead to a better result overall in the case of high percentage of missing pixels. Therefore, in the case that $\beta = 95\%$ it is sufficient to choose relatively low ranks, of $R_x = 20$ and $R_x = 30$.

In the case that less pixels are missing, $\beta = 75\%$, higher ranks can, in some circumstances, lead to a better estimation. This is especially illustrated by the results from the Grand Central



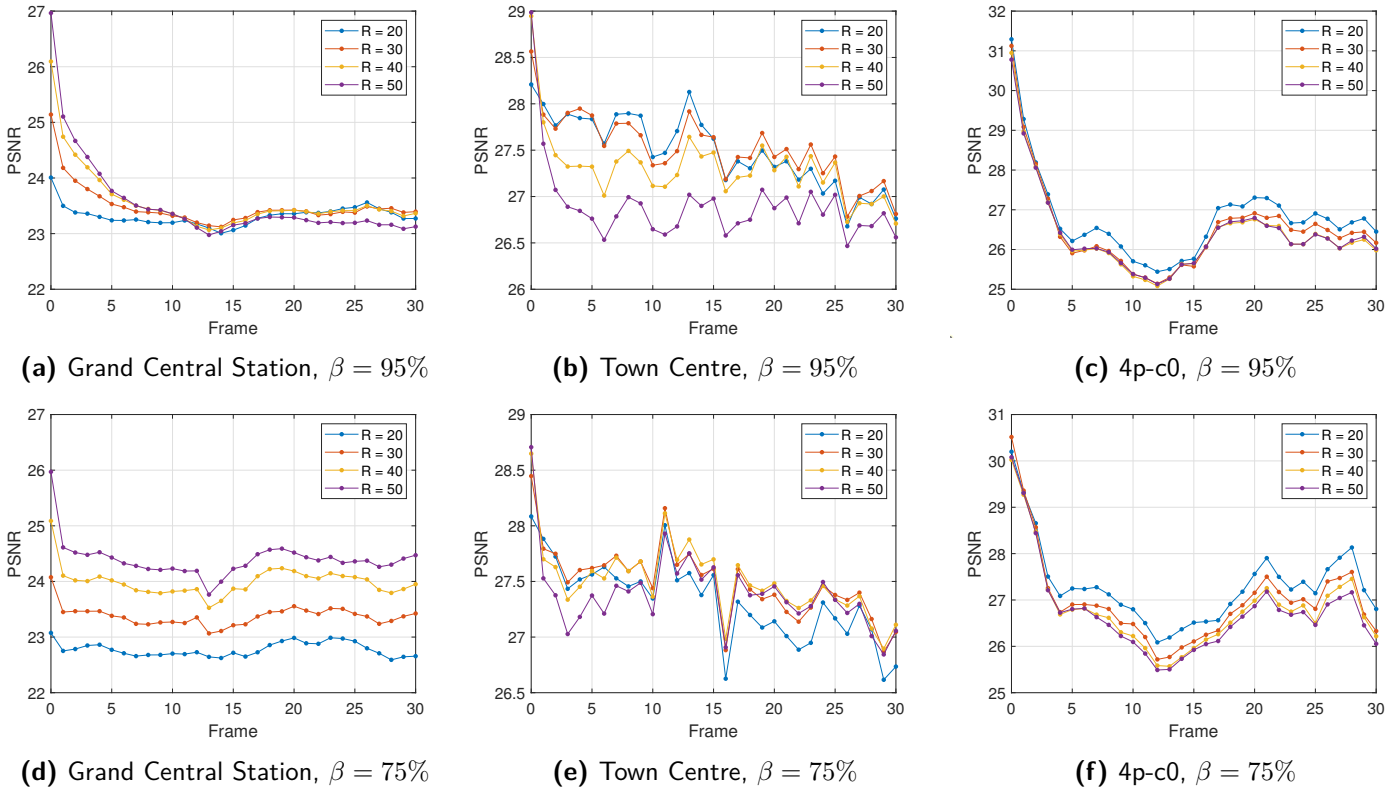
(a) Grand Central Station, $\beta = 95\%$ **(b)** Town Centre, $\beta = 95\%$ **(c)** 4p-c0, $\beta = 95\%$

(d) Grand Central Station, $\beta = 75\%$ **(e)** Town Centre, $\beta = 75\%$ **(f)** 4p-c0, $\beta = 75\%$

Figure 5-3: Relative error of Kalman estimation for different ranks of $\mathbf{x}[k]$. Blue: $R_x = 20$, red: $R_x = 30$, yellow: $R_x = 40$ and purple: $R_x = 50$.

Station video (Figure 5-3d and Figure 5-4d). However, for the 4p-c0 video the opposite seems to be true. Here, the lowest rank approximation of $\mathbf{x}[k]$ actually leads to the best results (Figure 5-3f and Figure 5-4f).

These differing results likely stem from the differences between the two videos. In the Grand Central Station video a lot is happening on the foreground of the video, but the people on the video take up a relatively small part of the frame. Whereas, in the 4p-c0 video only one person is walking through the frame and the camera is relatively close by. Therefore, it seems that a higher rank approximation of $\mathbf{x}[k]$ is beneficial when there is a lot happening on the foreground of the frame and the amount of missing pixels is not extremely high ($\leq 75\%$). Otherwise a lower rank approximation is preferred, as it leads to faster computations (Table 5-2).



(a) Grand Central Station, $\beta = 95\%$ **(b)** Town Centre, $\beta = 95\%$ **(c)** 4p-c0, $\beta = 95\%$

(d) Grand Central Station, $\beta = 75\%$ **(e)** Town Centre, $\beta = 75\%$ **(f)** 4p-c0, $\beta = 75\%$

Figure 5-4: PSNR of Kalman estimation for different ranks of $\mathbf{x}[k]$. Blue: $R_x = 20$, red: $R_x = 30$, yellow: $R_x = 40$ and purple: $R_x = 50$.

5-2-3 Comparison With State-Of-The-Art

The performance of the Kalman filter approach will be compared to that of a state-of-the-art method based on adaptive matrix completion, the PLMS algorithm [40], in this section. In the PLMS algorithm the following parameters were used: $\lambda = 0.8$ and $\mu = 1$. For the Grand Central Station and the Town Centre the TT-ranks of the frame were set to 30, and for the 4p-c0 video they were set to 20. To make a fair comparison, background subtraction was also used before applying the PLMS algorithm.

Ideally, it would also have been possible to compare the Kalman filter method with the method based on streaming tensor completion, however the authors of [26] could not be reached for code implementations of their method.

Speed of Calculations

Unlike the Kalman filter method, the computational speed of the PLMS method is not dependent on the amount of missing pixels. It only depends of the size of the frames. For frames of size 480×720 the average computation time per frame was $0.568\,s$ and for frames of size 288×360 it was $0.175\,s$. This is between $60 - 460$ times faster than the Kalman filter

Video	β	Computation Time (s)			
		$R_x = 20$	$R_x = 30$	$R_x = 40$	$R_x = 50$
Grand Central Station	95%	37	39	45	52
	75%	189	260	343	443
Town Centre	95%	34	38	43	51
	75%	183	253	365	440
4p-c0	95%	11	11	13	14
	75%	54	63	77	95

Table 5-2: Average computation time per frame in seconds.

estimation (Table 5-2). Thus, in terms of computational speed the PLMS algorithm performs better than the Kalman filter.

The reason for this difference in computational speed is as follows. The most computationally expensive step of the PLMS algorithm is the calculation of the singular value decomposition (SVD) of the video frame, which has a computational complexity of $O(MN\min(M,N))$ [42]. For the Kalman filter the most expensive step is the rounding of the TT's (around 40% of the total update), which has a complexity of $\mathcal{O}\left((d_M + d_N)R^3 \max(M_i, N_j)\right)$ (for $i = 1, 2 \ldots, d_M;\ j = 1, 2, \ldots, d_N$).

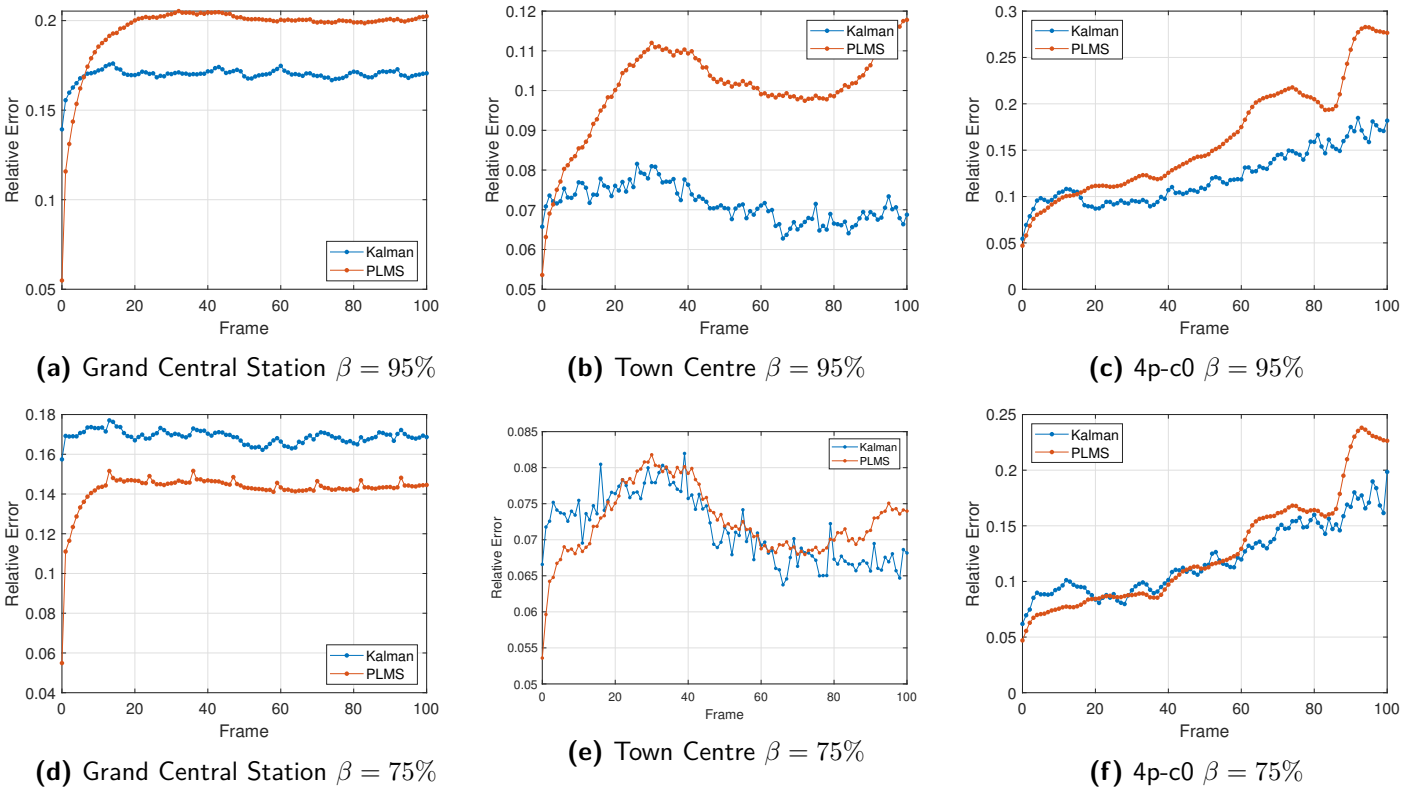
Although the complexity of the rounding step is smaller than that of the SVD, the PLMS algorithm is faster because the SVD function is only called once per frame. In the Kalman filter the rounding function needs to be called many times per frame due to the measurement-wise update, which causes the ranks of the TT's to increase after each measurement update. This is the reason why the Kalman filter filter is over six times slower when $\beta = 75\%$ compared to when $\beta = 95\%$, since with more measurements there is also an increase in the number of calls to the rounding function. In the Town Centre video for instance the rounding function is called 9061 times per frame when $\beta = 95\%$ and 45307 times when $\beta = 75\%$.

Relative Error and PSNR

Clearly, the PLMS algorithm performs faster computations. However, this does come at a cost. In Figure 5-5 and Figure 5-6 the difference in relative error and PSNR between the Kalman filter estimation and the PLMS estimation is shown for all three video. From these figures it can be determined that in the case that $\beta = 95\%$, the Kalman filter performs better than the PLMS algorithm.

This is further verified by looking at one of the estimated frames. In Figure 5-8 the estimated frames using the two different algorithms can be seen of the Grand Central Station video (for other videos see Appendix B-2). Clearly, the PLMS algorithm does not work if $\beta = 95\%$, as the estimated frame frame shows only the background (which is already known). Whereas, the Kalman filter estimate resembles the original (Figure 5-8a) more closely.

In the case that $\beta = 75\%$, the results based on the relative error and the PLMS are less conclusive. In some cases the PLMS method seems to perform better (Grand Central Station video), however, in other this does not necessarily seem to be the case (Town Centre and 4p-c0 video). Therefore, it is necessary to look at the actual estimation to determine which



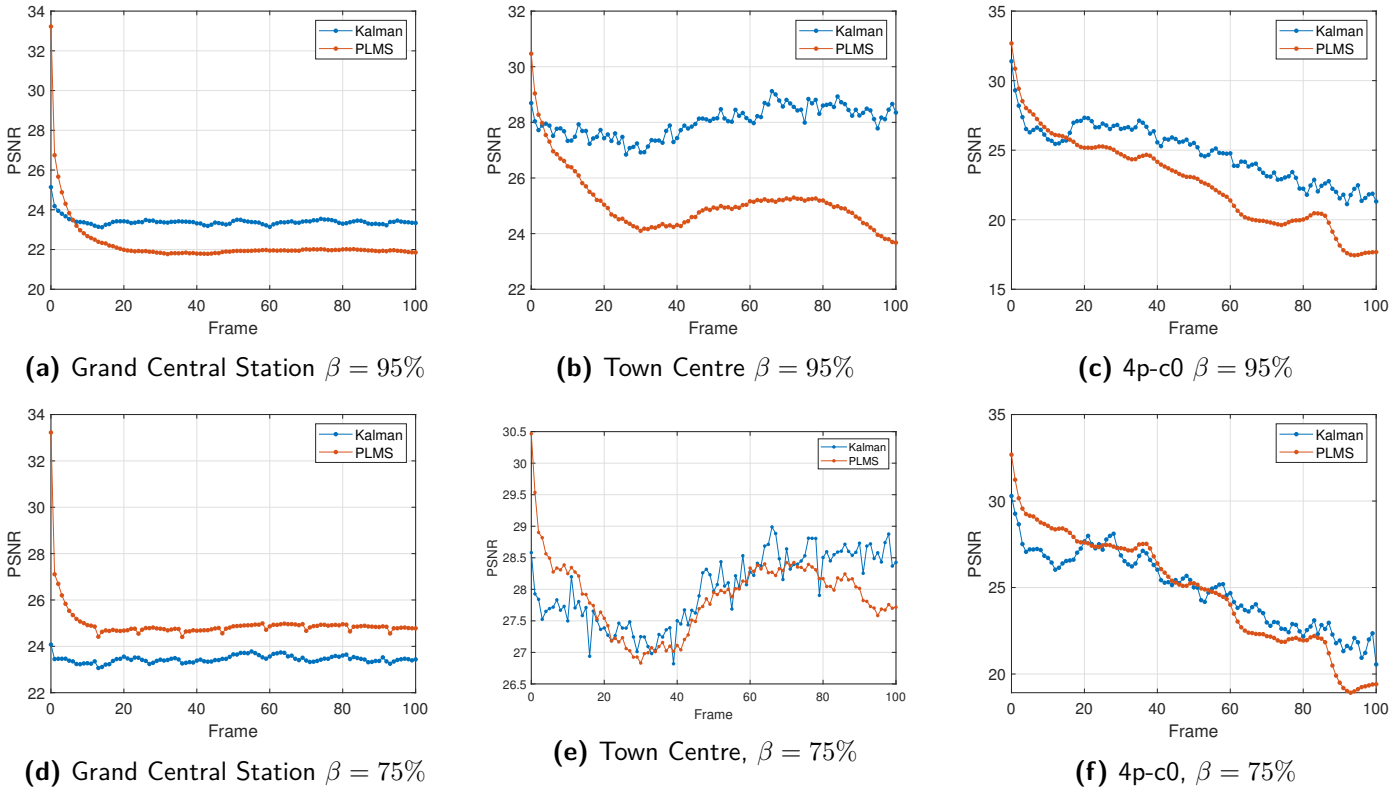
(a) Grand Central Station $\beta = 95\%$ **(b)** Town Centre $\beta = 95\%$ **(c)** 4p-c0 $\beta = 95\%$

(d) Grand Central Station $\beta = 75\%$ **(e)** Town Centre $\beta = 75\%$ **(f)** 4p-c0 $\beta = 75\%$

Figure 5-5: Comparison of relative error of the reconstruction using the Kalman filter (blue) and the PLMS algorithm (red).

method performs better. From Figure 5-8 one can clearly see a difference in the estimation. The Kalman filter estimate seems to contain a lot more noise, and the people in the frame are quite pixelated. In the PLMS estimated frame, there is a lot less noise, however, the people in the video frame seems more like shadow figures.

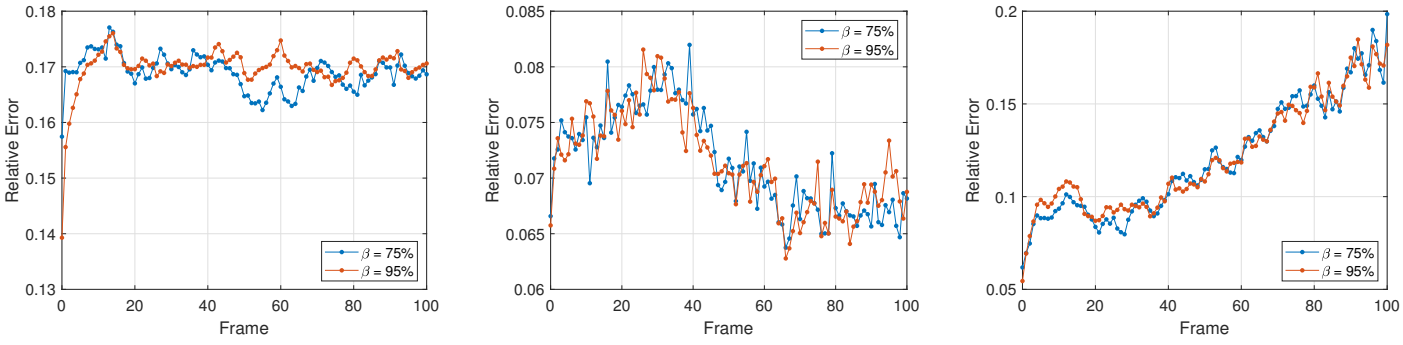
The PLMS estimate does show significant improvement compared to the estimate when $\beta = 95\%$. For the Kalman filter one cannot say the same. This is further verified by Figure 5-7 where the relative errors are plotted in the same figure. The Kalman filter seems to hit a certain barrier in terms of performance. A possible reason for this could be the use of a rank 1 TTm approximation of the state covariance matrix ($\mathbf{P}[k]$), which is far from ideal and likely limits the estimation performance.



(a) Grand Central Station $\beta = 95\%$ **(b)** Town Centre $\beta = 95\%$ **(c)** 4p-c0 $\beta = 95\%$

(d) Grand Central Station $\beta = 75\%$ **(e)** Town Centre, $\beta = 75\%$ **(f)** 4p-c0, $\beta = 75\%$

Figure 5-6: PSNR of the video reconstruction. Kalman filter method (blue) compared to PLMS algorithm (red). Blue: $R_x = 20$, red: $R_x = 30$, yellow: $R_x = 40$ and purple: $R_x = 50$.



(a) Grand Central Station **(b)** Town Centre **(c)** 4p-c0

Figure 5-7: Relative error of Kalman filter estimate for $\beta = 75\%$ (blue) and $\beta = 95\%$ (red).

(a) Original frame

(b) Kalman filter, $\beta = 95\%$

(c) PLMS, $\beta = 95\%$

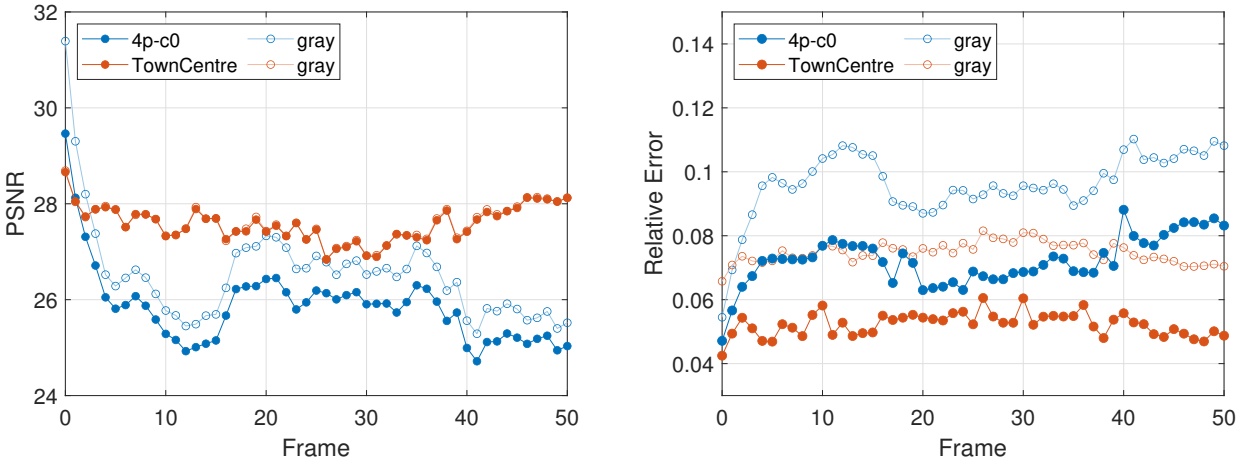
(d) Kalman filter, $\beta = 75\%$

(e) PLMS, $\beta = 75\%$

Figure 5-8: Estimated frames using the Kalman filter algorithm and the PLMS algorithm, Grand Central Station video [47].

5-2-4 Color videos

To showcase that the tensor-networked Kalman filter can also be applied to color videos, in this section the color version of the Town Centre and 4p-c0 video will be reconstructed. In Figure 5-10 a reconstruction of these videos is shown when $\beta = 95\%$. The relative error and and the PSNR are shown in Figure 5-9. The PSNR values are comparable to those of the grayscale videos. While the relative errors are even lower than those of grayscale videos. This is due to the fact that the relative error is sensitive to the size of the data, as you divide by the norm of the original frame (5-5). The relative error does, however, show a similar trend as the relative error of the grayscale video. All in all the results suggest that the estimation of the Kalman filter has a similar performance for color videos as for grayscale videos.



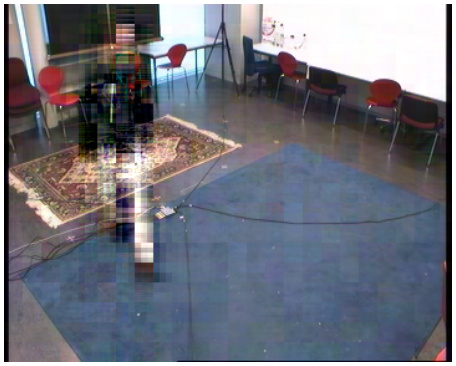
(a) PSNR **(b)** Relative Error

Figure 5-9: PSNR and Relative Error of Kalman filter reconstruction of the color version of the Town Centre (red) and 4p-c0 video (blue). The results of the reconstruction of the grayscaled video (with the same mask) are plotted alongside the results of the color videos (in the same color with 'o' markers).

(a) Town Centre, original frame.

(b) 4p-c0, original frame.

(c) Town Centre, Kalman reconstruction.



(d) 4p-c0. Kalman reconstruction.

Figure 5-10: Kalman filter reconstruction ($\beta = 95\%$).

Chapter 6

Conclusions & Recommendations for Future Research

From the results of the previous chapter it is clear that when the percentage of missing pixels is very high ($> 75\%$), the tensor-networked Kalman filter algorithm performs better in streaming video completion than the state-of-the-art methods based on adaptive matrix completion. When this percentage is lower, the algorithms based on adaptive matrix completion are still preferred, as the Kalman filter estimation is (currently) slower and its performance does not improve much when more pixels are available for estimation.

Though the Kalman filter algorithm does not currently work as well when the percentage of missing pixels is low ($< 75\%$), there are ways in which this could be improved. Both in terms of performance and computational speed. Below some possibilities will be proposed to improve this performance.

Recommendations to speed up calculations

First of all, the tensor-networked Kalman filter is quite slow, it is 60-460 times slower than the PLMS algorithm. The main reason why this is the case is due to the number of calls to the rounding function. The rounding function needs to be called often due to the fact that the Kalman update is performed measurement-by-measurement, to avoid having to compute the matrix inverse in tensor-train matrix (TTm)-format. A possible way to improve the computational speed of the tensor-networked Kalman filter, is to use block-wise updates instead of measurement-wise updates.

In the blockwise update Kalman filter, a 'block' of measurements are updated simultaneously [34]. Instead of scalar inversions of $s_j[k]$ now matrix inversion of smaller block matrices $\mathbf{S}_j[k]$ are used. Where the size each $\mathbf{S}_j[k]$ is equal to the size of the block. The advantage of using these block updates, is that fewer calls to the rounding function are necessary, since there are fewer iterations per update. A possible downside of this approach is that conversion from and to TTm-format are necessary in order to compute the inverse of $\mathbf{S}_j[k]$. The TTm

representation of $\mathbf{S}_j^{-1}[k]$ will likely have relatively high ranks, which could either slow down the calculations or lead to a less accurate estimation (in case of truncated ranks).

While calculating the inverse of a matrix in TTm-format is not yet desired, it is possible that at some point in the future new and improved tensor-train matrix inversion algorithms will be developed. If this is the case, there is also the possibility of foregoing the partitioned update Kalman filter approach and instead calculate the completion using the 'regular' Kalman filter update. This could reduce the number of calls to the rounding function. However, it does introduce a new, possibly time-consuming, calculation.

The rounding of the tensor-trains is the most time-consuming step of the Kalman filter. About 40% of the time is spend on rounding. However, this also means that about 60% of the time is spend on all the other Kalman filter steps: the summation and multiplication of the TT's and TTm's. These steps can be made more efficient by parallelizing them. This parallelization is possible because in both cases (summation and multiplication) each core is treated separately (Section 2-3-1, Appendix A-1-2). The parallelization of these calculations could therefore decrease the computation times by a factor d (equal to the number of cores).

Recommendations to improve performance

The performance of the Kalman filter does not seems to improve as much when the percentage of missing pixels decreases, as the proximal LMS (PLMS) algorithm. One possible reason for this is the use of a rank 1 TTm approximation of the covariance matrix. Unfortunately, this is necessary to ensure that the covariance matrix is positive definite. To be able to use higher ranks approximations, one would therefore have to find a way that 'forces' this tensor-train matrix to be positive definite.

If this is not possible, then one might try a completely different approach to using the Kalman filter for video completion. Instead of using tensor-trains to approximate the Kalman filter variables, it is also possible to use a patch-based approach. In this case the Kalman filter would not compute the reconstruction of the entire frame, but of patches within the frame. Similar to other patch-based video completion methods [11]. Thus, there would be multiple *explicit* Kalman filters working in parallel. Because of the possibility of parallelizing the computations, this could also significantly improve the computational speed. One downside of this method is that not all the information in the spatial domain is utilized. By only estimating small patches in the frame, there could be a lack of spatial coherency on the edges of those patches.

Appendix A

Algorithms

A-1 Tensor-Train Algorithms

A-1-1 Dot Product

Algorithm 5: Dot product[1]. [8, p. 386]

Data: $\mathcal{A} = \langle\!\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with TT-ranks $[R_0, R_1, \ldots, R_d]$,
$\mathcal{B} = \langle\!\langle \mathcal{B}^{(1)}, \mathcal{B}^{(2)}, \ldots, \mathcal{B}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with TT-ranks $[P_0, P_1, \ldots, P_d]$.
Result: $c = \langle \mathcal{A}, \mathcal{B} \rangle$ (dot product).
Initialization: $c = 1$;
for $k = 1, 2, \ldots, d$ **do**
  $\mathbf{Z} = c * \texttt{reshape}(\mathcal{B}^{(k)}, [P_{k-1}, I_k P_k])$
  $c = \texttt{reshape}(\mathcal{A}^{(k)}, [R_{k-1} I_k, R_k])^T * \texttt{reshape}(\mathbf{Z}, [R_{k-1} I_k, P_k])$
end

[1] code: https://gitlab.com/seline/thesis/-/blob/dd9fdd34becb438d8c49366eafc1f9b2cd74003b/Matlab_Code/@TensorTrain/dot.m

A-1-2 Matrix products

Algorithm 6: Matrix-Vector product[2]. [39, p. 9-11]

Data: Matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$ and vector $\mathbf{x} \in \mathbb{R}^{K}$ in TT-format:
$\mathcal{A} = \langle\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \cdots \times I_d \times J_d}$ and
$\mathcal{X} = \langle\langle \mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \ldots, \mathcal{X}^{(d)} \rangle\rangle \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_d}$ with cores $\mathcal{A}^{(n)} \in \mathbb{R}^{P_{n-1} \times I_n \times J_n \times P_n}$ and
$\mathcal{X}^{(n)} \in \mathbb{R}^{R_{n-1} \times J_n \times R_n}$.

Result: Matrix-vector product $\mathbf{y} = \mathbf{A}\,\mathbf{x}$ in TT-format,
$\mathcal{Y} = \langle\langle \mathcal{Y}^{(1)}, \mathcal{Y}^{(2)}, \ldots, \mathcal{Y}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with cores $\mathcal{Y}^{(n)} \in \mathbb{R}^{P_{n-1} R_{n-1} \times I_n \times P_n P_n}$.

for $n = 1, 2, \ldots, d$ **do**
- {Permute cores}
- $\mathcal{A}_p^{(n)} = \texttt{permute}(\mathcal{A}^{(n)}, [1,\ 2,\ 4,\ 3]) \in \mathbb{R}^{P_{n-1} \times I_n \times P_n \times J_n}$
- $\mathcal{X}_p^{(n)} = \texttt{permute}(\mathcal{X}^{(n)}, [2,\ 1,\ 3]) \in \mathbb{R}^{J_n \times R_{n-1} \times R_n}$
- {Matricize}
- $\mathbf{A}_n = \texttt{reshape}(\mathcal{A}_p^{(n)}, [P_{n-1} I_n P_n,\ J_n])$, $\mathbf{X}_n = \texttt{reshape}(\mathcal{X}_p^{(n)}, [J_n, R_{n-1} R_n])$
- {Contract}
- $\mathbf{Y}_n = \mathbf{A}_n\,\mathbf{X}_n \in \mathbb{R}^{P_{n-1} I_n P_n \times R_{n-1} R_n}$
- {Reshape and save core}
- $\mathcal{Y}^{(n)} \in \mathbb{R}^{P_{n-1} R_{n-1} \times I_n \times P_n R_n}$

end

Algorithm 7: Matrix-Matrix product[3]. [39, p. 9-11]

Data: Matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$ and matrix $\mathbf{B} \in \mathbb{R}^{K}$ in TT-format:
$\mathcal{A} = \langle\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \cdots \times I_d \times J_d}$ and
$\mathcal{B} = \langle\langle \mathcal{B}^{(1)}, \mathcal{B}^{(2)}, \ldots, \mathcal{B}^{(d)} \rangle\rangle \in \mathbb{R}^{J_1 \times M_1 \times J_2 \times M_2 \times \cdots \times J_d \times M_d}$ with cores
$\mathcal{A}^{(n)} \in \mathbb{R}^{P_{n-1} \times I_n \times J_n \times P_n}$ and $\mathcal{B}^{(n)} \in \mathbb{R}^{R_{n-1} \times J_n \times M_n \times R_n}$.

Result: Matrix-Matrix product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ in TT-format,
$\mathcal{C} = \langle\langle \mathcal{C}^{(1)}, \mathcal{C}^{(2)}, \ldots, \mathcal{C}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times M_1 \times I_2 \times M_2 \times \cdots \times I_d \times M_d}$ with cores
$\mathcal{Y}^{(n)} \in \mathbb{R}^{P_{n-1} R_{n-1} \times I_n \times P_n P_n}$.

for $n = 1, 2, \ldots, d$ **do**
- {Permute cores}
- $\mathcal{A}_p^{(n)} = \texttt{permute}(\mathcal{A}^{(n)}, [1,\ 2,\ 4,\ 3]) \in \mathbb{R}^{P_{n-1} \times I_n \times P_n \times J_n}$
- $\mathcal{B}_p^{(n)} = \texttt{permute}(\mathcal{B}^{(n)}, [2,\ 1,\ 3,\ 4]) \in \mathbb{R}^{J_n \times R_{n-1} \times M_n \times R_n}$
- {Matricize}
- $\mathbf{A}_n = \texttt{reshape}(\mathcal{A}_p^{(n)}, [P_{n-1} I_n P_n,\ J_n])$, $\mathbf{B}_n = \texttt{reshape}(\mathcal{B}_p^{(n)}, [J_n, R_{n-1} M_n R_n])$
- {Contract}
- $\mathbf{C}_n = \mathbf{A}_n\,\mathbf{B}_n \in \mathbb{R}^{P_{n-1} I_n P_n \times R_{n-1} M_n R_n}$
- {Reshape and save core}
- $\mathcal{C}^{(n)} \in \mathbb{R}^{P_{n-1} R_{n-1} \times I_n \times M_n \times P_n R_n}$

end

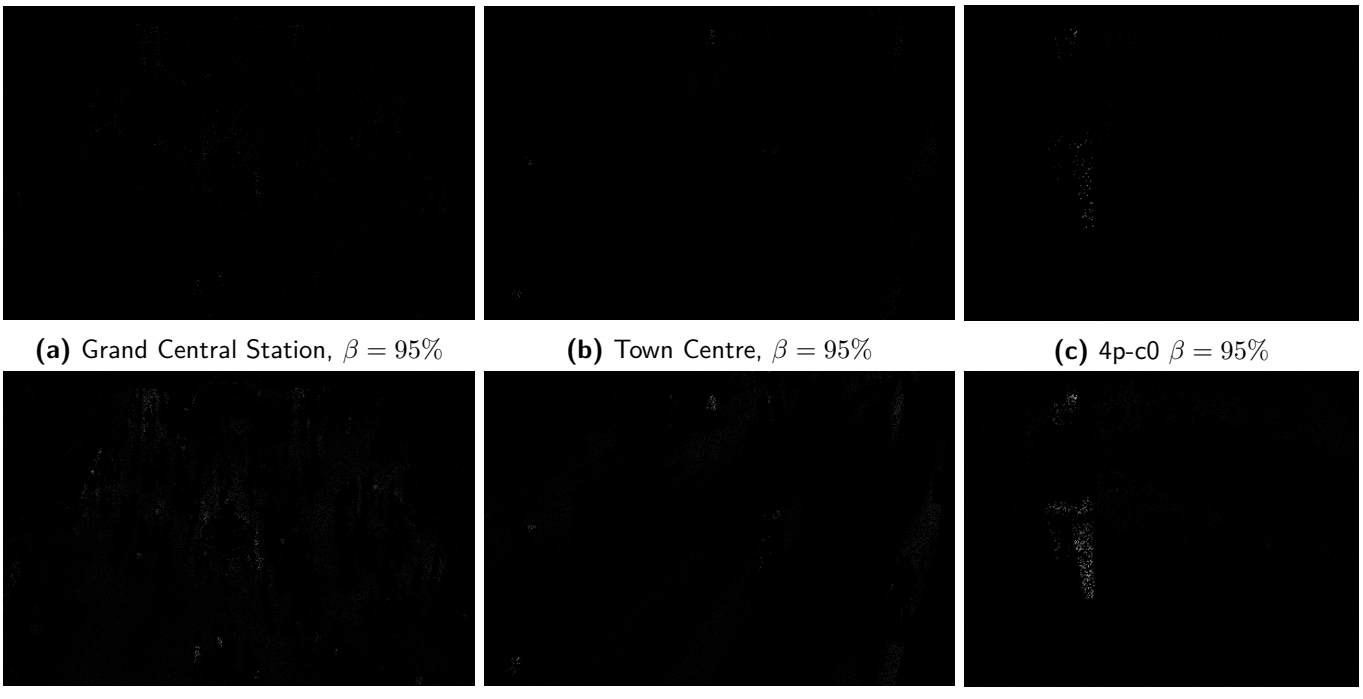
[2] code: https://gitlab.com/seline/thesis/-/blob/a907866468cef6f74655d84bb3255b06f7ff5958/Matlab_Code/TT_Functions/TTm_x_TT.m

[3] code: https://gitlab.com/seline/thesis/-/blob/a907866468cef6f74655d84bb3255b06f7ff5958/Matlab_Code/TT_Functions/TTm_x_TTm.m

Appendix B

Video Frames

B-1 Masked Frames



(a) Grand Central Station, $\beta = 95\%$ **(b)** Town Centre, $\beta = 95\%$ **(c)** 4p-c0 $\beta = 95\%$

(d) Grand Central Station, $\beta = 75\%$ **(e)** Town Centre, $\beta = 75\%$ **(f)** 4p-c0 $\beta = 75\%$

Figure B-1: Masked frames, for $\beta = 95\%$ and $\beta = 75\%$.

B-2 Reconstructed Frames

(a) Original frame

(b) Kalman filter, $\beta = 95\%$

(c) PLMS, $\beta = 95\%$

(d) Kalman filter, $\beta = 75\%$

(e) PLMS, $\beta = 75\%$

Figure B-2: Estimated frames using the Kalman filter algorithm and the PLMS algorithm, 4p-c0 video [9].

(a) Original frame

(b) Kalman filter, $\beta = 95\%$

(c) PLMS, $\beta = 95\%$

(d) Kalman filter, $\beta = 75\%$

(e) PLMS, $\beta = 75\%$

Figure B-3: Estimated frames using the Kalman filter algorithm and the PLMS algorithm, Town Centre video [1].

Bibliography

[1] Ben Benfold and Ian Reid. Stable multi-target tracking in real-time surveillance video. In *CVPR*, pages 3457–3464, June 2011.

[2] J. A. Bengua, H. N. Phien, H. D. Tuan, and M. N. Do. Efficient tensor completion for color image and video recovery: low-rank tensor train. *IEEE Transactions on Image Processing*, 26(5):2466–2479, May 2017.

[3] Emmanuel J. Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772, April 2009.

[4] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, Sep 1970.

[5] Martin Casdagli, Stephen Eubank, J.Doyne Farmer, and John Gibson. State space reconstruction in the presence of noise. *Physica D: Nonlinear Phenomena*, 51(1):52 – 98, 1991.

[6] Pattrawut Chansangiam, Patcharin Hemchote, and Praiboon Pantaragphong. Inequalities for Kronecker products and Hadamard products of positive definite matrices. *ScienceAsia*, 35:106–110, 03 2009.

[7] Yilun Chen, Y. Gu, and A. O. Hero. Sparse LMS for system identification. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3125–3128, April 2009.

[8] Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, and Danilo P. Mandic. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends in Machine Learning*, 9(4-5):249–429, 2016.

[9] F. Fleuret, J. Berclaz, R. Lengagne, and P. Fua. Multicamera people tracking with a probabilistic occupancy map. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):267–282, 2008.

[10] M. S. Grewal and A. P. Andrews. Applications of Kalman filtering in aerospace 1960 to the present [historical perspectives]. *IEEE Control Systems Magazine*, 30(3):69–78, Jun. 2010.

[11] Q. Guo, S. Gao, X. Zhang, Y. Yin, and C. Zhang. Patch-based image inpainting via two-stage low rank approximation. *IEEE Transactions on Visualization and Computer Graphics*, 24(6):2023–2036, June 2018.

[12] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.

[13] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, USA, 2nd edition, 2012.

[14] Y. Hu, D. Zhang, J. Ye, X. Li, and X. He. Fast and accurate matrix completion via truncated nuclear norm regularization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(9):2117–2130, Sep 2013.

[15] Quan Huynh-Thu and Mohammed Ghanbari. The accuracy of PSNR in predicting video quality for different video scenes and frame rates. *Telecommun. Syst.*, 49(1):35âĂŞ48, Jan 2012.

[16] S. Ilan and A. Shamir. A survey on data-driven video completion. *Computer Graphics Forum*, 34(6):60–85, 2015.

[17] H. Ji, C. Liu, Z. Shen, and Y. Xu. Robust video denoising using low rank matrix completion. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1791–1798, Jun 2010.

[18] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, Mar 1960.

[19] Boris N. Khoromskij. $\mathcal{O}(d \log n)$-Quantics approximation of N-d tensors in high-dimensional numerical modeling. *Constructive Approximation*, 34(2):257–280, Oct 2011.

[20] M. Kim, D. Park, D. K. Han, and H. Ko. A novel approach for denoising and enhancement of extremely low-light video. *IEEE Transactions on Consumer Electronics*, 61(1):72–80, Feb 2015.

[21] Young-Real Kim, Seung-Ki Sul, and Min-Ho Park. Speed sensorless vector control of induction motor using extended Kalman filter. *IEEE Transactions on Industry Applications*, 30(5):1225–1233, Sep. 1994.

[22] Youngjoo Kim and Hyochoong Bang. *Introduction to Kalman Filter and Its Applications*, chapter 2. IntechOpen, Rijeka, 2019.

[23] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, Sep 2009.

[24] Willem Laurenszoon van Doorn, Gijs Groote, Aniek Hiemstra, Thijs Veen, and Maarten ten Voorde. Reconstruction of faulty video data using the Kalman filter and tensor networks. unpublished, 2019.

[25] J. Liu, P. Musialski, P. Wonka, and J. Ye. Tensor completion for estimating missing values in visual data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1):208–220, Jan 2013.

[26] M. Mardani, G. Mateos, and G. B. Giannakis. Subspace learning and imputation for streaming big data matrices and tensors. *IEEE Transactions on Signal Processing*, 63(10):2663–2677, May 2015.

[27] MATLAB. *version 9.7 (R2019b)*. The MathWorks Inc., Natick, Massachusetts, 2019.

[28] I. V. Oseledets. Approximation of $2^d \times 2^d$ matrices using tensor decomposition. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2130–2145, 2010.

[29] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[30] I. V. Oseledets and S. V. Dolgov. Solution of linear systems and matrix inversion in the tt-format. *SIAM Journal on Scientific Computing*, 34(5):A2718–A2739, 2012.

[31] Roger Penrose. Applications of negative dimensional tensors. *Combinatorial Mathematics and its Applications*, pages 221–244, 1971.

[32] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition, 2003.

[33] François-Éric Racicot and Raymond Théoret. Forecasting stochastic volatility using the Kalman filter: An application to Canadian interest rates and price-earnings ratio. *Aestimatio, the IEB International Journal of Finance*, 1:28–47, 12 2010.

[34] M. Raitoharju and R. Piché. On computational complexity reduction methods for Kalman filter extensions. *IEEE Aerospace and Electronic Systems Magazine*, 34(10):2–19, 2019.

[35] Benjamin. Recht, Maryam. Fazel, and Pablo A. Parrilo. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. *SIAM Review*, 52(3):471–501, 2010.

[36] Simo Särkkä. *Bayesian Filtering and Smoothing*. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2013.

[37] Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. Tensor completion algorithms in big data analytics. *ACM Transactions on Knowledge Discovery from Data*, 13(1):1–48, Jan 2019.

[38] Robert F. Stengel. *Stochastic Optimal Control: Theory and Application*. John Wiley & Sons, Inc., USA, 1986.

[39] E M Stoudenmire and Steven R White. Minimally entangled typical thermal state algorithms. *New Journal of Physics*, 12(5):055026, May 2010.

[40] R. Tripathi, B. Mohan, and K. Rajawat. Adaptive low-rank matrix completion. *IEEE Transactions on Signal Processing*, 65(14):3603–3616, Jul 2017.

[41] Ledyard R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, Sep 1966.

[42] Vinita Vasudevan and M. Ramakrishna. A hierarchical singular value decomposition algorithm for low rank matrices, 2017.

[43] Michel Verhaegen and Vincent Verdult. *Filtering and System Identification: A Least Squares Approach*. Cambridge University Press, 2007.

[44] Greg Welch and Gary Bishop. An introduction to the Kalman filter. Technical Report 95-041, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[45] John W. Woods. Chapter 11 - Digital Video Processing. In John W. Woods, editor, *Multidimensional Signal, Image, and Video Processing and Coding (Second Edition)*, pages 415 – 466. Academic Press, Boston, second edition edition, 2012.

[46] Yangyang Xu, Ruru Hao, Wotao Yin, and Zhixun Su. Parallel matrix factorization for low-rank tensor completion. *Inverse Problems and Imaging*, 9(2):601–624, Mar 2015.

[47] Bolei Zhou, Xiaogang Wang, and Xiaoou Tang. Understanding collective crowd behaviors: Learning a mixture model of dynamic pedestrian-agents. *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2871–2878, 2012.

Glossary

List of Acronyms

TTm	tensor-train matrix
TT	tensor-train
SVD	singular value decomposition
LMS	least mean square
MSE	mean-squared error
PLMS	proximal LMS
SVD	singular value decomposition
CP	CANDECOMP/PARAFAC
fps	frames per second
SGD	stochastic gradient descent
TeOSGD	online SGD algorithm for tensor decomposition and imputation
BEP	bachelor end project
QTT	quantized tensor-train
PUKF	partitioned update Kalman filter
LTI	linear time-invariant
PSNR	peak signal-to-noise ratio

List of Symbols

$\mathbf{A}^{-1}$	Matrix inverse
$\mathbf{A}$	Matrix
$\|\cdot\|_\star$	Nuclear norm
$\|\cdot\|_F$	Frobenius norm
$\odot$	Hadamard product
$\otimes$	Kronecker product
$\mathcal{A}$	Tensor
$\mathbf{A}^T$	Matrix transpose
$\mathbf{a}$	Vector

Index