Improving the Computational Speed of a Tensor-Networked Kalman Filter for Streaming Video Completion

A.P. van Koppen

Master of Science Thesis

Delft University of Technology

Delft Center for Systems and Control

Improving the Computational Speed of a Tensor-Networked Kalman Filter for Streaming Video Completion

Master of Science Thesis

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

A.P. van Koppen

July 18, 2022

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

Abstract

Streaming video completion is the practice that aims to fill in missing or corrupted pixels in a video stream by using past uncorrupted data. A method to tackle this problem is recently introduced called a Tensor Networked Kalman Filter (TNKF). It shows promising results in terms of performance compared to state-of-the-art methods for high percentages of missing pixels ($\geq 95\%$). The main drawback of using a TNKF is the computational speed, which needs to be improved to compete with other existing methods and to be carried out in real-time by a regular computer. This work discusses three methods that reduce the computational load of the algorithm, which speeds up computations. The first method is replacing the existing algorithm with a Block Update TNKF. Secondly, the use of randomized rounding instead of deterministic rounding is investigated. The last method is the simplification of the TNKF update. Results that are presented in this report show that significant speedups of up to +132% can be achieved. In most situations, the considered speedup methods compromise the reconstruction's accuracy. This thesis discusses the effects this has on the quality of the reconstruction.

Table of Contents

List of Figures

List of Tables

Preface & Acknowledgements

This document is a part of my graduation thesis of the programme of MSc Systems & Control. This subject was proposed by my first thesis supervisor, and expands on research done by W. Laurenszoon et al. [11], S.J.S. de Rooij [30] and P. van Klaveren [18] in this field. All code written for this research can be found in the following Git repository: https://gitlab.com/bramvankoppen/thesis. I want to thank my supervisors dr. ir. K. Batselier and ir. C. Menzen for their continuous assistance and guidance during this thesis research.

Chapter 1

Introduction

1-1 Streaming video completion

Due to a variety of reasons, video feeds can encounter corruption or blockage of the pixels in their frames. In the case of surveillance videos, (parts of) the frame can be obstructed due to intentional or unintentional hardware failure. *Streaming video completion* is the practice that aims to fill in missing or corrupted pixels in a video stream as accurately as possible by using past uncorrupted data. This practice distinguishes itself from offline methods, in which all (uncorrupted) frames of the video data set can be used, including 'future' frames. This raises a challenge in optimizing the usage of the (limited) available data, as only past frames and present uncorrupted pixels are available for this online method. Furthermore, the reconstruction must be carried out in real-time. This allows the observer of the video to draw conclusions on the contents of the reconstructed frames. This real-time property of the problem calls for the completion method to have a sufficient computational speed. In Figure 1-1, an example is given of a corrupted frame in a surveillance video. Although the present frame contains almost no information, the past frames still have information about the position and movement of each object. Together with the measured pixels in the present frame, streaming video completion methods can perform a reconstruction.

Aside from surveillance videos, streaming video data exists in many other applications. One of these applications is camera data from intelligent vehicles. Since the computers in these vehicles must make decisions based on real-time video data that comes from multiple cameras on the car, corruption or blockage of data of one or more of these cameras could become an issue. There are different methods to tackle this problem, such as video inpainting [35]. This method combines spatial information of images from different perspectives to assist and constrain the repair of damaged frames in the video. This differs from the surveillance video application since multiple perspectives are used.

A single fixed camera allows for *background subtraction*, enabling a more efficient video data representation. For this, a low-rank Tensor-Train (TT) structure can be used, which will be further elaborated in Chapter 2 and 3. Background subtraction furthermore provides a better

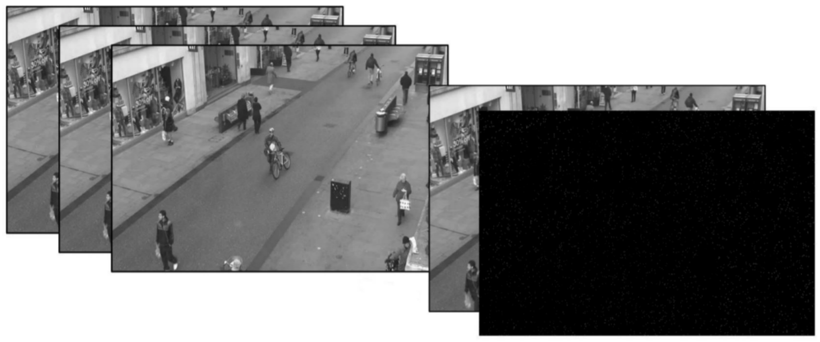
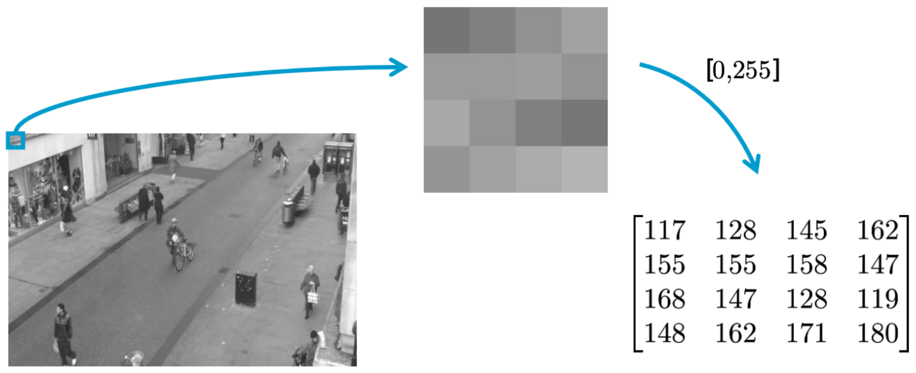


Figure 1-1: Example of a corrupted frame in a surveillance video [30, 4].

estimate of the frame, as now only the moving 'foreground' is reconstructed. Estimating the background of a video is a rather computationally expensive process as the mean of each pixel must be computed over a sequence of frames long enough to exclude all moving objects. This process only needs to be carried out once when the camera is installed, as a surveillance camera is fixed.

A video is a sequence of images or frames that can be displayed at a specific speed or frame rate. This frame rate is often expressed in the number of frames per second (fps) and usually ranges between 24 to 60 fps. Each video frame consists of $M \times N$ pixels, where M is the height, and N is the width of the frame in pixels. The numbers M and N determine the resolution of a video. An High-Definition (HD) video has a resolution of 1080×1920 pixels, for example. For a greyscale video, the color of a pixel can be represented by an 8-bit integer value between 0 and 255. The higher the number, the lower the pixel's intensity (0 for black, 255 for white). This is illustrated in Figure 1-2, where a group of $M \times N$ pixels is represented by a matrix $\mathbf{X} \in \mathbb{R}^{M \times N}$. This greyscale video with a T amount of frames can thus be expressed as a sequence of matrices: $\{\mathbf{X}[1], \mathbf{X}[2], \ldots, \mathbf{X}[T]\}$. In a color video, the frames are represented by three color components: red, green and blue (RGB) with their corresponding intensity value.



$$\begin{bmatrix} 117 & 128 & 145 & 162 \\ 155 & 155 & 158 & 147 \\ 168 & 147 & 128 & 119 \\ 148 & 162 & 171 & 180 \end{bmatrix}$$

Figure 1-2: Matrix representation of (part of) a greyscale video frame [30, 4].

1-2 State-of-the-Art

1-2-1 Adaptive matrix completion

One of the methods to solve the streaming video completion problem is called *adaptive matrix completion*. In matrix completion, an attempt is made to recover a matrix from a small subset of its entries. For low-rank matrices, this problem is reduced to a rank minimization problem. Because of non-convexity of the rank function and a computational complexity of NP-hard of the problem, the rank function is often replaced by the nuclear norm $\|\mathbf{X}\|_\star = \sum_i \sigma_i(\mathbf{X})$. Here $\sigma(\mathbf{X})$ are the singular values of the completed matrix $\mathbf{X}$. This results in the optimization problem in (1-1).

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

where $\mathbf{M} \in \mathbb{R}^{M\times N}$ is a low-rank matrix that is observed over a subset $\mathbf{\Omega}$ of its entries and $\mathbf{X}$ is the completed matrix. The goal of the method is to complete the sequence of low-rank matrices $\{\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \ldots, \mathbf{X}^{(T)}\}$ to reconstruct the corrupted video frames.

In [32], the Proximal Least Mean Squares (PLMS) algorithm is presented, which solves the matrix completion problem. This is a variation on the least mean squares (LMS) algorithm [6], which solves the streaming video completion problem well for relatively low percentages of missing pixels (<50%). In [30] it is shown that for high percentages of missing pixels (>75%), the PLMS algorithm does not succeed in effectively reconstructing the video frames.

1-2-2 Tensor-Networked Kalman Filter

Recently, a new solution to the streaming video completion problem has been introduced: a Tensor-Networked Kalman Filter (TNKF) [11, 30]. This algorithm exploits the structure of low-rank Tensor-Trains to represent the high dimensional video data efficiently. Since a video can be seen as a discrete sequence of states (frames) that can be expressed in a state-space model, the TNKF can use information in previous frames to estimate the corrupted pixels in the current frame. Using tensor networks in the Kalman filter operations of each time step ensures a memory efficient-way of computation. In [30] it is shown that the TNKF shows promising results in the streaming video completion problem for high percentages of missing or corrupted pixels (>75%), especially compared to the PLMS algorithm.

The goal of streaming video completion is to reconstruct frames in real-time, which makes it necessary for the algorithm to have a sufficiently high computational speed. Despite its promising performance for high percentages of missing pixels, the TNKF is 60-460 times slower than the PLMS algorithm in terms of computational speed [30]. Throughout this thesis, the algorithm's performance refers to the accuracy of the reconstructed frames. The surveillance camera industry has an average frame rate of 15 fps [15], and this average has been increasing quickly over the last years. In the case of a frame rate of 15 fps, the algorithm must be able to reconstruct a frame in about 0.06 seconds to be able to complete the video in real-time. In [30] it is shown that using a regular laptop computer, the PLMS algorithm took an average computation time of $0.568s$ per frame. This illustrates that given its vastly lower speed, the TNKF needs to be improved in terms of computational efficiency to reach the goal of real-time video reconstruction with a regular computer.

Aside from enabling real-time reconstruction with fewer requirements on the used hardware, reducing the computational load of the algorithm has other benefits. The energy usage from data centers has increased exponentially in recent years [14]. The reason for this is the increased digitalisation of the economy, which leads to a dynamic increase in the amount of data stored and processed in data centers. With rising energy prices and rising carbon emissions into the atmosphere, the use of more efficient algorithms is of increased importance. Reducing the computational load of algorithms reduces costs the carbon footprint of the considered process.

1-3 Research focus

In this research, an attempt is made to improve the efficiency of the TNKF algorithm by reducing the computational load of the algorithm, which speeds up operations. This is done by investigating the following three speedup methods.

1. *Block Update TNKF*, which aims to reduce the amount of Kalman filter update iterations.
2. *Randomized rounding algorithms*, which aim to reduce the computational load of the most expensive operation in the TNKF algorithm: the rounding procedure.
3. *Simplification of the TNKF update*, which aims to reduce the computational load of the TNKF update equations.

In Table 1-1, all research questions that will be addressed in this research are listed along with the corresponding section(s) of the results and discussion (Chapter 5). In the last section of each method, the achievable speedups and differences in performance are investigated. In the other sections, parameters are examined that influence this speedup. Furthermore, other consequences are looked into that occur when using a specific speedup method.

1-4 Chapter outline and basic notation

In Chapter 2, tensors are introduced, and the concept of tensor networks is elaborated. Subsequently, in Chapter 3, the Tensor-Networked Kalman Filter is introduced, and all operations that influence the computational speed are discussed. In Chapter 4, the three previously mentioned approaches are presented that speed up TNKF operations by reducing the computational load of the algorithm. In Chapter 5, these methods are analysed and discussed in terms of their performance and speed. Lastly, in Chapter 6 conclusions are drawn on the achieved speedups, and their trade-offs with performance. Furthermore, recommendations for future research are given in this last chapter. In Table 1-2, the basic notation is listed that will be used throughout this thesis work.

Section	**Research question**
5-2-1 - 5-2-3	1. *"What is the relation between the parameters of the Block Update TNKF and its computational speed, and how can the values of these parameters be best chosen?"*
5-2-4	2. *"What speedups can be achieved by replacing the Element Update TNKF with a Block Update TNKF, and what is the difference in performance?"*
5-3-1 - 5-3-5	3. *"What effects do randomized rounding of the state and covariance have on the performance and speed of the TNKF, and what parameters influence these effects?"*
5-3-6	4. *"What speedups can be achieved by using randomized rounding compared to deterministic rounding, and what is the difference in performance?"*
5-4-1	5. *"To what extent do the TT(m) cores of the state and covariance change through TNKF operations?"*
5-4-2	6. *"What speedups can be achieved by skipping the covariance update, and what is the corresponding loss in performance?"*

Table 1-1: Addressed research questions per section.

Notation	**Definition**
a or A	Scalar
$\mathbf{a}$	Vector
$\mathbf{A}$	Matrix
$\boldsymbol{\mathcal{A}}$	Tensor
$\mathbf{a}(i)$	i-th entry of a vector a
$\mathbf{A}(i,j)$	Element (i,j) of a matrix $\mathbf{A}$
$\boldsymbol{\mathcal{A}}(i_1,i_2,\dots,i_d) = a_{i_1,i_2,\dots,i_d}$	Element $(i_1,i_2,\dots,i_d)$ of a d-dimensional tensor $\boldsymbol{\mathcal{A}}$
$\mathbf{A}^{(n)}$	n-th matrix in a sequence of matrices $\{\mathbf{A}^{(1)},\mathbf{A}^{(2)},\dots,\mathbf{A}^{(N)}\}$
$\mathbf{A}^{-1}$	Matrix inverse
$\mathbf{A}^T$	Matrix transpose
$\langle\!\langle \boldsymbol{\mathcal{A}}^{(1)},\boldsymbol{\mathcal{A}}^{(2)},\dots,\boldsymbol{\mathcal{A}}^{(d)} \rangle\!\rangle$	Set of cores of d-dimensional tensor $\boldsymbol{\mathcal{A}}$ in Tensor Train format
$\lVert\cdot\rVert_F$	Frobenius norm
$\lVert\cdot\rVert_\star$	Nuclear norm
$\otimes$	Kronecker product
$\odot$	Khatri-Rao product
$\ast$	Hadamard product

Table 1-2: Basic notation.

Chapter 2

Tensor Networks

This chapter gives insight into tensor networks and the operations that can be done with data represented by tensor networks. Parts of this chapter are based on the literature study paper of this thesis [21].

2-1 Tensors and definitions

A *tensor* is a multidimensional array that can have a number of d dimensions. Such a d-dimensional or d-way tensor can be expressed as $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$. A first-order tensor is a vector; a second-order tensor is a matrix, and tensors of order three or higher are called higher-order tensors. Fibers of a tensor are defined by fixing every index but one, as depicted in Figure 2-1. In the case of a matrix, the mode-1 fibers are the columns, and the mode-2 fibers are the rows. Slices of a tensor are defined by fixing all but two indices, giving two-dimensional sections. Figure 2-2 shows the horizontal, lateral and frontal slices of a 3-dimensional tensor.

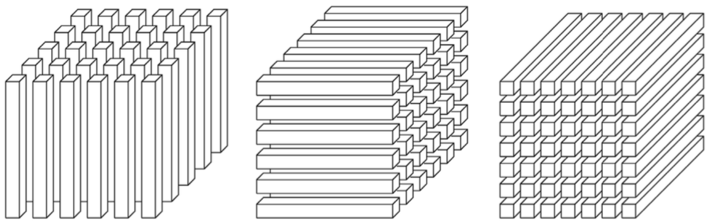


Figure 2-1: The mode-n fibers of a 3-dimensional tensor [20]. From left to right: the mode-1 or column fibers, the mode-2 or row fibers and mode-3 or tube fibers.

Different tensor operations used thoughout this paper are given in Definition 2-1.2 - 2-1.6. For a more extensive derivation of these operations the reader can consult [7, 17, 20].

Definition 2-1.1 (multi-index [7, p. 25])**.** A multi-index $i = \overline{i_1 i_2 \cdots i_d}$ refers to an index which takes all possible combinations of values of indices $i_1, i_2, \ldots, i_d$ in a specific order. For the

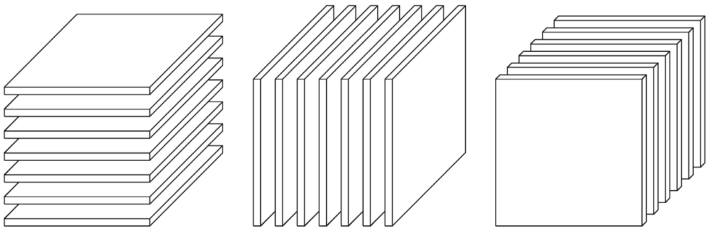


Figure 2-2: The slices of a 3-dimensional tensor [20]. From left to right: horizontal slices, lateral slices and frontal slices.

remainder of this paper the 'little-endian' ordering convention will be used

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

where $i_1, i_2, \dots, i_d$ represents the index along each mode and $I_1, I_2, \dots, I_d$ represents the size of these modes. It must be noted that this formula only works if the indices start counting from i_1.

Definition 2-1.2 (quantization [17])**.** Tensor quantization is the process of transforming lower-dimensional data into higher dimensional tensors with a relatively small mode size (typically 2,3 or 4). In (2-2), an example is given for the quantization of a matrix $\mathbf{A} \in \mathbb{R}^{M \times N}$ into a high dimensional tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d \times J_1 \times J_2 \times \cdots \times J_d}$, where $M = \prod_{k=1}^{d} I_k$ and $N = \prod_{k=1}^{d} J_k$, using the `reshape` function in MATLAB.

$$\boldsymbol{\mathcal{A}} = \texttt{reshape}(\mathbf{A}, I_1, I_2, \dots, I_d, J_1, J_2, \dots, J_d) \tag{2-2}$$

Definition 2-1.3 (mode-n matricization [20, p. 459])**.** A d-way tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ can be unfolded along each mode $n = 1, 2, \dots, d$ resulting in the mode-n matricization, which is a matrix of the form

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

In MATLAB, the `permute` and `reshape` functions can be used to compute the mode-n matricization.

Definition 2-1.4 (n-mode product [20, p. 460])**.** A d-way tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ and a matrix $\mathbf{B} \in \mathbb{R}^{J \times I_n}$ can be multiplied using an n-mode product,

$$\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}} \times_n \mathbf{B} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_d} \tag{2-4}$$

where the elements are defined as

$$\boldsymbol{\mathcal{C}}(i_1, \dots, i_{n-1}, j, i_{n+1}, \dots, i_d) = \sum_{i_n=1}^{I_n} \boldsymbol{\mathcal{A}}(i_1, i_2, \dots, i_d)\, \mathbf{B}(j, i_n) \tag{2-5}$$

Definition 2-1.5 (Tensor norm [20, p. 457])**.** The Frobenius norm of a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is the square root of the sum of the squares of all its elements, i.e.,

$$\|\boldsymbol{\mathcal{A}}\|_F = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_d=1}^{I_d} a_{i_1, i_2, \dots, i_d}^2} \tag{2-6}$$

Definition 2-1.6 (Kronecker product of tensors [7, p. 29])**.** The (right) Kronecker product of two tensors $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ and $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_d}$ results in a tensor $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_1 J_1 \times I_2 J_2 \times \cdots \times I_d J_d}$. The elements of this tensor are defined as

$$\boldsymbol{\mathcal{C}}\left(\overline{j_1 i_1}, \ldots, \overline{j_d i_d}\right) = \boldsymbol{\mathcal{A}}\left(i_1, \ldots, i_d\right) \boldsymbol{\mathcal{B}}\left(j_1, \ldots, j_d\right. \tag{2-7}$$

Definition 2-1.7 (Khatri-Rao product of matrices [16])**.** The Khatri-rao matrix product of two matrices with equal column dimension $\mathbf{X} \in \mathbb{R}^{I \times K}$ and $\mathbf{Y} \in \mathbb{R}^{J \times K}$ is defined as

$$\mathbf{X} \odot \mathbf{Y} = \left[\begin{array}{cccc} \mathbf{x}_1 \otimes \mathbf{y}_1 & \mathbf{x}_2 \otimes \mathbf{y}_2 & \ldots & \mathbf{x}_d \otimes \mathbf{y}_d \end{array}\right] \tag{2-8}$$

where each column of the resulting matrix is a Kronecker product of the columns of both original matrices.

2-2 Tensor diagrams

High dimensional tensors are difficult to visualise. To facilitate understanding of these tensors and enable implementation of their operations, tensor diagrams can be used. These diagrams are a graphical way of illustrating tensors and their operations invented by Roger Penrose [26]. Using nodes to represent the tensor *cores* and edges to represent its dimensions or *modes*, tensors can be visualised. Figure 2-3 shows some simple examples of these diagrams for low-dimensional arrays.

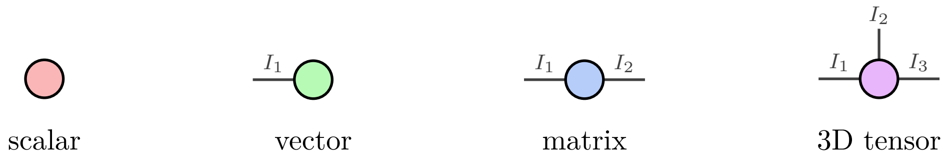


Figure 2-3: Tensor diagrams for different tensor orders.

Throughout this paper, the dimensions are counted clockwise as demonstrated in the 3D tensor diagram in Figure 2-3. Tensor products can also be visualised using tensor diagrams by interconnecting two (or more) cores with their edges. An interconnection between two cores represents a contraction, which is a summation over a particular index (e.g. (2-7)). In Figure 2-4, three simple examples of tensor products are displayed. The vector dot product is a summation of the products of each vector element $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{I_1} \mathbf{a}(i)\mathbf{b}(i)$ where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{I_1}$, and its tensor diagram are two vector cores interconnected with their dimension I_1. A matrix-vector product can mathematically be expressed as $\mathbf{A}\mathbf{b} = \sum_{j=1}^{I_2} \mathbf{A}(:, j)\mathbf{b}(j)$ where $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}, \mathbf{b} \in \mathbb{R}^{I_2}$, the corresponding tensor diagram consists of a matrix and a vector interconnected along their mutual dimension I_2. Lastly, the 3-mode product of a 3-way tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and a matrix $\mathbf{B} \in \mathbb{R}^{I_3 \times I_4}$ is shown, where the two cores are interconnected along the dimension I_3. The mathematical description for this product is given by $\boldsymbol{\mathcal{A}} \times_3 \mathbf{B} = \sum_{k=1}^{I_3} \boldsymbol{\mathcal{A}}(:, :, k)\mathbf{B}(l, k)$.

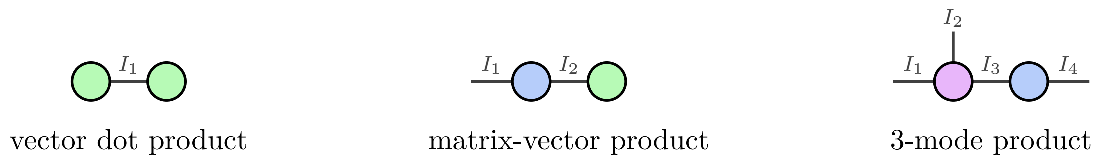


Figure 2-4: Tensor diagrams for simple tensor products.

2-3 Tensor-Train decomposition

The number of elements in a tensor and the storage consumption grow exponentially for increasing tensor order d, and numerical methods cannot efficiently handle problems with these high dimensional tensors. This phenomenon is known as the *curse of dimensionality*, and in many problems finding a solution to this remains a challenge. One of the most promising tensor representations is the Tensor-Train (TT) format, which is a tensor product format originally proposed in quantum physics where it is also known as matrix product states (MPS) [12]. The TT format, as given in Definition 2-3.1, is a representation of a tensor that offers a significant reduction in computational complexity when doing specific operations. This reduction is only the case when the considered tensor in TT format has low *ranks*. These ranks $\mathbf{r} = [R_0, R_1, \dots, R_d]$ determine the amount of information of the original tensor that is stored in the TT-format. The lower these ranks are, the more memory efficient the TT representation of the original tensor is.

Definition 2-3.1 (Tensor-Train [25, p. 2296])**.** The Tensor-Train decomposition of the $(i_1, i_2, \dots, i_d)$-th element of a d-dimensional tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is defined as

$$\boldsymbol{\mathcal{A}}(i_1, i_2, \dots, i_d) = \sum_{r_1, \dots, r_{d-1}} \boldsymbol{\mathcal{A}}^{(1)}(r_0, i_1, r_1) \boldsymbol{\mathcal{A}}^{(2)}(r_1, i_2, r_2) \cdots \boldsymbol{\mathcal{A}}^{(d)}(r_{d-1}, i_d, r_d) \tag{2-9}$$

where $\boldsymbol{\mathcal{A}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ for $n = 1, 2, \dots, d$ are the *TT-cores* that each represent a single dimension of the original tensor which are summed over the rank indices $r_1, \dots, r_{d-1}$ ($r_0 = r_1 = 1$). The ranks of the TT are described by $\mathbf{r} = [R_0, R_1, \dots, R_d]$, where $R_0 = R_d = 1$. Throughout this paper, the tensor train and its cores is described with $\boldsymbol{\mathcal{A}} = \langle\!\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \dots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\!\rangle$ for notation simplicity.

To convert a d-dimensional tensor to the Tensor-Train format, the TT-SVD algorithm [25] can be used as shown in Algorithm 1. Using a prescribed accuracy ζ or desired TT-ranks $[R_0, \dots, R_d]$, this algorithm makes use of a truncated singular value decomposition (SVD) of the tensor unfoldings to construct the TT-cores. The values at which these decompositions are truncated, are the TT-ranks. In Figure 2-5, a graphical representation of a vector's quantization into a 4-dimensional tensor is given. Subsequently, the tensor is transformed into a Tensor-Train with four cores, which can be done using the TT-SVD algorithm.

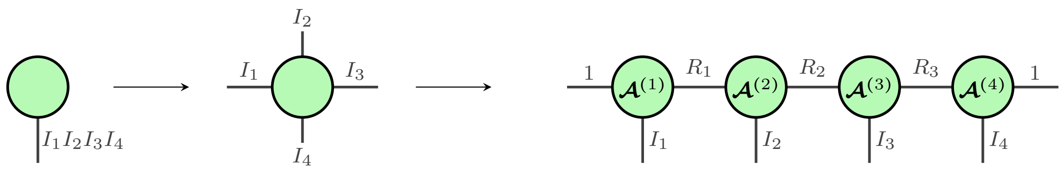


Figure 2-5: Quantization of a vector into a 4-dimensional tensor and subsequently the transformation into Tensor Train $\mathcal{A} \in \mathbb{R}^{I_1\times I_2\times I_3\times I_4}$ with 4 cores and ranks $\mathbf{r} = [1, R_1, R_2, R_3, 1]$ using the TT-SVD algorithm.

Algorithm 1: TT-SVD [25]

Data: $\mathcal{A} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_d}$, desired TT-ranks $\mathbf{r} = [R_0, \ldots, R_d]$ or accuracy ζ
Result: Tensor-Train $\mathcal{B} = \langle\langle \mathcal{B}^{(1)}, \mathcal{B}^{(2)}, \ldots, \mathcal{B}^{(d)} \rangle\rangle$, with desired ranks $\mathbf{r} = [R_0, \ldots, R_d]$ or where the approximation satisfies $\|\mathcal{A} - \mathcal{B}\|_F \leq \varepsilon \|\mathcal{A}\|_F$.

```
1  $\delta = \frac{\zeta}{\sqrt{d-1}} \|\mathcal{A}\|_F$ ;                 // Compute truncation parameter
2  $\mathbf{C} = \mathcal{A}$
3  for $n = 1, 2, \ldots, d-1$ do
4      $\mathbf{C} = \texttt{reshape}\Big(\mathbf{C}, \Big[R_{n-1}I_n, \frac{\texttt{numel}(\mathbf{C})}{R_{n-1}I_n}\Big]\Big)$ ;   // Left unfolding of tensor
5      $[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \texttt{svd}(\mathbf{C})$, where $\mathbf{C} = \mathbf{USV}^T + \mathbf{E}$, $\|\mathbf{E}\|_F \leq \delta$
6      $R_n = \texttt{size}(\mathbf{U}, 2)$
7      $\mathcal{B}^{(n)} = \texttt{reshape}\,(\mathbf{U}, [R_{n-1}, I_n, R_n])$
8      $\mathbf{C} = \mathbf{SV}^T$
9  end
10 $\mathcal{B}^{(d)} = \mathbf{C}$
```

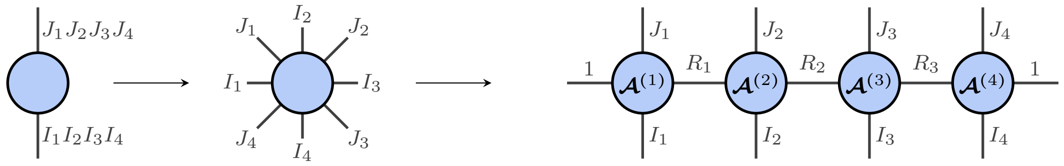
2-3-1 Tensor-Train matrix format

Aside from vectors, matrices can also be transformed into Tensor-Trains, which is called the Tensor-Train matrix (TTm) format [24]. The TTm format as defined in (2-10) consists of a similar structure as the TT format, however the cores now have an extra dimension. For example, the process of quantizing a matrix into a tensor and subsequently transforming it into TTm format (again making use of the TT-SVD along with some operations) is pictured in Figure 2-6. It must be noted that for the TTm format the order of dimensions is important before quantization, as the row and column dimensions must be ordered in a specific way before using the TT-SVD algorithm. Using the `permute` function in MATLAB, this can be ensured.

Definition 2-3.2 (Tensor-Train Matrix format [24])**.** A $(2d)$-dimensional tensor $\mathcal{A} \in \mathbb{R}^{I_1\times J_1\times\cdots\times I_d\times J_d}$ can be converted into the tensor-train matrix format, where the $(i_1, \ldots, i_d, j_1, \ldots, j_d)$-th element is equal to,

$$\mathcal{A}(i_1, \ldots, i_d, j_1, \ldots, j_d) = \sum_{r_1,\ldots,r_{d-1}} \mathcal{A}^{(1)}(r_0, i_1, j_1, r_1)\, \mathcal{A}^{(2)}(r_1, i_2, j_2, r_2) \cdots \mathcal{A}^{(d)}(r_{d-1}, i_d, j_d, r_d) \tag{2-10}$$

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 TTm-cores. R_n are the TTm-ranks with $R_0 = R_d = 1$.



$J_1 J_2 J_3 J_4$
$I_1 I_2 I_3 I_4$
J_1 I_2 J_2 I_1 I_3 J_4 I_4 J_3
1 $\mathcal{A}^{(1)}$ R_1 $\mathcal{A}^{(2)}$ R_2 $\mathcal{A}^{(3)}$ R_3 $\mathcal{A}^{(4)}$ 1
J_1 J_2 J_3 J_4
I_1 I_2 I_3 I_4

Figure 2-6: The example of quantization of a matrix into an 8-dimensional tensor and subsequently the transformation into Tensor Train matrix format with 4 cores using the TT-SVD algorithm.

2-3-2 Operations in TT(m) format

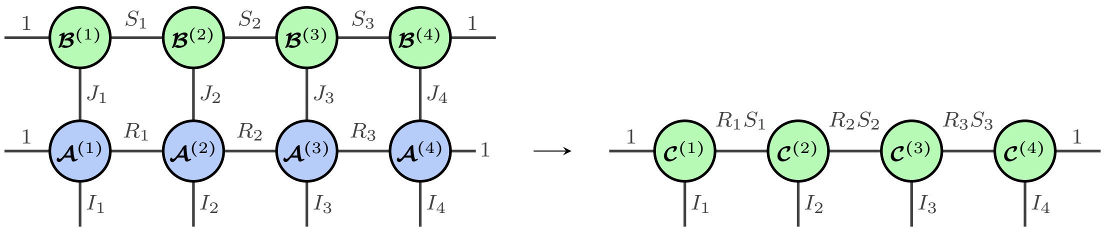
Addition. Having the TT or TTm decompositions of vectors and matrices allows us to do calculations in this format. The addition of two tensors is defined by (2-11) as formulated in [25]. The solution $\mathcal{C}$ has the ranks $R_n^{\mathcal{C}} = R_n^{\mathcal{A}} + R_n^{\mathcal{B}}$ and its inner and border cores are described by (2-12) and (2-13).

$$\mathcal{C} = \mathcal{A} + \mathcal{B} = \langle\!\langle \mathcal{A}^{(1)}, \dots, \mathcal{A}^{(d)} \rangle\!\rangle + \langle\!\langle \mathcal{B}^{(1)}, \dots, \mathcal{B}^{(d)} \rangle\!\rangle = \langle\!\langle \mathcal{C}^{(1)}, \dots, \mathcal{C}^{(d)} \rangle\!\rangle \tag{2-11}$$

$$\mathcal{C}^{(n)}(i_n) = \begin{bmatrix} \mathbf{A}^{(n)}(i_n) & 0 \\ 0 & \mathbf{B}^{(n)}(i_n) \end{bmatrix}, \quad n = 2, \dots, d-1 \tag{2-12}$$

$$\mathcal{C}^{(1)}(i_1) = \begin{bmatrix} \mathbf{A}^{(1)}(i_1) & \mathbf{B}^{(1)}(i_1) \end{bmatrix}, \quad \mathcal{C}^{(d)}(i_d) = \begin{bmatrix} \mathbf{A}^{(d)}(i_d) \\ \mathbf{B}^{(d)}(i_d) \end{bmatrix} \tag{2-13}$$

Matrix-vector product. Apart from addition and subtraction, multiplications can also be done in TT(m) format, such as the matrix-vector product (Figure 2-7). Here the TTm $\mathcal{A} = \langle\!\langle \mathcal{A}^{(1)}, \dots, \mathcal{A}^{(4)} \rangle\!\rangle$ (where $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$) with 4 cores is multiplied with the TT $\mathcal{B} = \langle\!\langle \mathcal{B}^{(1)}, \dots, \mathcal{B}^{(4)} \rangle\!\rangle$ (where $\mathcal{B}^{(n)} \in \mathbb{R}^{S_{n-1} \times K_n \times S_n}$). The resulting vector in TT format has the cores $\mathcal{C}^{(n)} \in \mathbb{R}^{R_{n-1}S_{n-1} \times I_n \times R_n S_n}$. The ranks of the solution are the ranks of the TT and TTm multiplied.

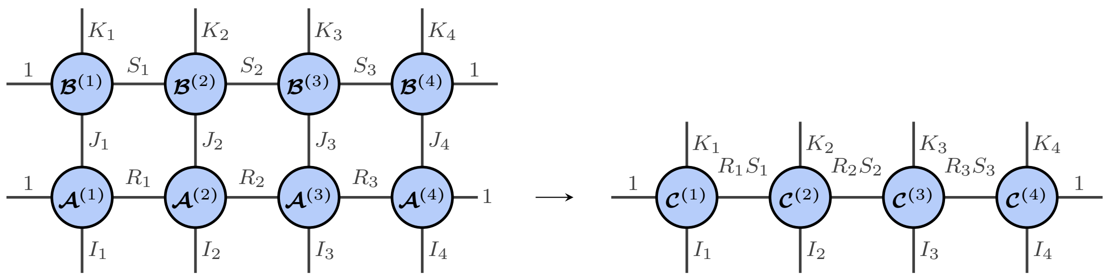


1 $\mathcal{B}^{(1)}$ S_1 $\mathcal{B}^{(2)}$ S_2 $\mathcal{B}^{(3)}$ S_3 $\mathcal{B}^{(4)}$ 1
J_1 J_2 J_3 J_4
1 $\mathcal{A}^{(1)}$ R_1 $\mathcal{A}^{(2)}$ R_2 $\mathcal{A}^{(3)}$ R_3 $\mathcal{A}^{(4)}$ 1
I_1 I_2 I_3 I_4
1 $\mathcal{C}^{(1)}$ R_1S_1 $\mathcal{C}^{(2)}$ R_2S_2 $\mathcal{C}^{(3)}$ R_3S_3 $\mathcal{C}^{(4)}$ 1
I_1 I_2 I_3 I_4

Figure 2-7: MTensor diagram for the TT(m) matrix-vector product.

Matrix-matrix product. In a similar way the matrix-matrix product of two TTm's can be visualised, shown in Figure 2-8. Here the TTm $\mathcal{A} = \langle\!\langle \mathcal{A}^{(1)}, \dots, \mathcal{A}^{(4)} \rangle\!\rangle$ (where $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ for $n = 1, \dots, 4$) with 4 cores is multiplied with the TTm $\mathcal{B} = \langle\!\langle \mathcal{B}^{(1)}, \dots, \mathcal{B}^{(4)} \rangle\!\rangle$ (where $\mathcal{B}^{(n)} \in \mathbb{R}^{S_{n-1} \times J_n \times K_n \times S_n}$ for $n = 1, \dots, 4$). The resulting matrix in TTm format has the cores $\mathcal{C}^{(n)} \in \mathbb{R}^{R_{n-1}S_{n-1} \times I_n \times K_n \times R_n S_n}$

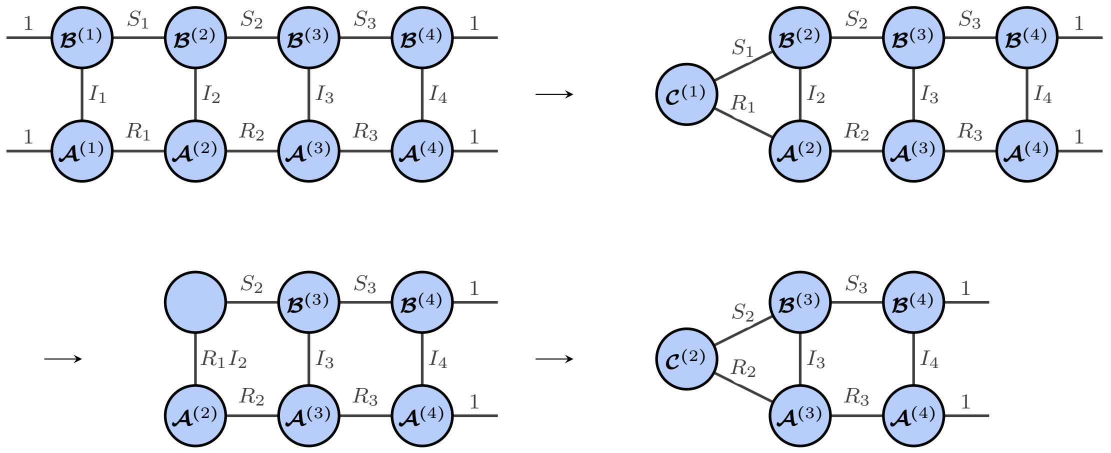
Inner product. The dot or inner product of two vectors in TT format with equal modes can be done by connecting the cores over these modes. The computation process then requires



K_1 K_2 K_3 K_4
$\mathcal{B}^{(1)}$ $\mathcal{B}^{(2)}$ $\mathcal{B}^{(3)}$ $\mathcal{B}^{(4)}$
S_1 S_2 S_3
J_1 J_2 J_3 J_4
$\mathcal{A}^{(1)}$ $\mathcal{A}^{(2)}$ $\mathcal{A}^{(3)}$ $\mathcal{A}^{(4)}$
R_1 R_2 R_3
I_1 I_2 I_3 I_4
1
$\longrightarrow$
$\mathcal{C}^{(1)}$ $\mathcal{C}^{(2)}$ $\mathcal{C}^{(3)}$ $\mathcal{C}^{(4)}$
R_1S_1 R_2S_2 R_3S_3

Figure 2-8: Tensor diagram for the TTm matrix-matrix product.

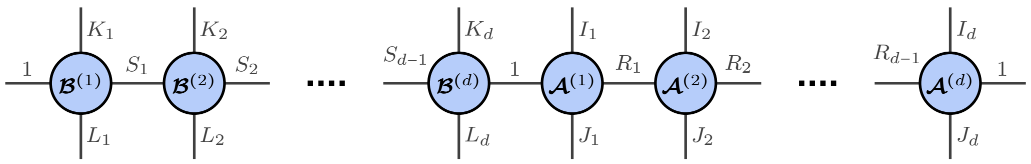
contraction of the cores by 'absorbing' each core step by step in a 'zipper'-like manner, as visualized in Figure 2-9. The final two cores can then be contracted to yield a scalar value, which is the inner product of both TT's. The intermediate solutions $\mathcal{C} = \langle\!\langle \mathcal{C}^{(1)}, \dots, \mathcal{C}^{(d-1)} \rangle\!\rangle$ are the so-called *partial contraction matrices*. Using a simple algorithm, these matrices can be saved and used for *randomized rounding algorithms* which will be elaborated further in this report.



$\mathcal{B}^{(1)}$ $\mathcal{B}^{(2)}$ $\mathcal{B}^{(3)}$ $\mathcal{B}^{(4)}$
S_1 S_2 S_3
I_1 I_2 I_3 I_4
$\mathcal{A}^{(1)}$ $\mathcal{A}^{(2)}$ $\mathcal{A}^{(3)}$ $\mathcal{A}^{(4)}$
R_1 R_2 R_3
1
$\longrightarrow$
$\mathcal{C}^{(1)}$
$R_1 I_2$
$\mathcal{C}^{(2)}$

Figure 2-9: Tensor diagram of the TT inner product of two TT's $\mathcal{A}$ and $\mathcal{B}$ with 4 cores. The intermediate solutions $\mathbf{C} = \langle\!\langle \mathbf{C}^{(1)}, \dots, \mathbf{C}^{(d-1)} \rangle\!\rangle$ are the partial contraction matrices [10].

Kronecker product. The Kronecker product of $d_{\mathcal{A}}$-dimensional TTm $\mathcal{A} = \langle\!\langle \mathcal{A}^{(1)}, \dots, \mathcal{A}^{(d_{\mathcal{A}})} \rangle\!\rangle$ (where $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ for $n = 1, \dots, d_{\mathcal{A}}$) and $d_{\mathcal{B}}$-dimensional TTm $\mathcal{B} = \langle\!\langle \mathcal{B}^{(1)}, \dots, \mathcal{B}^{(d_{\mathcal{B}})}$ $\rangle\!\rangle$ (where $\mathcal{B}^{(n)} \in \mathbb{R}^{S_{n-1} \times K_n \times L_n \times S_n}$ for $n = 1, \dots, d_{\mathcal{B}}$) results in a TTm $\mathcal{C}$ which are both TTm's 'stitched' together in opposite order, where the last core of $\mathcal{B}$ is connected to the first core of $\mathcal{A}$ with a dimension of 1. The corresponding tensor diagram is shown in Figure 2-10.



K_1 K_2 K_d I_1 I_2 I_d
$\mathcal{B}^{(1)}$ $\mathcal{B}^{(2)}$ $\mathcal{B}^{(d)}$ $\mathcal{A}^{(1)}$ $\mathcal{A}^{(2)}$ $\mathcal{A}^{(d)}$
S_1 S_2 S_{d-1} R_1 R_2 R_{d-1}
1
L_1 L_2 L_d J_1 J_2 J_d

Figure 2-10: Tensor diagram of $\mathcal{A} \otimes \mathcal{B}$, where $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ and $\mathcal{B}^{(n)} \in \mathbb{R}^{S_{n-1} \times K_n \times L_n \times S_n}$.

TT(m) value extraction. Extracting one or multiple values from vector or matrix represented as a d-dimensional tensor in TT(m) format $\boldsymbol{\mathcal{A}} = \langle\!\langle \boldsymbol{\mathcal{A}}^{(1)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\!\rangle$ can be done by firstly computing the selection matrices for each mode. How these matrices are formed, is described in detail in [19]. For these selection matrices, the following property holds.

$$\mathbf{S} = \mathbf{S}_d \odot \mathbf{S}_{d-1} \odot \ldots \odot \mathbf{S}_1 \tag{2-14}$$

where the Khatri-Rao product of the d binary matrices gives the original selection matrix $\mathbf{S}$. The selection matrices satisfy $\mathbf{S}_n \in \mathbb{R}^{I_n \times K}$, where K is the amount of extracted values. The k-th column of the selection matrix $\mathbf{S}_n$ is defined as the standard basis vector $\mathbf{e}_{i_n}^{(n)} \in \mathbb{R}^{I_n} (n = 1, 2 \ldots, d)$ of the k-th extracted entry $(k = 1, \ldots, K)$. When multiplying this matrix with the original vector that is reconstructed from the TT format, we get the desired extracted value(s). The algorithms to compute these matrices for TT and TTm format (Algorithm 10 and 11) are listed in Appendix A. To extract the values, the original tensor in TT(m) format is contracted while performing Khatri-Rao products between its cores and the selection matrices. In Algorithm 12 and 13 the psuedo-code is shown for the extraction from a TT or TTm, respectively. These algorithms are listed in Appendix A.

TTm column extraction. To extract one or multiple columns from a matrix in TTm format, a similar method can be used as the mentioned TT(m) value extraction. For computational efficiency in the TNKF, the extracted columns must stay in TTm format to avoid converting to the TT-format using the TT-SVD algorithm. To do this, firstly the selection matrices are computed using Algorithm 11. These matrices again satisfy the property in (2-14). Next, using a method where the Khatri-Rao product is computed with the selection matrices and each TTm core. During this process, use is made of the TT-SVD algorithm making it possible to truncate the ranks of the extracted columns. This property can be exploited, which will be discussed later in this thesis. The algorithm for the extraction of columns in TTm format (Algorithm 14) is listed in Appendix A.

Scalar multiplication. To do scalar multiplication of a tensor in TT format, one of the cores must be multiplied with the scalar value. Orthogonality, which is explained further in this section, can be preserved if the site-k core is multiplied with the scalar value.

Transpose. The transpose of a matrix is TTm format can be computed by permuting the inner dimensions of the cores. If a matrix $\mathbf{A}$ is represented by a TTm $\boldsymbol{\mathcal{A}}$ with cores $\boldsymbol{\mathcal{A}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$, its transpose $\mathbf{A}^T$ is a TTm with cores $\boldsymbol{\mathcal{A}}^{(n)} \in \mathbb{R}^{R_{n-1} \times J_n \times I_n \times R_n}$.

Orthogonalization. A tensor in TT format can be put into orthogonal form, defined in Definition 2-3.3. The orthogonality of the cores of a TT is essential for the rounding algorithm that will be discussed further in this chapter. Algorithm 2 shows the pseudo-code for transforming a d-dimensional tensor in TT format into a site-k orthogonal TT [31]. Using the thin QR factorization, each core can be orthogonalized by absorbing the $\mathbf{R}$ matrix that contains the norm into the next core. The k-th site refers to the core that contains the total norm of the TT, while all other cores are right or left orthogonal.

Definition 2-3.3 (Site-k orthogonality [7, p. 143])**.** A d-dimensional tensor in TT format $\boldsymbol{\mathcal{A}} = \langle\!\langle \boldsymbol{\mathcal{A}}^{(1)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\!\rangle$ is called in site-k orthogonal form with $1 \leq k \leq d$ if the following holds for the unfoldings of the cores,

$$\begin{aligned} \left(\mathbf{A}_{\mathrm{L}}^{(n)}\right)^T \mathbf{A}_{\mathrm{L}}^{(n)} &= \mathbf{I}_{R_n}, \quad n = 1, 2, \ldots, k-1 \\ \mathbf{A}_{\mathrm{R}}^{(n)} \left(\mathbf{A}_{\mathrm{R}}^{(n)}\right)^T &= \mathbf{I}_{R_{n-1}}, \quad n = k+1, k+2, \ldots, d \end{aligned} \tag{2-15}$$

where the left and right unfolding of the cores are defined as $\mathbf{A}_{\mathrm{L}}^{(n)} = \mathbf{A}^{(n)} \in \mathbb{R}^{R_{n-1}I_n \times R_n}$ and $\mathbf{A}_{\mathrm{R}}^{(n)} = \mathbf{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n R_n}$. When $k = d$, the TT is in left-orthogonal form and when $k = 1$ the TT is in right-orthogonal form. From the relations in (2-15) it follows that the TT is site-k orthogonal, the matrices $\mathbf{A}_{\mathrm{L}}^{(n)}$ and $\mathbf{A}_{\mathrm{R}}^{(n)}$ have orthonormal rows and columns.

Algorithm 2: Orthogonalization [25]

```
Data: $d$-dimensional tensor $\boldsymbol{\mathcal{A}}$ in TT-format $\boldsymbol{\mathcal{A}} = \langle\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \dots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$
      with ranks $\mathbf{r} = [R_0, \dots, R_d]$
Result: Site-$k$ orthogonal tensor $\boldsymbol{\mathcal{A}}$ in TT format
1  for $n = 1, 2, \dots, k-1$ do
2      $\mathbf{A}_L$ = reshape($\boldsymbol{\mathcal{A}}^{(n)}, R_{n-1}I_n, R_n$) ;        // Left-to-right orthogonalization
3      $[\mathbf{Q}, \mathbf{R}]$ = qr($\mathbf{A}_L, 0$)
4      $R_n$ = size($\mathbf{Q}, 2$)
5      $\boldsymbol{\mathcal{A}}^{(n)}$ = reshape($\mathbf{Q}, R_{n-1}, I_n, R_n$)
6      $\boldsymbol{\mathcal{A}}^{(n+1)} = \boldsymbol{\mathcal{A}}^{(n+1)} \times_1 \mathbf{R}$
7  end
8  for $k = d, d-1, \dots, k+1$ do
9      $\mathbf{A}_R$ = reshape($\boldsymbol{\mathcal{A}}^{(n)}, R_{n-1}, I_n R_n$) ;        // Right-to-left orthogonalization
10     $[\mathbf{Q}, \mathbf{R}]$ = qr($\mathbf{A}_R^T, 0$)
11     $R_{n-1}$ = size($\mathbf{Q}, 2$)
12     $\boldsymbol{\mathcal{A}}^{(n)}$ = reshape($\mathbf{Q}^T, R_{n-1}, I_n, R_n$)
13     $\boldsymbol{\mathcal{A}}^{(n-1)} = \boldsymbol{\mathcal{A}}^{(n-1)} \times_3 \mathbf{R}^T$
14 end
```

Rounding. While performing operations with TT(m)'s, their ranks can increase. TT addition, for example, results in the summation of the ranks of both TT(m)'s while the TT inner product results in multiplication of ranks. This rank increase results in increased computational load during TT operations, which calls for a method to truncate these ranks: TT-rounding. Algorithm 3 shows the process of rounding a vector in TT format. Firstly, Algorithm 2 is used to orthogonalize the TT from right-to-left yielding a left orthogonal TT. After this, a compression sweep is carried out using the SVD with rank truncation to complete the rounding of the TT. The SVD is used here, as it exposes the range of a matrix. This allows for the truncation of information or rank of this matrix. Using Algorithm 3, the ranks of the considered TT can be truncated to any desired (positive) value.

Algorithm 3: TT-rounding [25]

```
Data: $d$-dimensional tensor $\boldsymbol{\mathcal{A}}$ in the TT-format
      $\boldsymbol{\mathcal{A}} = \langle\!\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$, desired TT-ranks $\mathbf{r} = [R_0, \ldots, R_d]$
Result: $\boldsymbol{\mathcal{B}}$ in the TT-format with ranks equal to $\mathbf{r} = [R_0, \ldots, R_d]$
1 $\boldsymbol{\mathcal{A}}$ = orthogonalization($\boldsymbol{\mathcal{A}}, 1$) ;                // Algorithm 2
2 for $n = 1, 2, \ldots, d-1$ do
3     $\mathbf{A}_n$ = reshape($\boldsymbol{\mathcal{A}}^{(n)}, [R_{n-1}, I_n, R_n]$) ;    // Right-to-left compression
4     $[\mathbf{U}, \mathbf{S}, \mathbf{V}]$ = svd($\mathbf{A}_n$)
5     $R_n$ = size($\mathbf{U}, 2$)
6     $\boldsymbol{\mathcal{A}}^{(n)}$ = reshape($\mathbf{U}, R_n, I_n, R_{n+1}$)
7     $\boldsymbol{\mathcal{A}}^{(n+1)} = \boldsymbol{\mathcal{A}}^{(n+1)} \times_1 (\mathbf{VS})^T$
8 end
```

Chapter 3

Tensor-Networked Kalman Filter for Streaming Video Completion

This chapter gives a concise overview of how the Tensor-Networked Kalman filter works and what design variables influence its performance and speed. For a more in depth description of the Tensor-Networked Kalman Filter for streaming video completion it is recommended to read the work by de Rooij (2019) [30]. In [18], an attempt is made to improve the performance of the TNKF by implementing a Modified Gram-Schmidt (MGS) algorithm in TT format. Parts of this chapter are based on findings in [11, 30, 21].

3-1 Kalman filter

A Kalman filter is a model-based estimator that uses measurements to estimate a system's varying quantities or states. The Kalman filter belongs to a class known as *observers*, which uses the model of the system and measurements at time step k to approximate the state vector at the next time step $k+1$. The fact that a Kalman filter uses information from past states to estimate the current state makes it a recursive estimator. As described in Chapter 1, a video is a discrete-time system by nature as it consists of a discrete sequence of events. Since the streaming video completion problem requires real-time reconstruction of (parts of) a frame, a Kalman filter is a suitable solution given its properties.

3-1-1 State-space equations

In (3-1) a discrete-time state-space model is shown, which is a mathematical representation of a (physical) system [33]. $\mathbf{A}[k] \in \mathbb{R}^{n\times n}$ represents the state-transition matrix, $\mathbf{B}[k] \in \mathbb{R}^{n\times m}$ the input matrix, $\mathbf{C}[k] \in \mathbb{R}^{\ell\times m}$ the measurement matrix and $\mathbf{D}[k] \in \mathbb{R}^{\ell\times n}$ the output matrix. These are called the system matrices and describe the system's characteristics. $\mathbf{x}[k] \in \mathbb{R}^{n}$,

$\mathbf{u}[k] \in \mathbb{R}^m$ and $\mathbf{y}[k] \in \mathbb{R}^\ell$ denote the state, input and output vectors, respectively. The vectors $\mathbf{w}[k] \in \mathbb{R}^n$ and $\mathbf{v}[k] \in \mathbb{R}^\ell$ denote the process noise and measurement noise, respectively.

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

In order to use a Kalman filter to estimate the frames of a video, firstly, the state-space model must be defined. Each frame of a greyscale video is represented by a matrix $\mathbf{X}[k] \in \mathbb{R}^{M\times N}$, which can be transformed into a state vector by vectorizing the entries of the matrix using the `reshape` function in MATLAB: $\mathbf{x}[k] = \texttt{reshape}(\mathbf{X}[k], MN, 1)$. In the case of a color video, this must be done three times for each color, as each frame contains values for the red, green and blue color intensity. This process is described in (3-2).

$$\mathbf{X}[k] = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,N} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M,1} & x_{M,2} & \cdots & x_{M,N} \end{bmatrix} \quad \rightarrow \quad \mathbf{x}[k] = \left[\begin{array}{c} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{M,1} \\ \hline x_{1,2} \\ x_{2,2} \\ \vdots \\ x_{M,2} \\ \hline \vdots \\ \hline x_{1,N} \\ x_{2,N} \\ \vdots \\ x_{M,N} \end{array}\right] \tag{3-2}$$

The state vector $\mathbf{x} \in \mathbb{R}^{MN}$ can then be quantized into a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{M_1\times M_2\times\cdots\times M_{d_M}\times N_1\times N_2\times\cdots\times N_{d_N}}$, using quantization parameters $M = M_1M_2\cdots M_{d_M}$ and $N = N_1N_2\cdots N_{d_N}$. Subsequently, it can be put into TT format using the TT-SVD algorithm described in Algorithm 1.

Since surveillance cameras are fixed, most of the pixels in the frames have a constant value over time when no moving objects pass through them and light intensity does not change. Because of this property, it is possible to do background subtraction. This is done by computing the mean of each pixel over a long sequence of time and subtracting the resulting frame from the original frame to determine the foreground ($\mathbf{X}_{\text{foreground}} = \mathbf{X}_{\text{frame}} - \mathbf{X}_{\text{background}}$). This holds advantages that eventually result in more memory efficiency during computations. The state vector that is used during computations can then be described as

$$\mathbf{x}_{\text{F}}[k] = \mathbf{x}[k] - \mathbf{x}_{\text{B}} \tag{3-3}$$

where $\mathbf{x}_{\text{F}}[k]$ is the estimated foreground, $\mathbf{x}[k]$ is the original vectorized video frame and $\mathbf{x}_{\text{B}}$ is the estimated background. An example of the difference between a regular frame and the background of surveillance footage is plotted in Figure 3-1. The background is estimated fairly well, but since the background is estimated over a limited sequence of frames (584 in total), slightly darker spots can be observed where persons moved.

Choosing the state-transition matrix $\mathbf{A}[k]$ is very difficult for a video since this is a highly unpredictable process. For surveillance videos or videos of any other type, information from

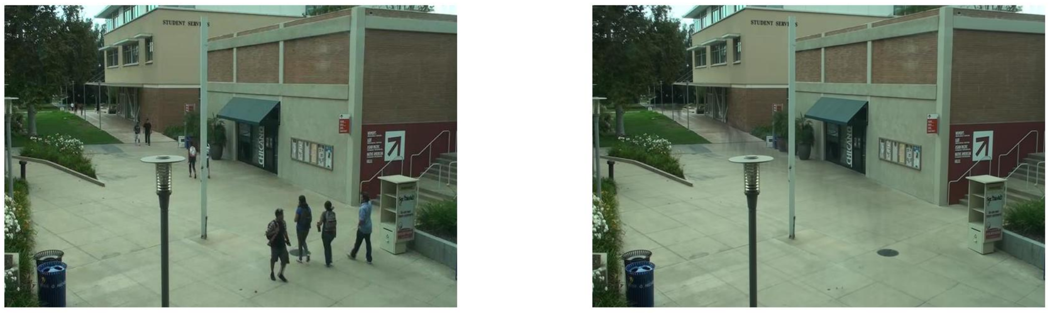


Figure 3-1: Regular frame (left) and background frame (right) estimated from a 584 frame video sequence [34].

past frames is not necessarily a good predictor for future frames. Two consecutive frames, however, do have many similarities if the video feed's frame rate is high enough. Because of this high similarity and high unpredictability of the video, the state-transition matrix is chosen as identity and the difference between each frame is captured within the process noise $\mathbf{w}[k] = \mathbf{x}[k+1] - \mathbf{x}[k]$. For the location of the uncorrupted pixels, direct access and a constant location is assumed which is incorporated in the measurement matrix $\mathbf{C}$. Because of this, there is no measurement noise ($\mathbf{v}[k] = 0$). This leads to the state-space model for a surveillance video described by (3-4) [11].

$$\begin{aligned} \mathbf{x}[k+1] &= \mathbf{x}[k] + \mathbf{w}[k] \\ \mathbf{y}[k] &= \mathbf{C}\mathbf{x}[k] \end{aligned} \tag{3-4}$$

where $\mathbf{x}[k] \in \mathbb{R}^{MN}$ is the vectorized video frame at time k and $\mathbf{y}[k] \in \mathbb{R}^{J}$ contains the uncorrupted pixels. The process noise is assumed to be independent, zero-mean Gaussian white noise with probability distribution $\mathbf{w}[k] \sim \mathcal{N}(0, \mathbf{W}[k])$.

The process noise covariance matrix $\mathbf{W}[k]$ can be defined in such a way that its values are dependent on the distance of pixels relative to each other. This can be done by choosing the matrices $\mathbf{W}_1[k] \in \mathbb{R}^{N\times N}$ and $\mathbf{W}_2[k] \in \mathbb{R}^{M\times M}$ as Toeplitz band matrices. These matrices have their highest values on the diagonal and these values gradually decrease further away from the diagonal. We define the process noise covariance matrix as the Kronecker product of these two matrices, multiplied by the variance between the two previous frames $\sigma_W^2[k]$ as shown in (3-5). Equations (3-6) - (3-8) define the variance, Toeplitz matrices and their elements. Here, α is the bandwidth of the Toeplitz matrices, which is set around a value of 10 in [30].

$$\mathbf{W}[k] = \sigma_W^2[k]\mathbf{W}_1[k] \otimes \mathbf{W}_2[k] \tag{3-5}$$

$$\sigma_W^2[k] = \frac{1}{I}\sum_{i=1}^{I}\left(x_i[k-1] - x_i[k-2]\right)^2, \quad \mathbf{x} \in \mathbb{R}^{I=MN} \tag{3-6}$$

$$\mathbf{W}_n(\alpha) = \begin{bmatrix} w(0) & w(1) & \cdots & w(\alpha) & 0 & \cdots & 0 \\ w(1) & w(0) & w(1) & \cdots & w(\alpha) & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & w(\alpha) & \cdots & w(1) & w(0) \end{bmatrix}, \quad n = \{1, 2\} \tag{3-7}$$

$$w(d) = \max\left(0, 1 - \frac{d}{\alpha + 1}\right) \tag{3-8}$$

Here, d is the distance between two pixels. Again using the TT-SVD algorithm, both matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ can be put into TTm-format. This ensures the process noise covariance matrix $\mathbf{W}$ to be in TTm format as well since the outer product of two TT(m)'s is a 'stitched' version of both in reverse order as explained in the previous chapter (Figure 2-10). In [30] it is proven that this choice of $\mathbf{W}[k]$ ensures positive definiteness of this matrix. This property is necessary to ensure the stability of the TNKF.

3-1-2 Kalman filter algorithm

Using the previously derived state-space model, the Kalman filter equations can be constructed that estimate the corrupted frames. As described in [33], the Kalman filter algorithm can be divided into two stages: the prediction step (3-9) and the update step (3-10). Note that the equations presented in (3-9) and (3-10) are rewritten to match the state-space model in (3-4). The hat operator, $\hat{}$, means an estimate of a variable. The superscripts '−' and '+' denote predicted (prior) and updated (posterior) estimates, respectively.

Prediction step:

$$\begin{array}{ll} \text{Predicted state estimate} & \hat{\mathbf{x}}[k]^- = \hat{\mathbf{x}}[k-1]^+ \\ \text{Predicted covariance} & \mathbf{P}[k]^- = \mathbf{P}[k-1]^+ + \mathbf{W}[k] \end{array} \tag{3-9}$$

Update step:

$$\begin{array}{ll} \text{Measurement residual} & \mathbf{v}[k] = \mathbf{y}[k] - \mathbf{C}\hat{\mathbf{x}}[k]^- \\ \text{Residual covariance matrix} & \mathbf{S}[k] = \mathbf{C}\mathbf{P}[k]^-\mathbf{C}^T \\ \text{Kalman gain} & \mathbf{K}[k] = \mathbf{P}[k]^-\mathbf{C}^T\mathbf{S}[k]^{-1} \\ \text{Updated state estimate} & \hat{\mathbf{x}}[k]^+ = \hat{\mathbf{x}}[k]^- + \mathbf{K}[k]\mathbf{v}[k] \\ \text{Updated covariance matrix} & \mathbf{P}[k]^+ = \mathbf{P}[k]^- - \mathbf{K}[k]\mathbf{S}[k]\mathbf{K}[k]^T \end{array} \tag{3-10}$$

The state covariance matrix $\mathbf{P}[k] \in \mathbb{R}^{MN \times MN}$ can be assumed to be the covariance of the foreground, where we assume the background remains stationary. This yield the following expression for the covariance matrix, where $\mathbb{E}$ denotes the expectation.

$$\mathbf{P} = \mathbb{E}\left[\left(\mathbf{x}_\mathrm{F} - \overline{\mathbf{x}}_\mathrm{F}\right)\left(\mathbf{x}_\mathrm{F} - \overline{\mathbf{x}}_\mathrm{F}\right)^T\right] \tag{3-11}$$

Because the TNKF is a recursive estimator, initial information is necessary to perform the estimation. The Kalman filter assumes a Gaussian distribution of the initial state vector $\mathbf{x}[0] \sim \mathcal{N}(\overline{\mathbf{x}}[0], \mathbf{P}[0])$, where $\overline{\mathbf{x}}[0]$ is the mean of the initial state vector $\mathbf{x}[0]$ and $\mathbf{P}[0]$ is the initial state covariance matrix. Choosing $\mathbf{P}[0] = \sigma_P^2\mathbf{I} = \mathbf{I}$ generally gives good results. Choosing the last uncorrupted frame as the initial state vector and using relatively low values for the covariance matrix may cause problems during reconstruction, however. Since the variance is low, the Kalman Filter cannot properly update the state when large parts of the frames are moving. This can result in a 'shadow effect' of moving objects within the frames when the moving object is relatively large. This can be solved by choosing the background of the video as initial state vector, and calculating the variance σ_P^2 using the difference between the background pixels (the mean) and the last uncorrupted frame (3-12).

$$\sigma_P^2 = \frac{1}{I}\sum_{i=1}^{I}\left(x_i\left[t_c-1\right]-\bar{x}_i\right)^2, \quad \mathbf{x}\in\mathbb{R}^{I=MN} \tag{3-12}$$

Due to the large size of the matrix, it can be initialized in the TTm format using the previously defined quantization parameters. This is done by defining matrices $\mathbf{P}_1 \in \mathbb{R}^{N\times N}$ and $\mathbf{P}_2 \in \mathbb{R}^{M\times M}$ where $\mathbf{P} = \mathbf{P}_1 \otimes \mathbf{P_2} \in \mathbb{R}^{MN\times MN}$, and then quantizing both matrices to the tensors $\boldsymbol{\mathcal{P}}_1 \in \mathbb{R}^{N_1\times N_1\times\cdots\times N_{d_N}\times N_{d_N}}$ and $\boldsymbol{\mathcal{P}}_2 \in \mathbb{R}^{M_1\times M_1\times\cdots\times M_{d_M}\times M_{d_M}}$. The result of the Kronecker product of both is $\boldsymbol{\mathcal{P}} \in \mathbb{R}^{(M_1\times M_1\times\cdots\times M_{d_M}\times M_{d_M})\times(N_1\times N_1\times\cdots\times N_{d_N}\times N_{d_N})}$ and can be transformed into a TTm-decomposed version of the state covariance matrix $\mathbf{P}$, again using the TT-SVD algorithm. This process is visualised in Figure 2-6 in the previous chapter.

3-1-3 Partitioned Update Kalman Filter

During the Kalman filter update step, the inversion of $\mathbf{S}[k] \in \mathbb{R}^{J\times J}$ is time consuming because of its computational complexity of $\mathcal{O}(J^3)$. When the number of measurements J is large, the computational complexity results in a high computational load. The Partitioned Update Kalman Filter (PUKF) [28] can be used to avoid this. The PUKF computes the Kalman update sequentially for every measurement and uses the partially updated state-vector as a prior for the next measurement. Here, the locations of the measurements are stored in a vector $\mathbf{c} \in \mathbb{R}^J$ and computing the inverse of $\mathbf{S}$ is done with J scalar inversions. Using the TT value extraction described in the previous chapter to select values and columns from the tensors in TT format, the equations can be updated in each step. This results in further simplified equations for the PUKF, which are given in (3-13) - (3-17).

$$v_j[k] = y_j[k] - \mathbf{x}_{j-1}(c_j) \tag{3-13}$$

$$s_j[k] = \mathbf{P}_{j-1}(c_j, c_j) \tag{3-14}$$

$$\mathbf{k}_j[k] = \frac{\mathbf{P}_{j-1}(:, c_j)}{s_j[k]} \tag{3-15}$$

$$\hat{\mathbf{x}}_j[k]^+ = \hat{\mathbf{x}}_{j-1}[k]^+ + \mathbf{k}_j[k]v_j[k] \tag{3-16}$$

$$\mathbf{P}_j[k]^+ = \mathbf{P}_{j-1}[k]^+ - \mathbf{k}_j[k]s_j[k]\mathbf{k}_j[k]^T \tag{3-17}$$

3-1-4 Corrupted data

The corrupted video data that is used as an input in the TNKF is created by taking the element-wise or Hadamard product of the frames with a *mask matrix* $\mathbf{J}$, as shown in (3-18).

$$\mathbf{X}_c[k] = \mathbf{J} * \mathbf{X}[k] \tag{3-18}$$

Here, the elements of $\mathbf{J}$ have a value of 1 at the uncorrupted pixel location and a value of 0 at the corrupted pixel location.

3-2 Tensor-Networked Kalman Filter

In order to update the previously derived equations, an algorithm can be iterated that updates the state and covariance in TT(m) format. In [2], the TNKF is introduced, and its operations in TT format are discussed. Now that all essential variables and equations have been defined, these can be combined into the TNKF algorithm, shown in Algorithm 4. Using the previously defined TT(m) operations and algorithms, the TNKF recursively estimates the current frame.

Algorithm 4: TN Kalman Filter [30]

```
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: $\mathbf{x}[k], \mathbf{P}[k]$.
1  $\mathbf{P}_0 = \mathbf{P}[k-1] + \mathbf{W}[k]$ ;                          // Prediction
2  $\mathbf{P}_0 = \texttt{rounding}(\mathbf{P}_0, R_P)$ ;                       // Algorithm 3
3  $\mathbf{x}_0 = \mathbf{x}[k]$
4  for $j = 1, 2, \ldots, J$ do
5      $v_j = y_j[k] - \mathbf{x}_{j-1}(c_j)$ ;                                  // Update
6      $s_j = \mathbf{P}_{j-1}(c_j, c_j)$
7      $\mathbf{k}_j = \mathbf{P}_{j-1}(:, c_j) / s_j$
8      $\mathbf{x}_j = \mathbf{x}_{j-1} + \mathbf{k}_j v_j$
9      $\mathbf{P}_j = \mathbf{P}_{j-1} - \mathbf{k}_j s_j \mathbf{k}_j^T$
10     if $\max(\text{rank}(\mathbf{P}_j)) \leq R_P^{\max}$ then
11         $\mathbf{P}_j = \texttt{rounding}(\mathbf{P}_j, R_P)$ ;               // TT-rounding
12     end
13     if $\max(\text{rank}(\mathbf{x}_j)) \leq R_x^{\max}$ then
14         $\mathbf{x}_j = \texttt{rounding}(\mathbf{x}_j, R_x)$
15     end
16 end
17 Set: $\mathbf{x}[k] = \mathbf{x}_J$ and $\mathbf{P}[k] = \mathbf{P}_J$
```

During the operations of the TNKF, the ranks of the Tensor-Trains can increase. To prevent the ranks from blowing up to such high values that it dramatically slows down computations, a rounding step (Algorithm 3) is necessary. Executing the rounding algorithm ensures that when the TT-formats of the state and covariance reach a specific maximum rank ($R_{\mathbf{x}}^{\max}$ and $R_{\mathbf{P}}^{\max}$), they are rounded back to the desired ranks ($R_{\mathbf{x}}$ and $R_{\mathbf{P}}$). For the desired state rank $R_{\mathbf{x}}$, higher values result in a better reconstruction, but this gives higher computational loads. In [30] it is found that to ensure stability of the completion, the desired covariance rank must be chosen as $R_{\mathbf{P}} = 1$.

Chapter 4

Speedup Methods

To speed up the computations of the TNKF, three methods are proposed in this chapter. The first method uses a Block Update Kalman Filter [27, 8], which updates the state and covariance with 'blocks' of measurements instead of scalar elements to reduce the amount of Kalman Filter iterations. The second method substitutes the standard deterministic rounding algorithm with randomized rounding algorithms, which use randomization to decrease the computational complexity of the rounding procedure [10]. Lastly, the principal angles between the orthonormal bases of the state TT and covariance TTm are investigated. This is done to analyze if further simplification of the TNKF update operations is possible.

4-1 Block Update Tensor-Networked Kalman Filter

In the previous chapter, the Partitioned Update Kalman Filter (PUKF) with an elementwise update was described as implemented in [30]. This simplifies the Kalman filter equations, as many TT(m) operations can be done with scalar multiplication. A downside to this is that a lot of iterations are necessary, as the Kalman Filter equations must be computed for each measured pixel. In order to tackle this, a Block Update Tensor-Networked Kalman Filter [27, 8] is proposed, which is an adaptation to the PUKF. The Block Update TNKF combines blocks of measurements and simultaneously updates the state and covariance for these measurements. This cuts the number of Kalman filter update iterations down with a factor equal to the block size, which reduces the amount of TT(m) operations such as rounding, multiplication and addition. In this section, the Block Update TNKF algorithm (Algorithm 5) is presented and discussed in terms of its operations.

```
Algorithm 5: Block Update TNKF [27, 30]
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 $\mathbf{c} \in \mathbb{R}^{J}$. Number of blocks $N = J/B$, where $B$ is the block size.
Result: Updated state and covariance: $\mathbf{x}[k], \mathbf{P}[k]$.
1  $\mathbf{P}_0 = \mathbf{P}[k-1] + \mathbf{W}[k]$ ;                         // Prediction
2  $\mathbf{P}_0 = \texttt{rounding}(\mathbf{P}_0, R_P)$ ;                     // Algorithm 3
3  $\mathbf{x}_0 = \mathbf{x}[k]$
4  for $j = 1, 2, \dots, N$ do
5      $\mathbf{v}_j = \mathbf{y}_j[k] - \mathbf{x}_{j-1}(\mathbf{c}_j)$ ;                 // Update
6      $\mathbf{S}_j = \mathbf{P}_{j-1}(\mathbf{c}_j, \mathbf{c}_j)$
7      $\mathbf{K}_j = \mathbf{P}_{j-1}(:, \mathbf{c}_j)\, \mathbf{S}_j^{-1}$
8      $\mathbf{x}_j = \mathbf{x}_{j-1} + \mathbf{K}_j \mathbf{v}_j$
9      $\mathbf{P}_j = \mathbf{P}_{j-1} - \mathbf{K}_j \mathbf{S}_j \mathbf{K}_j^T$
10     if $\max(\mathrm{rank}(\mathbf{P}_j)) \leq R_P^{\max}$ then
11         $\mathbf{P}_j = \texttt{rounding}(\mathbf{P}_j, R_P)$ ;             // TT-rounding
12     end
13     if $\max(\mathrm{rank}(\mathbf{x}_j)) \leq R_x^{\max}$ then
14         $\mathbf{x}_j = \texttt{rounding}(\mathbf{x}_j, R_x)$
15     end
16 end
17 Set: $\mathbf{x}[k] = \mathbf{x}_N$ and $\mathbf{P}[k] = \mathbf{P}_N$
```

In Algorithm 5, lines 5, 6 and 7 are of particular importance. In line 5, we compute the residual by subtracting the state from the output. Since we now update the state and covariance using blocks of measurements, the extracted values from the state will be a vector $\mathbf{x}_{j-1}(\mathbf{c}_j) \in \mathbb{R}^B$. This vector can be constructed using the described value extraction method in Chapter 2. In line 6, the matrix $\mathbf{S}_j \in \mathbb{R}^{B\times B}$ is extracted from the covariance matrix using an adaptation of the same value extraction method. Lastly, in line 7 the Kalman gain matrix $\mathbf{K}_j \in \mathbb{R}^{I\times B}$ is computed by the TTm matrix-matrix product of selected columns from the covariance matrix $\mathbf{P}_{j-1}(:, \mathbf{c}_j) \in \mathbb{R}^{I\times B}$ and the inverted matrix $\mathbf{S}_j^{-1}$. As described in Chapter 2, the extraction of columns from a matrix in TTm format requires using the singular value decomposition. This allows truncation of the ranks of the resulting TTm to a specified value without an additional rounding step. Furthermore, this returns a site-d orthogonal TTm. Looking at the singular values the cores of the TTm allows us to determine at what ranks they can be truncated. This reduces the computational load of the rounding procedure.

It must be noted that the ranks of the terms $\mathbf{K}_j \mathbf{v}_j$ and $\mathbf{K}_j \mathbf{S}_j \mathbf{K}_j^T$ are larger (B and B^2, respectively) compared to the terms $\mathbf{k}_j v_j$ and $\mathbf{k}_j s_j \mathbf{k}_j^T$ in the Element Update TNKF (both rank 1). This increases the computational load of each rounding step, mainly for the covariance matrix. In the next chapter, further investigation will be done by analyzing different Block Update TNKF settings and comparing the results with the Element Update TNKF.

4-2 Randomized TT-rounding

In [30] it is concluded that during computations to reconstruct corrupted frames, the TNKF algorithm spends around 40% of the total computation time on the rounding procedure. 10 - 15% of this is spent on rounding the state TT and 25 - 30% on the covariance TTm. The reason for these high percentages is that the ranks of the state TT and covariance TTm blow up quickly during TNKF computations. When adding two TT(m)'s, the ranks of both TT(m)'s also sum. During TT(m) multiplication, the ranks multiply. Since the goal of using a Tensor-Networked Kalman Filter is to use low-rank TT representations of the state and covariance, these ranks must be kept relatively low to keep computational load within reasonable bounds. This means the rounding function has to be called thousands of times per frame (depending on the missing pixel percentage) to ensure state TT and covariance TTm have low ranks.

In [10], different randomized rounding algorithms are introduced that aim to reduce the computational load of the rounding procedure. These algorithms are generalizations of randomized low-rank matrix approximation algorithms that can significantly reduce computational load compared to the deterministic TT-rounding algorithm. The first part of this section further discusses this by elaborating on the randomized SVD. Next, the randomized rounding algorithms are presented and compared with deterministic rounding in terms of computational complexity. Parts of this section are based on findings in [10]. For a more detailed description and discussion of the algorithms, the reader is referred to [10].

4-2-1 Randomized SVD

In recent literature, the randomized SVD is proven to reduce computational load while still maintaining acceptable performance compared the deterministic SVD [1, 3, 5]. In the randomized rounding algorithms, use is made of adaptations of the randomized SVD. The prototypical algorithm for the computation of the randomized SVD of a matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$, as proposed in [13], is given in Algorithm 6. A more detailed description of this algorithm is given in [13]. This algorithm computes a low-rank factorization $\mathbf{USV}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{S}$ is diagonal and non-negative. This algorithm computes a rank-(K) approximation of the SVD using randomization, which offers a reduction in computational load compared to deterministic computation of the SVD. The oversampling parameter s is required to make sure the range of the matrix $\mathbf{A}$ is sufficiently approximated. This value depends on the dimensions of $\mathbf{A}$ and its singular spectrum.

In line 1 of Algorithm 6, a matrix $\mathbf{O} \in \mathbb{R}^{J \times (K+s)}$ is generated with i.i.d. standard Gaussian random variables. This can be done using the `randn` command in MATLAB. Next, the power iteration $(\mathbf{AA}^T)^q$ is carried out to increase the decay of singular values while retaining the same left singular vectors of $\mathbf{A}$. Common values for the exponent are $q = 1$ or $q = 2$. Next, in line 3, a matrix $\mathbf{Q}$ is formed whose columns form an orthonormal basis for the range of $\mathbf{Y}$. After this, the SVD can be computed of the smaller matrix $\mathbf{B}$ along with some matrix multiplications.

Algorithm 6: Prototypical randomized SVD algorithm [13]

```
Data: A matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$, target number $K$, oversampling parameter $s$ and exponent $q$.
Result: Approximate rank-$(K+s)$ factorization $\mathbf{USV}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal
        and $\mathbf{S}$ is diagonal and non-negative.
1 Generate a matrix $\mathbf{O} \in \mathbb{R}^{J \times (K+s)}$ with i.i.d. standard Gaussian random variables
2 $\mathbf{Y} = (\mathbf{AA}^T)^q \mathbf{AO}$
3 Construct a matrix $\mathbf{Q}$ whose columns form an orthonormal basis for the range of $\mathbf{Y}$
4 $\mathbf{B} = \mathbf{Q}^T \mathbf{A}$
5 Compute the SVD of the smaller matrix: $\mathbf{B} = \mathbf{WSV}^T$
6 $\mathbf{U} = \mathbf{QW}$
```

4-2-2 Randomized rounding algorithms

In this section, three randomized rounding algorithms are presented that are based on randomized low-rank matrix approximation, as discussed in the previous section. Each randomized rounding algorithm is presented and compared to the deterministic rounding algorithm (Algorithm 3) in terms of operations.

1. *Orthogonalize-then-Randomize (OtR)* (Algorithm 7), which replaces the truncated SVD step in the standard TT-rounding algorithm with a generalization of the randomized SVD method. In line 5 of Algorithm 7, the left unfolding of the TT-core is multiplied with a matrix $\mathbf{\Omega}_n \in \mathbb{R}^{R_n \times \ell_n}$ that has i.i.d. standard Gaussian random variables. Next, the matrix $\mathbf{Q}$ is formed whose columns form an orthonormal basis for the range of $\mathbf{Z}_n$ using the thin QR factorization. After this, the core of the rounded TT is computed and saved, and the next core is prepared for the following iteration. Lines 5, 6 and 7 in Algorithm 7 are based on lines 2, 3 and 4 in Algorithm 6 in which the same principles are applied. Out of all presented algorithms, OtR resembles deterministic rounding (Algorithm 3) the most as it also uses orthogonalization and compression. Since the orthogonalization procedure has a relatively large computational cost, it is shown in [10] to have a relatively limited speedup compared to deterministic rounding. This will be discussed further in this section.

2. *Randomize-then-Orthogonalize (RtO)* (Algorithm 8), which uses randomized projections of each core by computing contractions with a TT-tensor that contains random entries. These projections form an accurate approximation of the original TT-cores, but allow performing the compression sweep on smaller matrices. It will be clear in the next subsection, that the computational cost of Algorithm 7 is dominated by the first orthogonalization phase of the algorithm. In order to tackle this, Algorithm 8 uses randomization before the compression sweep. This avoids the expensive orthogonalization of the original TT-tensor to reduce computational cost. Firstly, a random Gaussian tensor is generated $\boldsymbol{\mathcal{R}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ of the same dimensions as the to be rounded tensor in TT-format. Using the `randn` command in MATLAB, we ensure this tensor's entries to be random, independent, normally distributed entries with mean of 0 and variance of 1. In [10], it is suggested to multiply the variance with a normalization term of $1/(\ell_{n-1})I_n\ell_n$ for $n = 1\ldots, d$ to prevent overflow in the rounding computations. This is not necessary

Algorithm 7: Randomized TT-Rounding: Orthogonalize-then-Randomize [10]

```
Data: A $d$-dimensional tensor $\boldsymbol{\mathcal{Y}}$ in TT format $\boldsymbol{\mathcal{Y}} = \langle\!\langle \boldsymbol{\mathcal{Y}}^{(1)}, \boldsymbol{\mathcal{Y}}^{(2)}, \dots, \boldsymbol{\mathcal{Y}}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$
      with ranks $\mathbf{r} = [R_0, \dots, R_d]$ and target TT ranks $\boldsymbol{\ell} = [\ell_0, \dots, \ell_d]$
Result: A $d$-dimensional tensor $\boldsymbol{\mathcal{X}}$ in TT format
      $\boldsymbol{\mathcal{X}} = \langle\!\langle \boldsymbol{\mathcal{X}}^{(1)}, \boldsymbol{\mathcal{X}}^{(2)}, \dots, \boldsymbol{\mathcal{X}}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with ranks $\boldsymbol{\ell} = [\ell_0, \dots, \ell_d]$
$\boldsymbol{\mathcal{Y}}$ = orthogonalization($\boldsymbol{\mathcal{Y}}$, 1) ;                         // Algorithm 2
$\boldsymbol{\mathcal{X}}^{(1)} = \boldsymbol{\mathcal{Y}}^{(1)}$
for $n = 1, 2, \dots, d-1$ do
    $\mathbf{Z}_n$ = reshape($\boldsymbol{\mathcal{X}}^{(n)}$, [], $R_n$)
    $\mathbf{Y}_n = \mathbf{Z}_n \boldsymbol{\Omega}_n$ ;                      // random matrix $\boldsymbol{\Omega}_n \in \mathbb{R}^{R_n \times \ell_n}$
    [$\mathbf{X}_n$, ~] = qr($\mathbf{Y}_n$, 0) ;                     // thin QR factorization
    $\mathbf{M}_n = \mathbf{X}_n^T \mathbf{Z}_n$
    $\boldsymbol{\mathcal{X}}^{(n+1)} = \mathbf{M}_n$(reshape($\boldsymbol{\mathcal{Y}}^{(n+1)}$, $R_n$, []))
    $\boldsymbol{\mathcal{X}}^{(n)}$ = reshape($\mathbf{X}_n$, $\ell_{n-1}$, $I_n$, $\ell_n$)
end
```

for the rounding in the TNKF, as the ranks remain relatively small. After forming this Gaussian tensor, it is put into TT-format using the TT-SVD algorithm, yielding the TT $\boldsymbol{\mathcal{R}} = \langle\!\langle \boldsymbol{\mathcal{R}}^{(1)}, \boldsymbol{\mathcal{R}}^{(2)}, \dots, \boldsymbol{\mathcal{R}}^{(d)} \rangle\!\rangle$ (where $\boldsymbol{\mathcal{R}}^{(n)} \in \mathbb{R}^{\ell_{n-1} \times I_n \times \ell_n}$ for $n = 1, \dots, d$). The TT-SVD algorithm allows us to specify its ranks, which are set as desired $\boldsymbol{\ell} = [\ell_0, \dots, \ell_d]$. The right-to-left partial contractions are computed using a similar technique as the TT inner product described in Chapter 2. The algorithm that is used for this process (Algorithm 16), is listed in Appendix A. Next, the compression is carried out similarly as in Algorithm 7, however now using the partial contraction matrices $\mathbf{W}_n$ instead of random matrices $\boldsymbol{\Omega}_n$. Since the Randomize-then-Orthogonalize algorithm produces a left-orthogonal tensor $\boldsymbol{\mathcal{X}}$, this structure can be exploited to truncate the ranks further if necessary while skipping the orthogonalization phase.

3. *Two-Sided-Randomization (TSR)* (Algorithm 9), which eliminates the need for separate orthogonalization and compression sweeps. Here, products are computed with two random tensors followed by a compression step. Contrary to the first two randomized rounding algorithms, this algorithm is based on the generalized Nyström method [23]. This method avoids the orthogonalization step when computing a low-rank approximation using a two-sided randomization approach. Let us define two matrices $\boldsymbol{\Omega} \in \mathbb{R}^{n \times s}$ and $\boldsymbol{\Phi} \in \mathbb{R}^{t \times m}$ with Gaussian random entries and where $r \le s \le \min\{m, n\}$. Here, r denotes the target rank. A low-rank matrix approximation for $\mathbf{X}$ is then computed as

$$\mathbf{X} \approx \mathbf{Y}(\boldsymbol{\Psi}\mathbf{X}\boldsymbol{\Omega})^{\dagger}\mathbf{Z} \tag{4-1}$$

where $\mathbf{Y} = \mathbf{X}\boldsymbol{\Omega}$ and $\mathbf{Z} = \boldsymbol{\Phi}\mathbf{X}$. To implement the pseudo-inverse (denoted by †), it is suggested in [23] compute the QR factorization $\boldsymbol{\Phi}\mathbf{X}\boldsymbol{\Omega} = \mathbf{Q}\mathbf{R}$. From this, the low-rank approximation $\left(\mathbf{Y}\mathbf{R}^{-1}\right)\left(\mathbf{Q}^{\top}\mathbf{Z}\right)$ can be obtained. In [23], it is recommended to set the sketch parameters to $s = r$ and $t = \lceil 1.5r \rceil$. An adaptation of this generalized Nyström low-rank matrix approximation method is used in the TSR algorithm.

Algorithm 8: Randomized TT-Rounding: Randomize-then-Orthogonalize [10]

```
Data: A $d$-dimensional tensor $\boldsymbol{\mathcal{Y}}$ in TT format $\boldsymbol{\mathcal{Y}} = \langle\!\langle \boldsymbol{\mathcal{Y}}^{(1)}, \boldsymbol{\mathcal{Y}}^{(2)}, \ldots, \boldsymbol{\mathcal{Y}}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$
      with ranks $\mathbf{r} = [R_0, \ldots, R_d]$ and target TT ranks $\boldsymbol{\ell} = [\ell_0, \ldots, \ell_d]$
Result: A $d$-dimensional tensor $\boldsymbol{\mathcal{X}}$ in TT format
      $\boldsymbol{\mathcal{X}} = \langle\!\langle \boldsymbol{\mathcal{X}}^{(1)}, \boldsymbol{\mathcal{X}}^{(2)}, \ldots, \boldsymbol{\mathcal{X}}^{(d)} \rangle\!\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with ranks $\boldsymbol{\ell} = [\ell_0, \ldots, \ell_d]$
1  Generate random Gaussian TT-tensor $\boldsymbol{\mathcal{R}}$ with target TT ranks $\boldsymbol{\ell} = [\ell_0, \ldots, \ell_d]$
2  $\{\mathbf{W}_1, \ldots, \mathbf{W}_{d-1}\} = \texttt{PartialContractionsRL}(\boldsymbol{\mathcal{Y}}, \boldsymbol{\mathcal{R}})$
3  $\boldsymbol{\mathcal{X}}^{(1)} = \boldsymbol{\mathcal{Y}}^{(1)}$
4  for $n = 1, 2, \ldots, d-1$ do
5      $\mathbf{Z}_n = \texttt{reshape}(\boldsymbol{\mathcal{X}}^{(n)}, [], R_n)$
6      $\mathbf{Y}_n = \mathbf{Z}_n \mathbf{W}_n$          // partial contraction matrix $\mathbf{W}_n \in \mathbb{R}^{R_n \times \ell_n}$
7      $[\mathbf{X}_n, \sim] = \texttt{qr}(\mathbf{Y}_n, 0)$          // thin qr factorization
8      $\mathbf{M}_n = \mathbf{X}_n^T \mathbf{Z}_n$
9      $\boldsymbol{\mathcal{X}}^{(n+1)} = \mathbf{M}_n(\texttt{reshape}(\boldsymbol{\mathcal{Y}}^{(n+1)}, R_n, []))$
10     $\boldsymbol{\mathcal{X}}^{(n)} = \texttt{reshape}(\mathbf{X}_n \ell_{n-1}, I_n, \ell_n)$
11 end
```

In the TSR algorithm (Algorithm 9), two random Gaussian TT-tensors $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{\mathcal{R}}$ need to be defined. This is done using the method described for the RtO algorithm. $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{\mathcal{R}}$ are used to compute the left-to-right and right-to-left partial contraction matrices, respectively. The algorithms used for this (Algorithm 15 and 16) are listed in Appendix A. After the partial contraction matrices have been computed, the randomization phase is completed (lines 1 - 4). Then, using the SVD of a product of partial contraction matrices, the right and left factor matrices $\mathbf{L}_n$ and $\mathbf{R}_n$ are computed (lines 5 - 9), which are used to construct the solution TT $\boldsymbol{\mathcal{X}}$. Contrary to Algorithm 8, the Two-Sided-Randomization approach does not produce an orthogonal tensor. Using restructuring, orthogonality of the cores can be achieved again yielding a TT that can be rounded further while skipping orthogonalization.

A variable in the TSR algorithm is the choice of the oversampling parameters, which are the ranks $\boldsymbol{\rho} = [\rho_0, \ldots, \rho_d]$ of the Gaussian TT-tensor $\boldsymbol{\mathcal{R}}$. In the numerical experiments in [10], the oversampling is set to $\rho_n = [1.5\ell_n]$ following the suggestion made in [23]. While testing the algorithm, it became clear this does not hold for the application of streaming video completion, as further discussed in subsection 5-3-4. The oversampling must be set higher to achieve similar performance to the other rounding algorithms. A downside of this is that this increases the size of the right-to-left partial contraction matrices of the original TT(m) and $\boldsymbol{\mathcal{R}}$, slowing down the rounding procedure.

Algorithm 9: Randomized TT-Rounding: Two-Sided-Randomization [10]

```
Data: A $d$-dimensional tensor $\boldsymbol{\mathcal{Y}}$ in TT format $\boldsymbol{\mathcal{Y}} = \langle\langle \boldsymbol{\mathcal{Y}}^{(1)}, \boldsymbol{\mathcal{Y}}^{(2)}, \ldots, \boldsymbol{\mathcal{Y}}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$
      with ranks $\mathbf{r} = [R_0, \ldots, R_d]$ and target TT ranks $\boldsymbol{\ell} = [\ell_0, \ldots, \ell_d]$ and
      $\boldsymbol{\rho} = [\rho_0, \ldots, \rho_d]$
Result: A $d$-dimensional tensor $\boldsymbol{\mathcal{X}}$ in TT format
      $\boldsymbol{\mathcal{X}} = \langle\langle \boldsymbol{\mathcal{X}}^{(1)}, \boldsymbol{\mathcal{X}}^{(2)}, \ldots, \boldsymbol{\mathcal{X}}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with ranks $\boldsymbol{\ell} = [\ell_0, \ldots, \ell_d]$
1  Generate random Gaussian TT-tensor $\boldsymbol{\mathcal{L}}$ with target TT ranks $\boldsymbol{\ell} = [\ell_0, \ldots, \ell_d]$
2  Generate random Gaussian TT-tensor $\boldsymbol{\mathcal{R}}$ with target TT ranks $\boldsymbol{\rho} = [\rho_0, \ldots, \rho_d]$
3  $\{\mathbf{W}_1^{\mathrm{L}}, \ldots, \mathbf{W}_{d-1}^{\mathrm{L}}\}$ = PartialContractionsLR($\boldsymbol{\mathcal{Y}}, \boldsymbol{\mathcal{L}}$)
4  $\{\mathbf{W}_1^{\mathrm{R}}, \ldots, \mathbf{W}_{d-1}^{\mathrm{R}}\}$ = PartialContractionsRL($\boldsymbol{\mathcal{Y}}, \boldsymbol{\mathcal{R}}$)
5  for $n = 1, 2, \ldots, d-1$ do
6      $[\mathbf{U}_n, \boldsymbol{\Sigma}_n, \mathbf{V}_n]$ = svd($\mathbf{W}_n^{\mathrm{L}} \mathbf{W}_n^{\mathrm{R}}$, 'econ') ;          // economy sized SVD
7      $\mathbf{L}_n = \mathbf{W}_n^{\mathrm{R}} \mathbf{V}_n (\boldsymbol{\Sigma}_n^{\dagger})^{1/2}$
8      $\mathbf{R}_n = (\boldsymbol{\Sigma}_n^{\dagger})^{1/2} \mathbf{U}_n^T \mathbf{W}_n^{\mathrm{L}}$
9  end
10 $\boldsymbol{\mathcal{X}}^{(1)}$ = reshape($\boldsymbol{\mathcal{Y}}^{(1)}, [], R_1$)$\mathbf{L}_1$
11 for $n = 2, 3, \ldots, d-1$ do
12     $\boldsymbol{\mathcal{X}}^{(n)} = \mathbf{R}_{n-1}$(reshape(reshape($\boldsymbol{\mathcal{Y}}^{(n)}, [], R_n \mathbf{L}_n$), $R_{n-1}$, []))
13     $\boldsymbol{\mathcal{X}}^{(n)}$ = reshape($\boldsymbol{\mathcal{X}}^{(n)}, \ell_{n-1}, I_n, \ell_n$)
14 end
15 $\boldsymbol{\mathcal{X}}^{(d)} = \mathbf{R}_{d-1}$(reshape($\boldsymbol{\mathcal{Y}}^{(d)}, R_{d-1}$, []))
```

4-2-3 Computational cost comparison

In order to compare the considered algorithms in terms of computational cost, we sum the cost of all operations carried out in each algorithm. Below, the algorithms are listed with their corresponding computational cost analysis. Here, N is the number of TT-cores of the original TT which has the ranks $\mathbf{r} = [1, R, \ldots, R, 1]$. The rounded TT has the desired ranks $\boldsymbol{\ell} = [1, \ell, \ldots, \ell, 1]$. I is the core size (which is assumed constant) of the TT. Since the order of magnitude of the computational cost is the same for each algorithm, we express the cost of each operation in the number of floating-point operations (flops). Using this common measure for computational cost allows us to compare the algorithms more in-depth. A flop can denote one addition, subtraction, multiplication or division of floating-point numbers. The amount of flops of each matrix-matrix product, for example, is defined by the sizes of both matrices. If $\mathbf{A} \in \mathbb{R}^{M\times N}$ and $\mathbf{B} \in \mathbb{R}^{N\times P}$, then the matrix-matrix product is $\mathbf{AB} = \mathbf{A}(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_P) = (\mathbf{Ab}_1, \mathbf{Ab}_2, \ldots, \mathbf{Ab}_P)$. Each product $\mathbf{Ab}_i$ costs $2MN$ flops, assuming MN multiplications and MN summations take place. This gives a total flop count of $2MNP$ for the matrix-matrix multiplication.

1. *Orthogonalize-then-Randomize* (Algorithm 7). In line 1, orthogonalization takes place, which has a computational cost of $(N-2)I(5R^3)$ flops. Next the compression loop is carried out, starting with a matrix multiplication of the left unfolding of the TT core $\mathbf{Z_n}$ of size $I\ell \times R$ with the random matrix $\boldsymbol{\Omega}_n$ of size $R \times \ell$. This results in a cost of $2IR\ell^2$ flops. In line 6, the thin QR factorization of the resulting matrix $\mathbf{Y}_n$ is computed, having a cost of $4I\ell^3 + \mathcal{O}(\ell^3)$ flops. Lastly, in lines 7 and 8, two matrix-matrix multiplications are carried out costing $IR\ell^2$ and $2IR^2\ell$ flops, respectively. This gives a total cost described in (4-2).

$$\begin{aligned} C_{\text{OtR}} &= (N-2)I(5R^3) + (N-2)\left(2IR^2\ell + 4IR\ell^2 + 4I\ell^3\right) + \mathcal{O}\left(IR^2 + NR^3\right) \\ &= I(N-2)\cdot\left(5R^3 + 2R^2\ell + 4R\ell^2 + 4\ell^3\right) + \mathcal{O}\left(IR^2 + NR^3\right) \quad \text{flops} \end{aligned} \tag{4-2}$$

2. *Randomize-then-Orthogonalize* (Algorithm 8). In line 2, the partial contractions algorithm is executed with a cost of $(N-2)I(2R^2\ell + 2R\ell^2)$ flops. Next, the compression loop is iterated, identical to the loop from Algorithm 7. This illustrates that the only difference between these two algorithms is in the orthogonalization phase. This gives a total cost described in (4-3).

$$\begin{aligned} C_{\text{RtO}} &= (N-2)I(2R^2\ell + 2R\ell^2) + (N-2)\left(2IR^2\ell + 4IR\ell^2 + 4I\ell^3\right) + \mathcal{O}\left(IR^2 + NR^3\right) \\ &= I(N-2)\cdot\left(4R^2\ell + 6R\ell^2 + 4\ell^3\right) + \mathcal{O}\left(IR^2 + NR^3\right) \quad \text{flops} \end{aligned} \tag{4-3}$$

3. *Two-Sided-Randomization* (Algorithm 9.). In lines 3 and 4, the partial contraction algorithm is executed two times resulting in a cost of $(N-2)I(4R^2\ell + 4R\ell^2)$ flops if we assume $\boldsymbol{\ell} = \boldsymbol{\rho}$. In the loop from line 5 - 9, the two matrix multiplications are the relevant contributors to the cost. The first multiplication has matrices of size $\ell \times R$ and $R \times IR$ and the second $\ell \times R$ and $R \times I\ell$, yielding a combined cost of $(N-2)(2IR^2\ell + 2IR\ell^2)$

flops. The cost of the final for loop is $\mathcal{O}(NR^2)$ flops. This gives a total cost described in (3).

$$\begin{aligned} C_{\mathrm{TSR}} &= (N-2)I(4R^2\ell + 4R\ell^2) + (N-2)\left(2IR^2\ell + 2IR\ell^2\right) + \mathcal{O}\left(NR^2\right) \\ &= I(N-2)\cdot\left(6R^2\ell + 6R\ell^2\right) + \mathcal{O}\left(IR\ell + NR^2\right) \quad \text{flops} \end{aligned} \tag{4-4}$$

In Table 4-1, the derived computational costs for the randomized algorithms compared to deterministic TT-rounding are summarized where the lower order terms are discarded. From the first two rows, it is observed that the orthogonalization sweep dominates the computational cost of the deterministic TT-rounding approach for large ranks. When we simplify the terms using the ratio between target rank and original rank of the tensor $\beta = \ell/R$, we get a clearer view of the computational cost comparison. For high values of β (β close to 1), no significant speedup will result from using randomized rounding algorithms. However, when the target rank is a smaller fraction of the original rank, a large speedup can be achieved when looking at the simplified computational costs. The speedup becomes significant for the Randomize-then-Orthogonalize and the Two-Sided-Randomization algorithm for relatively low ratio values ($\beta << 1$).

TT Algorithm	**Computational cost (flops)**	**Simplified cost (flops)**
Orth	$(N-2)I(5R^3)$	–
Contr	$(N-2)I(2R^2\ell + 2R\ell^2)$	–
TT-rounding	$(N-2)I(5R^3 + 6R^2\ell + 2R\ell^2)$	$(N-2)IR^3(5 + 6\beta + 2\beta^2)$
Orth-then-Rand	$(N-2)I(5R^3 + 2R^2\ell + 4R\ell^2 + 4\ell^3)$	$(N-2)IR^3(5 + 2\beta + 4\beta^2 + 4\beta^3)$
Rand-then-Orth	$(N-2)I(5R^2\ell + 4R^2\ell + 4\ell^3)$	$(N-2)IR^3(4\beta + 6\beta^2 + 4\beta^3)$
Two-Sided-Rand	$(N-2)I(6R^2\ell + 6R\ell^2)$	$(N-2)IR^3(6\beta + 6\beta^2)$

Table 4-1: Computational costs of randomized algorithms compared to deterministic TT-rounding [10].

4-3 Simplification of the TNKF update

In the TNKF algorithm (Algorithm 4), the state and covariance are updated in line 8 and 9 (4-5).

$$\begin{aligned}\mathbf{x}_j &= \mathbf{x}_{j-1} + \mathbf{k}_j v_j \\ \mathbf{P}_j &= \mathbf{P}_{j-1} - \mathbf{k}_j s_j \mathbf{k}_j^T\end{aligned} \tag{4-5}$$

Because of the way the TNKF is built, the whole state TT and covariance TTm are updated for each measurement resulting in thousands of updates per reconstructed frame (depending on the missing pixel %). Since each update is a tiny 'step' in reconstructing a frame, it is interesting to see to what extent the state vector and covariance matrix change during these updates. If it turns out that these changes are minimal or only occur at certain moments, computations could be further simplified.

To analyze this, the principal angles (definition 4-3.1) between the orthonormal bases of the state $\mathbf{x}_{j-1}$ and the summed term $\mathbf{k}_j v_j$ can be evaluated. The same can be done for the original covariance $\mathbf{P}_{j-1}$ and the subtracted term $\mathbf{k}_j s_j \mathbf{k}_j^T$. These angles are a measure for the amount of 'change' in the state TT and covariance TTm during the TNKF update. The larger the principal angles are, the more the added or subtracted term differs from the state TT or covariance TTm.

Definition 4-3.1 (Principal angles between orthonormal bases [36, p. 3])**.** Let the columns of $\mathbf{X} \in \mathbb{R}^{M\times P}$ and $\mathbf{Y} \in \mathbb{R}^{N\times Q}$ form orthonormal bases for the subspaces $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$, respectively. Let the SVD of $\mathbf{X}^T\mathbf{Y}$ be $\mathbf{USV}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices and $\mathbf{S}$ is a $P \times Q$ diagonal matrix with the real diagonal elements $s_1(\mathbf{X}^T\mathbf{Y}), \dots, s_m(\mathbf{X}^T\mathbf{Y})$ in non-decreasing order with $M = \min(P, Q)$. Then the following holds

$$\cos \boldsymbol{\theta}^{\uparrow}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}}) = \mathbf{s}\left(\mathbf{X}^T\mathbf{Y}\right) = \left[s_1\left(\mathbf{X}^T\mathbf{Y}\right), \dots, s_m\left(\mathbf{X}^T\mathbf{Y}\right)\right] \tag{4-6}$$

where $\boldsymbol{\theta}^{\uparrow}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}})$ denotes the vector of principal angles between $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ arranged in nondecreasing order and $\mathbf{s}(\mathbf{A})$ denotes the vector of singular values of matrix $\mathbf{A}$. The up arrow in the term $\boldsymbol{\theta}^{\uparrow}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}})$ stands for the nondecreasing order.

To compute the principal angles, orthonormal bases of the desired terms are required. These bases can be computed by firstly orthogonalizing the considered TT(m) to site-d using Algorithm 2. As elaborated in Chapter 2, this results in an orthogonal TT(m) where the total norm is in the last core. The left unfoldings of cores 1 to $d-1$ of the TT(m) have orthonormal rows and columns, as shown in (2-15). This means that if we contract these cores, this yields an orthonormal matrix. After orthogonalization, we call these cores 1 to $d-1$ in TT(m) format $\boldsymbol{\mathcal{G}}$. After computing the cores of $\boldsymbol{\mathcal{G}}$, the outer product is taken with an identity matrix of size equal to the last core dimension, which is I_d for the state and I_dJ_d for the covariance. This is done to ensure the correct size of the formed orthonormal bases. In the case of the state, we can rewrite the TT to (4-7). This format expresses the multilinearity of a TT, i.e. by fixing the cores 1 to $d-1$, the TT becomes linear in the remaining core. This property can be used to analyze which other TT's could be represented by only changing the last core. Here, the matrix $\mathbf{U}$ denotes the orthonormal basis and is not to be mistaken with the $\mathbf{U}$ matrix of the SVD.

$$\mathbf{x} = \mathbf{U}\mathbf{x}_d \tag{4-7}$$

In (4-7), the orthonormal matrix $\mathbf{U} \in \mathbb{R}^{I_1 I_2 \cdots I_d \times R_{d-1} I_d R_d}$ is multiplied with the vectorized last core of the TT $\mathbf{x}_d \in \mathbb{R}^{R_{d-1} I_d R_d}$ to form the original vector $\mathbf{x} \in \mathbb{R}^{I_1 I_2 \cdots I_d}$. This shows that if the orthonormal basis remains (close to) the same basis through TNKF iterations, the vector in TT format can be described by the product of this orthonormal basis and the vectorized last core. If we use this to rewrite the state update equation in (4-5), we get the expression in (4-8).

$$\mathbf{x}_j = \mathbf{U}_{\mathbf{x_{j-1}}} \mathbf{x}_d + \mathbf{U}_{\mathbf{k}_j v_j} \mathbf{z}_d \tag{4-8}$$

where $\mathbf{U}_{\mathbf{x_{j-1}}} \in \mathbb{R}^{I_1 I_2 \cdots I_d \times R_{d-1} I_d R_d}$ is the orthonormal basis for the state $\mathbf{x}_{j-1}$ and $\mathbf{U}_{\mathbf{k}_j v_j} \in \mathbb{R}^{I_1 I_2 \cdots I_d \times K_{d-1} I_d K_d}$ is the orthonormal basis for the added term $\mathbf{k}_j v_j$ (which has ranks $[K_0, \ldots, K_d]$). $\mathbf{x}_d \in \mathbb{R}^{R_{d-1} I_d R_d}$ denotes the vectorized last core of the state TT and $\mathbf{z}_d \in \mathbb{R}^{K_{d-1} I_d K_d}$ denotes the vectorized last core of the added term.

If we can show that the principal angles between the orthonormal bases $\mathbf{U}_{\mathbf{x_{j-1}}}$ and $\mathbf{U}_{\mathbf{k}_j v_j}$ are (close to) zero, then the relation in (4-8) can be rewritten to (4-9).

$$\mathbf{x}_j \approx \mathbf{U}_{\mathbf{x_{j-1}}} (\mathbf{x}_d + \begin{bmatrix} \mathbf{z}_d \\ \mathbf{0} \end{bmatrix}) \tag{4-9}$$

where $\mathbf{0}$ is a zero vector of size $R_{d-1} I_d - K_{d-1} I_d$, given that $R_{d-1} > K_{d-1}$. If this relation holds, it means that the state TT can be updated by doing an elementwise addition of the last core of the TT. This does not increase the ranks of the TT, which could result in a significant reduction of the TNKF algorithm's computational load. This analysis can be done in a similar manner for the covariance. The expressions for the orthonormal bases of the state and covariance are shown in (4-10) (4-11). In Figure 4-1, the tensor diagram is shown for the orthonormal basis of the state vector, $\mathbf{U}_{\mathbf{x}_{j-1}}$.

$$\begin{aligned} \mathbf{U}_{\mathbf{x}_{j-1}} &= \mathbf{I}_{I_d} \otimes \mathcal{G}_{\mathbf{x}_{j-1}} \\ \mathbf{U}_{\mathbf{k}_j v_j} &= \mathbf{I}_{I_d} \otimes \mathcal{G}_{\mathbf{k}_j v_j} \end{aligned} \tag{4-10}$$

$$\begin{aligned} \mathbf{U}_{\mathbf{P}_{j-1}} &= \mathbf{I}_{I_d J_d} \otimes \mathcal{G}_{\mathbf{x}_{j-1}} \\ \mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j} &= \mathbf{I}_{I_d J_d} \otimes \mathcal{G}_{\mathbf{k}_j s_j \mathbf{k}_j} \end{aligned} \tag{4-11}$$

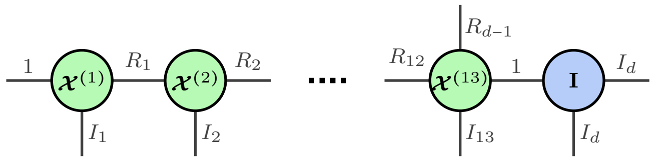


Figure 4-1: Tensor diagram for the orthonormal basis $\mathbf{U}_{\mathbf{x}_{j-1}}$ where $\mathcal{G}_{\mathbf{x}_{j-1}} = \langle\langle \mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \ldots, \mathcal{X}^{(13)} \rangle\rangle$ ($d = 14$).

When the orthonormal bases have been computed, we can check their principal angles before addition/subtraction takes place. For these angles to be zero, the relations in (4-12) must hold for the state and covariance as defined in definition 4-3.1. This can be checked by computing the singular values of $\mathbf{U}^T_{\mathbf{x}_{j-1}} \mathbf{U}_{\mathbf{k}_j v_j}$ and $\mathbf{U}_{\mathbf{P_{j-1}}}{}^T \mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j}$. If the considered equation in (4-12) holds, the resulting singular value vector will contain only ones. The more the angles between the orthonormal bases differ, the lower the singular values in this vector will be.

$$\begin{aligned}\mathbf{s}(\mathbf{U}_{\mathbf{x}_{j-1}}^T\mathbf{U}_{\mathbf{k}_j v_j}) &= \mathbf{1}_{K_{d-1}I_d}\\ \mathbf{s}(\mathbf{U}_{\mathbf{P}_{j-1}}^T\mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j}) &= \mathbf{1}_{K_{d-1}I_d J_d}\end{aligned} \tag{4-12}$$

In (4-12), $\mathbf{U}_{\mathbf{x}_{j-1}}^T\mathbf{U}_{\mathbf{k}_j v_j}$ is of size $R_{d-1}I_d \times K_{d-1}I_d$ and $\mathbf{U}_{\mathbf{P}_{j-1}}^T\mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j}$ is of size $R_{d-1}I_d J_d \times K_{d-1}I_d J_d$. Here, R_{d-1} is the rank between the last two cores of the state TT or covariance TTm, K_{d-1} is the rank between the last two cores of the added/subtracted term. The term $\mathbf{1}_{K_{d-1}I_d}$ represents a vector of size $K_{d-1}I_d$ with only ones as elements. The tensor network diagrams corresponding to these products are shown in Figure 4-2 (state) and Figure 4-3 (covariance).

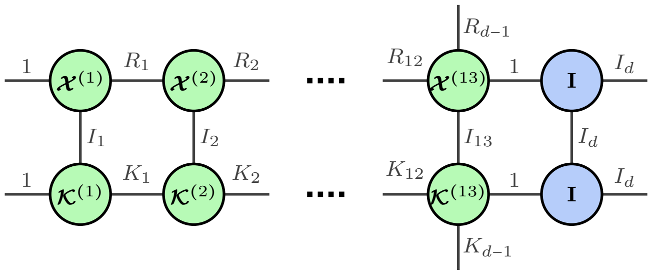


Figure 4-2: Tensor diagram for the product of $\mathbf{U}_{\mathbf{x}_{j-1}}^T\mathbf{U}_{\mathbf{k}_j v_j}$ ($d = 14$).

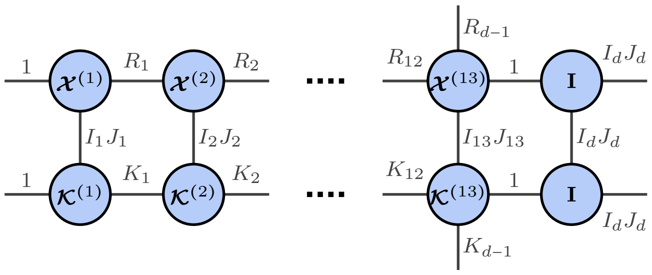


Figure 4-3: Tensor diagram for the product of $\mathbf{U}_{\mathbf{P}_{j-1}}^T\mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j}$ ($d = 14$).

It is also interesting to see to what extent the state and covariance change during iterations. This can be analyzed by computing the principal angles between the state and covariance at time $j-1$ with the state and covariance at time j and checking how closely the equations in (4-13) hold. Again the singular values can be investigated to verify this.

$$\begin{aligned}\mathbf{s}(\mathbf{U}_{\mathbf{x}_{j-1}}^T\mathbf{U}_{\mathbf{x}_j}) &= \mathbf{1}_{R_{d-1}I_d}\\ \mathbf{s}(\mathbf{U}_{\mathbf{P}_{j-1}}^T\mathbf{U}_{\mathbf{P}_j}) &= \mathbf{1}_{R_{d-1}I_d J_d}\end{aligned} \tag{4-13}$$

If the above relations hold, this means the direction of the orthonormal bases of the state and covariance do not change during iterations. In the next chapter, further investigation will be done if this is the case and how this property could be exploited.

Chapter 5

Results & Discussion

In this chapter, all proposed speedup methods are analyzed in terms of computational speed and performance. Firstly, the experiments are introduced by discussing the default TNKF settings and giving the performance and computational speed indicators. Next, the Block Update TNKF is compared to the Element Update TNKF and the design parameters are investigated. In the third section, all randomized rounding algorithms are compared to deterministic rounding and conclusions are drawn about the trade-offs between computational speed and performance. Lastly, an analysis is done to see to what extent the state and covariance change throughout iterations and an attempt is made to further simplify computations. All previously mentioned research questions will be discussed and answered throughout this chapter.

5-1 Experiment introduction

5-1-1 Video data and default settings

During experimentation, two different video data sets will be used: Campus [34] and Grand Central Station [4] (Figure 5-1), which both have a frame size of 480×720 pixels. Examples of a frame of both data sets are shown in Figure 5-1.

Both data sets have been selected since they have different properties. The Campus video has larger moving parts in the 'front' of the frame and smaller moving parts in the 'back' of the frame. Furthermore, it is in color, allowing us to assess the investigated methods' ability to reconstruct colors correctly. Given the frame size of both data sets, the quantization parameters (defined in Chapter 2) must be chosen, as elaborated in Chapter 3. These are set at $M_i = \{5, 3, 2, 2, 2, 2, 2\}$ for the height and $N_i = \{2, 2, 2, 2, 3, 3, 5\}$ for the width. These parameters have been chosen as small as possible to achieve maximum compression rate.

In [30] it is shown that for high percentages of missing pixels (95%), the TNKF outperforms other reconstruction methods such as the PLMS algorithm [32] in reconstruction quality. For lower percentages (75%), the PLMS algorithm performs better than the TNKF. For this

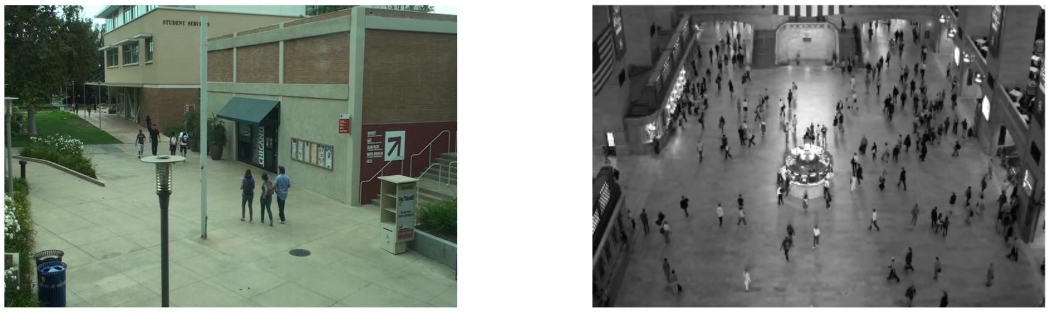


Figure 5-1: Video datasets. Campus [34] in color (left) and Grand Central Station [4] in gray (right).

reason, only high percentages of missing pixels are considered in this analysis ($\geq 95\%$). To test the ability of the considered methods to reconstruct frames with a high percentage of missing pixels, this value is set to 99% as default for the Campus data set. The Grand Central Station video has a lot of small moving parts in the frame as numerous persons are walking through the station. Since the moving parts are smaller, the default missing pixel percentage is set to 95% for this data set. This gives the TNKF more information to base the reconstruction on. This data set is in grayscale, meaning less information is available for the TNKF.

In [30], the influence of the state and covariance rank parameters on the computational speed were analyzed. Generally, a higher desired state rank $R_{\mathbf{x}}$ resulted in better reconstruction quality but significantly increased the computation time. $R_{\mathbf{x}} = 30$ was found to be a value that gave sufficient performance while maintaining relatively fast computations. The desired covariance rank must be set at $R_{\mathbf{P}} = 1$ to ensure stability and to keep computation times within reasonable bounds. After some testing, the default values for the maximum state rank and maximum covariance rank are set at $R_{\mathbf{x}}^{\max} = 50$ and $R_{\mathbf{P}}^{\max} = 5$ since these proved to be (close to) the fastest settings.

For the process noise rank and bandwidth, the values $R_{\mathbf{W}} = 5$ and $\alpha = 10$ are taken since this ensures the added process noise is full rank. In [30] it is shown that for videos with small moving parts, this value for the bandwidth is a good choice as the correlation between pixels drops to relatively low values within this bandwidth. The initial state $\mathbf{x}_0$ is set to the last uncorrupted frame, as this gives good results when the moving objects in the frame are relatively small, as is the case for the Campus and Grand Central Station data set. The default rounding algorithm is set to deterministic rounding (Algorithm 3). All default TNKF settings are summarized in Table 5-1. Examples of the frames' corrupted foreground of both data sets are shown in Appendix B. Here, the missing pixel percentage is set to $\gamma = 0.99$ for the Campus data set and $\gamma = 0.95$ for the Grand Central Station data set.

Parameter	Symbol	Setting
Frame height quantization parameters	M_i	{5,3,2,2,2,2,2}
Frame width quantization parameters	N_i	{2,2,2,2,3,3,5}
Missing pixel %	γ	0.99 (Campus) / 0.95 (Station)
Desired state rank	$R_{\mathbf{x}}$	30
Maximum state rank	$R_{\mathbf{x}}^{\max}$	50
Desired covariance rank	$R_{\mathbf{P}}$	1
Desired covariance rank	$R_{\mathbf{P}}^{\max}$	5
Process noise rank	$R_{\mathbf{W}}$	5
Bandwidth	α	10
Initial state	$\mathbf{x}_0$	'LastFrame'
Rounding algorithm	-	'Deterministic'

Table 5-1: Default (Block Update) TNKF experimentation settings.

5-1-2 Performance and computational speed indicators

As earlier mentioned, the performance of the algorithms refers to the accuracy of their reconstructed frames. To assess the performance of the researched methods, the relative error of the reconstructed frames will be computed. For grayscale videos, this error ($\varepsilon[k]$) is computed by dividing the Frobenius norm of the difference between the correct frame $\mathbf{X}[k] \in \mathbb{R}^{M\times N}$ and the estimated frame $\widehat{\mathbf{X}}[k] \in \mathbb{R}^{M\times N}$ at time k by the Frobenius norm of the correct frame, as shown in (5-1).

$$\varepsilon[k] = \frac{\|\mathbf{X}[k] - \widehat{\mathbf{X}}[k]\|_F}{\|\mathbf{X}[k]\|_F} \tag{5-1}$$

For color videos, the relative error is computed in a similar manner. Instead of matrices, now the frames are 3-dimensional tensors: $\boldsymbol{\mathcal{X}}[k] \in \mathbb{R}^{M\times N\times 3}$. The relative error is then computed by (5-2).

$$\varepsilon[k] = \frac{\|\boldsymbol{\mathcal{X}}[k] - \widehat{\boldsymbol{\mathcal{X}}}[k]\|_F}{\|\boldsymbol{\mathcal{X}}[k]\|_F} \tag{5-2}$$

The computational speed is measured by timing the algorithm during the reconstruction. Subsequently, the average time spent per frame is computed over a large enough set of reconstructed frames (≥ 10). The times spent on rounding, extraction, multiplication, addition and other operations are also saved using the profiler in MATLAB. To compute the speedup that is achieved, the computation time per frame from the speedup method is divided by the original computation time per frame for all frames. Next, a mean and standard deviation is computed for all speedup percentages.

All computations are done using MATLAB version R2020b [22]. The computer that is has a Intel Core i7-6700 CPU running at 3.40 GHz and 16.0 GB of installed RAM.

5-2 Block Update TNKF

As described in the previous chapter, the goal of using the Block Update TNKF is to speed up computations. In this section, the Block Update TNKF is analyzed and compared with the Element Update TNKF in terms of computational speed and performance. To do so, the default settings listed in the previous section are chosen. Firstly, an initial comparison is made between both algorithms to see the influence of the block size on the computational speed. Subsequently, the effect of truncating the extracted column ranks is investigated. These subsections answer the first research question by giving insight into the relation between the parameters of the Block Update TNKF and its computational speed, and showing how the values of these parameters can be best chosen. This section is closed off with a final comparison with the state-of-the-art for both data sets. This answers the second research question through determining the speedups that can be achieved by replacing the Element Update TNKF with the Block Update TNKF and by assessing what the corresponding difference in performance is.

5-2-1 Block size influence

The first Block Update TNKF variable that is investigated is the block size. As mentioned in the previous chapter, the block size determines the number of measurements that is used to update the state and covariance each iteration. To assess what the fastest block size is and what speedup this yields when compared with the Element Update TNKF, we make an initial comparison. The average computation time per frame is investigated for reconstructing 10 frames of the Campus video data set. For the Block Update TNKF, this is done for block sizes ranging from 2 to 16. The results are plotted in Figure 5-2.

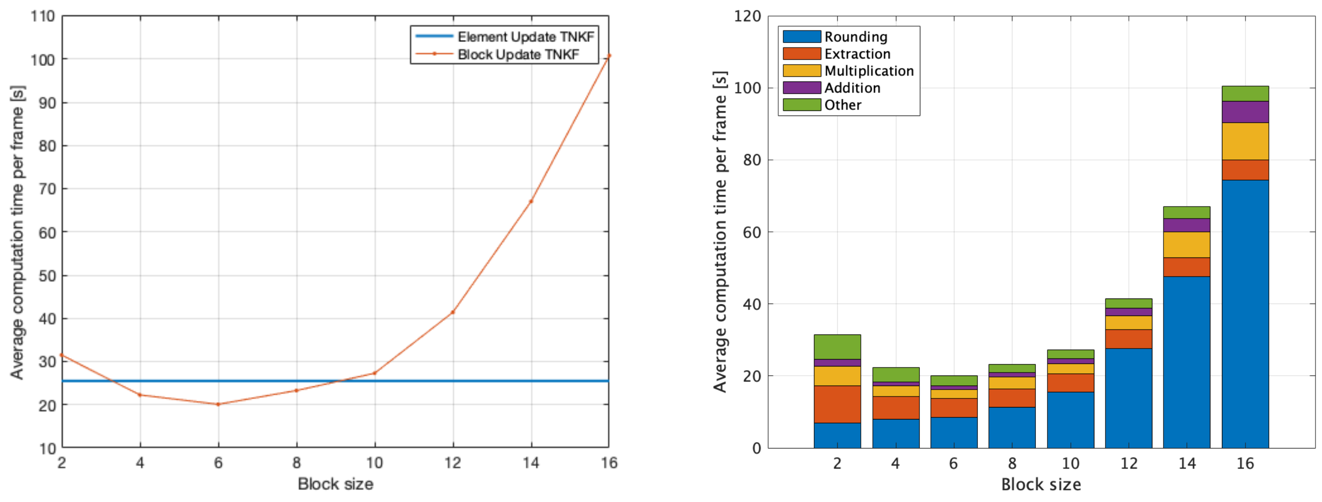


Figure 5-2: Computation time per frame of the Block Update TNKF compared with the Element Update TNKF (left) and average time spent on TT(m) operations (right), for varying block sizes. Campus data set, 10 frame averages.

In the left plot of Figure 5-2 we see that the Block Update TNKF outperforms the Element

Update TNKF in terms of computational speed for block sizes 4 - 8. In the right plot, it is observed that for smaller block sizes, the computation time per frame increases since more time is spent on TT(m) value and column extraction and the conversion from and to TT format. For larger block sizes, the computation time exponentially increases due to the time spent during the rounding of TT(m)'s. Choosing large block sizes has another disadvantage, as more information is discarded during each covariance TTm rounding step. The effect this has on the performance of the TNKF will be analysed further in this paper. In the simulation from Figure 5-2, the extracted columns $\mathbf{P}_{j-1}(:, \mathbf{c}_j)$ in TTm format are kept at full rank. To further speed up the Block Update KF, we can look at the rank truncation of these extracted columns as described in chapter 4. Truncating the rank at a specific maximum value (which we call $R_c^{\max}$) results in a decrease in the amount of rounding function calls. This is because of the lower rank of $\mathbf{K}_j\mathbf{S}_j\mathbf{K}_j$ term that is subtracted from the covariance matrix $\mathbf{P}_j$ (Algorithm 5, line 9). Furthermore, the computational load of each rounding step decreases as the maximum rounded rank is lower. As earlier mentioned, rounding of this matrix is a bottleneck in the algorithm, so tackling this can significantly speed up computations.

5-2-2 Extracted column rank truncation: performance

In order to determine at what rank the columns can be truncated, we can look at the singular values of the last core of the extracted columns in TTm format (as they are in site-d canonical form) for different block sizes. In Figure 5-3, the singular values of extracted columns of the covariance matrix $\mathbf{P}$ are plotted. This covariance matrix is taken during the reconstruction of a frame from the Campus video dataset.

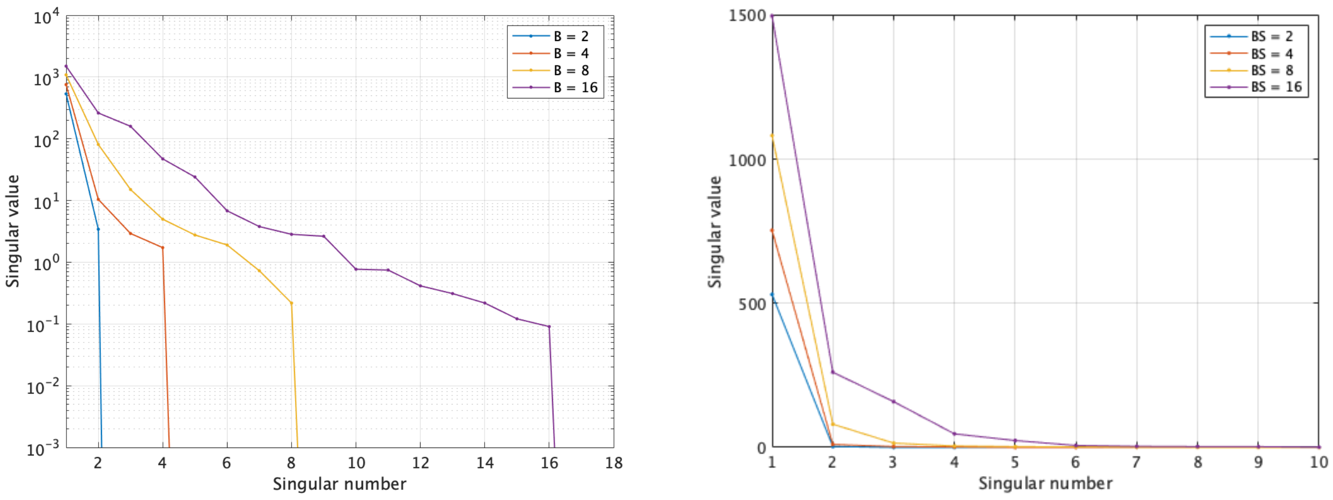


Figure 5-3: Singular values of the extracted columns of covariance matrix $\mathbf{P}$ for block sizes in powers of 2, logarithmic scale (left) and linear scale (right). Campus data set.

From the left plot with a logarithmic scale, it is observed that the singular values drop at the value of the block size. Since the singular values for all block sizes drop to almost zero (< 10e-12), the maximum rank of the columns in TTm format can be set equal to the block size without losing information. From the right plot with a linear scale, it is clear that the singular values drop relatively quickly to low values for this data set. This indicates that the ranks of the column TTm's can be truncated in order to further speed up computations.

We do compromise on performance when we truncate the ranks, as some information from the extracted columns of the covariance matrix is discarded. This results in a trade-off between computational speed and performance. The relative error is assessed for a reconstructed sequence of 30 frames to see how extracted column rank truncation influences the performance. The block size is kept constant at $B = 16$ while the maximum extracted column ranks are varied from $B/4$ to B. The results are plotted in Figure 5-4.

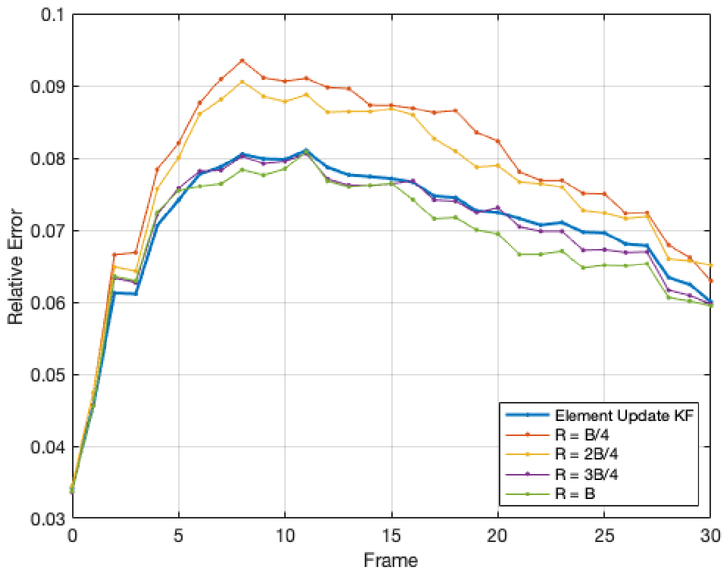


Figure 5-4: Relative error of a reconstructed sequence of 30 frames using the Block Update TNKF with block size $B = 16$, for varying maximum extracted column rank. Campus data set.

The orange line ($R_c^{\max} = B/4$) and yellow line ($R_c^{\max} = B/2$) in Figure 5-4 indicate that the relative error of the reconstructed frames increases significantly when truncating the ranks to lower values. When doing less truncation on the ranks, the performance comes close to the element update KF as indicated by the blue, purple and green lines. Interestingly, when the columns are kept at full rank, the Block Update TNKF seems to outperform the Element Update TNKF for this specific sequence as the relative error generally has lower values. Determining what truncation parameter $R_c^{\max}$ to choose thus comes down to a trade-off between computational speed and performance.

5-2-3 Extracted column rank truncation: speed

To see what effect rank truncation of the extracted columns has on the computational speed of the Block Update TNKF, we use a block size of $B = 16$ and assess the average computation time per frame for varying maximum extracted column rank. The results are plotted in Figure 5-5.

The left plot in Figure 5-5 shows that the computational speed drastically increases when the ranks of the columns are truncated. When the ranks of the extracted columns in TTm format

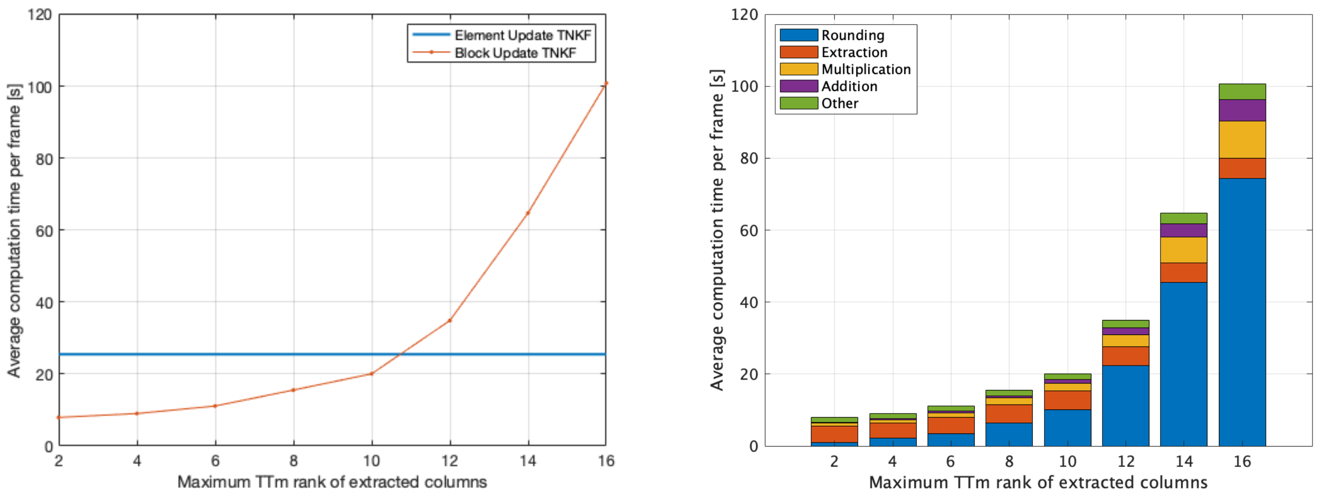


Figure 5-5: Average computation time per frame (left) and average time spent on TT(m) operations (right) of a reconstructed sequence of 30 frames by the Block Update KF with block size $B = 16$, for varying maximum extracted column rank. Campus data set, 30 frame average.

are truncated to low values, the computational complexity of each rounding step decreases due to the lower ranks. Since rounding is the main driver of the computational costs for higher values of $R_c^{\max}$, this increases the computation time per frame considerably. This is substantiated by the right plot of Figure 5-5, where again, the time spent on rounding exponentially increases when the maximum extracted column ranks are increased. For high values of $R_c^{\max}$, the time spent on TTm operations such as multiplication and addition also increases as a result of doing these computations with larger ranks.

When $R_c^{\max}$ is chosen low enough, there is no upper limit for the chosen block size other than the total amount of missing pixels in the frame. For very large block sizes, the Block Update TNKF can reconstruct a frame very fast compared to the Element Update TNKF. This does, as illustrated in the next section, decrease performance substantially. It could be that performance is less of an issue than computational speed when the missing pixel percentage is lower. Here the Block Update TNKF with extracted column rank truncation could be an interesting solution.

5-2-4 Comparison with state-of-the-art

To see what speedups the Block Update TNKF can achieve compared to the Element Update TNKF and what the corresponding performance is, the fastest value for the block size from the analysis in Figure 5-2 ($B = 6$) is investigated. This is done by decreasing the maximum TTm rank of the extracted columns from 6 to 1. The resulting computational speed and performance are compared with the Element Update TNKF, and results are plotted in Figure 5-6.

In order to determine what speedup we have achieved, we can analyze the relative errors and the average computation time per frame for both algorithms. From the left plot, it is clear

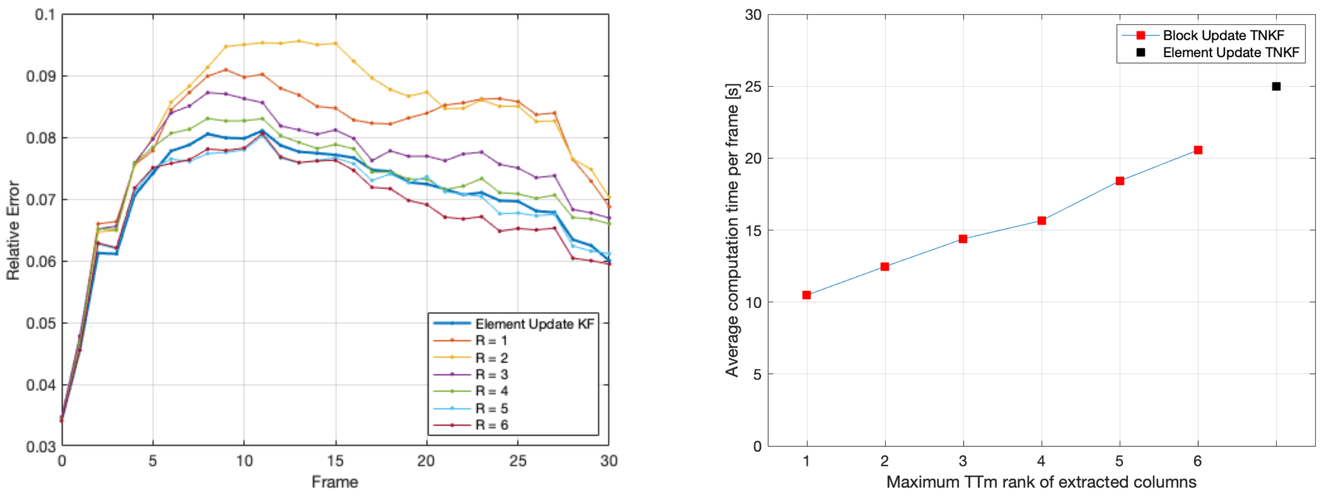


Figure 5-6: Relative error (left) and average computation time per frame (right) comparison of the Block Update TNKF and the Element Update TNKF for a 30 frame reconstructed sequence. Campus data set.

that the maximum rank of the extracted columns of $R_c^{\max} = 6$ gives the best performance compared to the Element Update KF. Lower values result in a larger relative error, thus compromising performance. In the right plot, the average computation time per frame of the Block Update TNKF for the same extracted column ranks and the Element Update TNKF are depicted. Selecting the block size $B = 6$ and column rank of $R_c^{\max} = 6$ results in a total speedup of $\mathbf{+22\%}$, while at the same time improving the performance slightly. When lower values of $R_c^{\max}$ are chosen, even larger speedups (up to $\mathbf{+138\%}$) can be achieved, depending on the desired level of performance.

It must be noted that truncating these ranks generally results in worse performance compared to the Element Update TNKF. Truncating the ranks of the extracted columns is only desirable when performance is less important than computational speed. An example of this could be a relatively low missing pixel percentage. An advantage of using the Block Update TNKF is that it allows this rank truncation to speed up computations, while the Element Update TNKF does not have this option.

To validate the realised speedup, we test the Block Update TNKF for the second video data set: Grand Central Station. In Figure 5-7, the relative error per frame is plotted for a 30-frame sequence using both the Block Update TNKF and the Element Update TNKF. The Block Update TNKF settings are chosen as $B = 6$ and $R_c^{\max} = 6$, the missing pixel percentage at 95%.

Interestingly, the left plot in Figure 5-7 shows that the Block Update TNKF performs worse for these settings and frames as the relative error is generally about 0.05 higher compared to the Element Update TNKF. The reason for this is the difference in the number of iterations between both algorithms. The Element Update TNKF updates the state and covariance for each measured pixel. The Block Update TNKF reduces this with a factor of the block size B. This, however, results in a large rank increase in the covariance TTm (of B^2) each

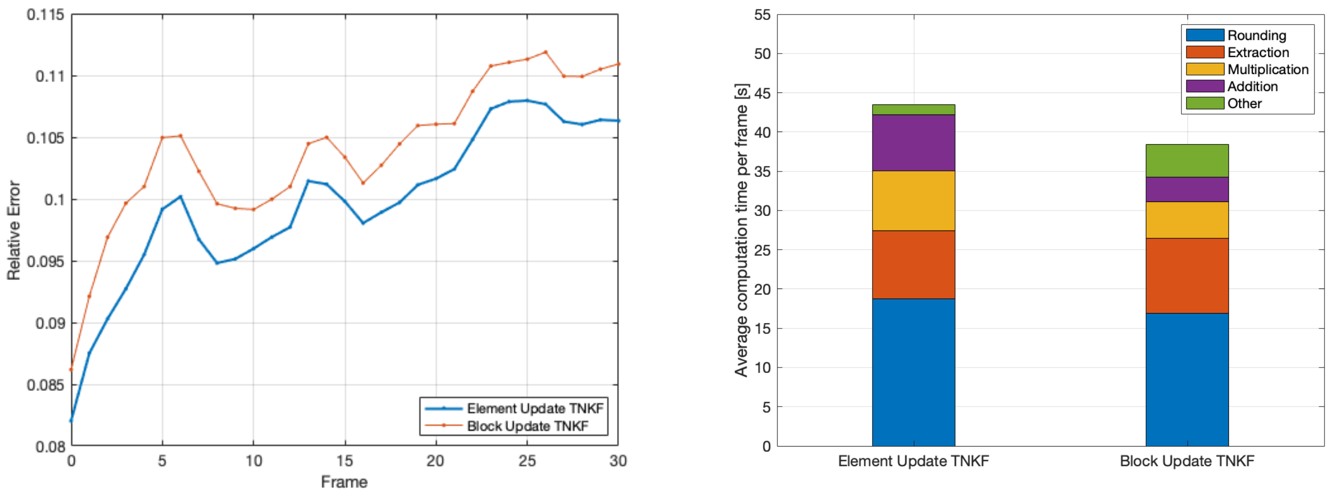


Figure 5-7: Performance (left) and speed (right) of the Block Update TNKF vs. the Element Update TNKF. Grand Central Station data set.

iteration. Since the ranks must be truncated back to a maximum value of 1, more information is discarded each iteration compared to the Element Update TNKF. There, the ranks of the covariance TTm do not overshoot the threshold value ($R_{\mathbf{P}}^{\max}$) by far when the TTm is rounded. This seems to affect the performance of the Block Update TNKF for this data set, many small moving objects can be better reconstructed using more information from the covariance matrix. To tackle this, the maximum rank of the covariance matrix $R_P^{\max}$ can be increased to a value that allows for a more exact computation of the TNKF equations. This slows down computations considerably, however, as now the ranks of the covariance matrix are larger. This results in more computational load from the TTm operations. The Campus data set suffers less from this, as there are fewer measured pixels in moving parts of the screen. This makes the update of the covariance matrix less important for the quality of the reconstruction. This shows that the larger the block size is chosen, the more information from the covariance matrix is discarded during rounding which decreases the performance. In the last section of this chapter, the covariance matrix's influence on the reconstruction quality is further investigated.

In the Table 5-2, the average relative error differences and speedups of the Block Update TNKF ($B = 6$ and $R_c^{\max} = 6$) are listed compared with the Element Update TNKF for both data sets. The average relative error difference is computed by subtracting the relative error of the reconstructed frame of the Block Update TNKF from the relative error of the same reconstructed frame while using the Element Update TNKF. This is done for each frame of the reconstructed sequence, before computing the average relative error over all frames of the considered sequence. A positive value (green) for this indicates that the Block Update TNKF performs better than the Element Update TNKF. For negative values (red) it performs worse.

Data set (Frame #)	**Missing pixel percentage**	**Average relative error difference**	**Speedup percentage**
Campus (50 - 80)	99%	1.9e-03	+**20.1** ± **2.2**%
Campus (200 - 230)	99%	-2.8e-03	+**21.2** ± **3.9**%
Grand Central Station (50 - 80)	95%	-4.33e-03	+**13.1** ± **1.8**%
Grand Central Station (200 - 230)	95%	-5.2e-03	+**17.1** ± **1.6**%

Table 5-2: Average relative error difference and speedup percentages of the Block Update TNKF compared with the Element Update TNKF.

The table shows that the Block Update TNKF results in speedups ranging from **+13.1%** to **+21.2%** for the conducted experiments. The usage of this algorithm does compromise the performance, which is a result of more discarded covariance information during the rounding procedure. This does not result in instability of the reconstruction, however, as the relative error stays within reasonable bounds.

5-3 Randomized rounding algorithms

In this section, an attempt is made to reduce the computational load of the bottleneck in the TNKF: the TT(m) rounding procedure. This is done by replacing the standard deterministic rounding algorithm with randomized rounding algorithms introduced in [10] and elaborated in Chapter 4. Firstly, the effect of randomized rounding of the state TT is assessed. During these simulations, the ranks of the covariance matrix are truncated using the deterministic rounding algorithm (Algorithm 3). Furthermore, a suitable oversampling parameter for the state rounding with the TSR algorithm is chosen. How this parameter influences performance will be investigated later in this section. After this, instability of the TNKF algorithm that occurs due to randomized rounding of the covariance matrix is discussed. In these subsections, the third research question is addressed by investigating the effects of randomized rounding of the state and covariance on the performance and speed of the TNKF and examining what parameters influence these effects. In the last subsection, the randomized rounding algorithms are compared with deterministic rounding in the Element Update TNKF to determine the total speedups and differences in performance. This answers the fourth research question.

5-3-1 State rounding

As mentioned in Chapter 4, the rounding of the state TT accounts for 10 - 15% of the total computational costs of the TNKF. In order to investigate the effect of using randomized rounding on the state TT we reconstruct a 30-frame sequence from both data sets (Campus and Grand Central Station). This is done using all considered rounding algorithms in the Block Update TNKF (Algorithm 5). In Figure 5-8, the resulting relative error per frame and average state rounding time per frame are shown for a 30-frame reconstructed sequence in the Campus data set.

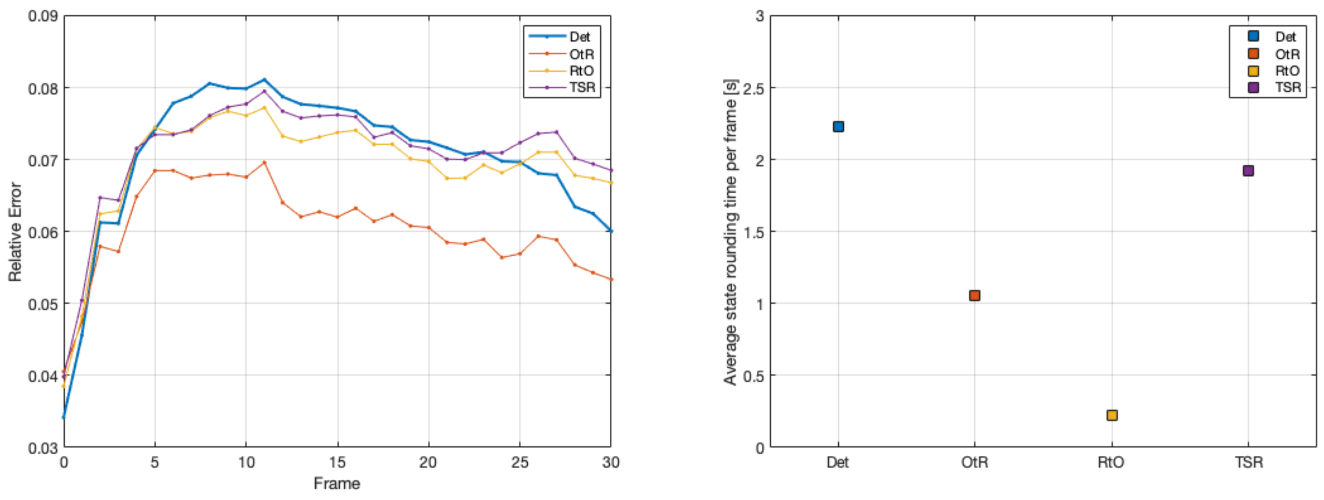
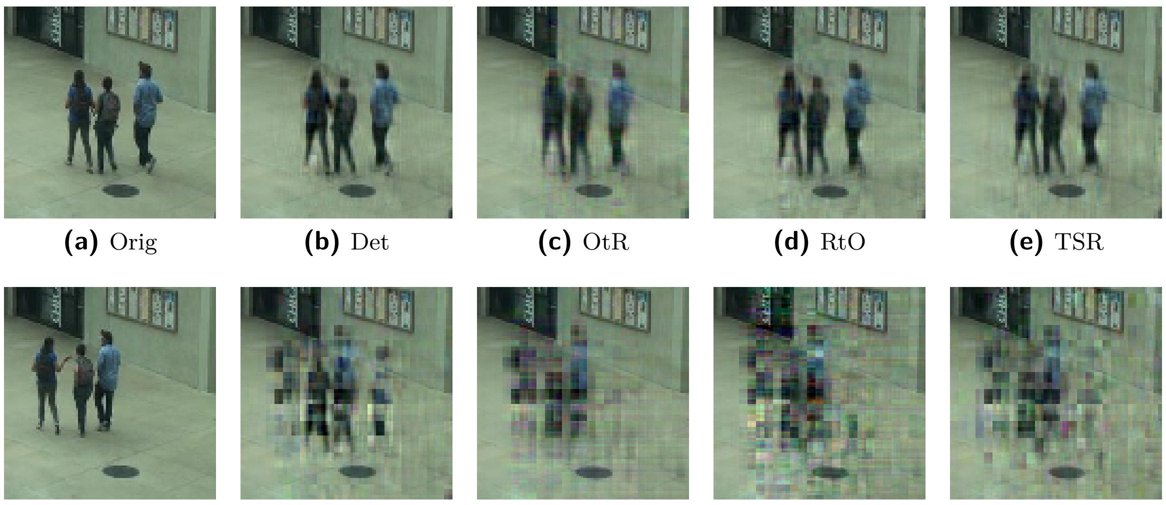


Figure 5-8: Initial comparison of rounding the state TT with randomized rounding algorithms and deterministic rounding. Left: relative error per frame, right: average state rounding time per frame (30 frame average). Block Update TNKF, Campus data set.

From the left plot in Figure 5-8 we see that the RtO (Algorithm 8) and TSR (Algorithm 9) randomized TT-rounding algorithms are fairly comparable to deterministic rounding in terms of performance. The OtR rounding (Algorithm 7) seems to perform the best out of all the rounding algorithms, including deterministic rounding. When looking at the right plot, interesting results are observed. All randomized rounding algorithms have sped up the rounding of the state TT. There RtO algorithm has by far the largest speedup: **10.1**×.

To see how the differences in relative error translate into visual differences in the reconstructed frames, a specific moving part (walking persons) is analyzed for all rounding algorithms. The resulting frames at time $T = 0$ (first corrupted frame) and $T = 30$ (around one second after corruption) are plotted in Figure 5-9. In Table 5-3, the relative errors of the full frames of all algorithms are listed for both reconstructed frames.



(a) Orig **(b)** Det **(c)** OtR **(d)** RtO **(e)** TSR

Figure 5-9: Moving part of reconstructed frames 1 (top row) and 30 (bottom row) using all rounding algorithms. The first frame of both rows is the original frame. Campus data set.

	Det	**OtR**	**RtO**	**TSR**
Relative error frame 1	3.41e-02	4.06e-02	3.85e-02	3.98e-02
Relative error frame 30	5.95e-02	5.34e-02	6.68e-02	6.85e-02

Table 5-3: Relative error of full reconstructed frames 1 and 30 using all rounding algorithms. Campus data set.

From Figure 5-9, it can be seen that the first reconstructed frames are very similar for all rounding algorithms, as already indicated in the relative error plot in Figure 5-8. We see significant differences when looking at the reconstructed frames about 1 second after the corruption (frame 30). The deterministic rounded frame suffers from the 'shadow effect' mentioned in [30], where the moving parts of the initial frame appear as shadows behind the reconstructed parts. This results from the default setting for the initial state $\mathbf{x}_0$, which is set at the last uncorrupted frame. The randomized rounding algorithms seem to eliminate this effect by rounding the state TT differently for these particular frames and settings. This is an interesting observation, as the covariance matrix is rounded with the deterministic rounding algorithm in all cases. From these algorithms, OtR performs the best as the silhouettes of the persons and the colors of their clothes can still be observed quite well. The RtO and TSR algorithms seem to compromise the most in terms of performance, as substantiated by the relative errors in Table 5-3. One reason for high relative error of the RtO and TSR algorithms is the column distortion effect in their reconstructed frames. By adding randomization to the state rounding, the columns of pixels where moving parts are in the frame seem to be influenced. This distortion increases the longer the reconstruction takes place. The full reconstructed frames of this analysis can be found in Figure B-3 listed in Appendix B.

Next, we investigate how the randomized rounding algorithms perform on the second data set: Grand Central Station. The resulting relative error and average state rounding time per frame are depicted in Figure 5-10.

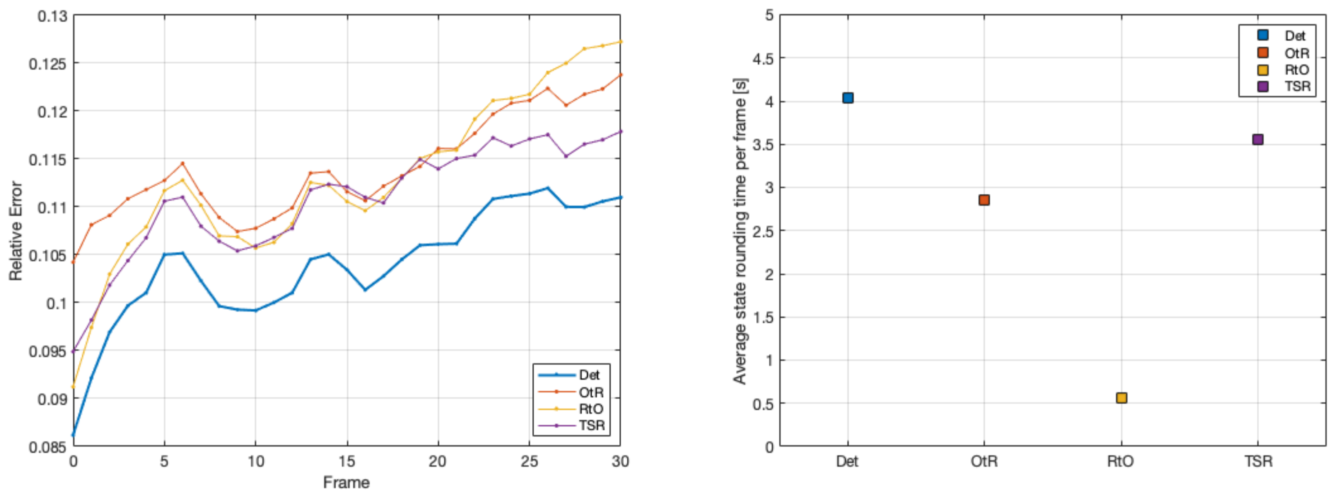


Figure 5-10: Comparison of randomized rounding algorithmss with deterministic rounding. Left: relative error per frame, right: average state rounding time per frame. Block Update TNKF, Grand Central Station data set.

In the right plot of Figure 5-10 we again see similar speedups achieved by the randomized rounding algorithms compared to the first simulation. This time, the RtO algorithm achieves a speedup of **7.23**× compared to deterministic rounding of the state TT. In the left plot, however, we see a difference as the randomized rounding algorithms clearly perform worse compared to deterministic rounding for this data set. Here, OtR did not outperform the other rounding algorithms and TSR seems to perform the best out of the randomized algorithms.

To see how this translates into the reconstructed parts of the frame, a specific moving part (walking persons) is analyzed for all rounding algorithms. The resulting frames at time $T = 0$ (first corrupted frame) and $T = 30$ (around one second after corruption) are plotted in Figure 5-11. In Table 5-3, the relative errors of the full frames of all algorithms are again listed for both reconstructed frames.

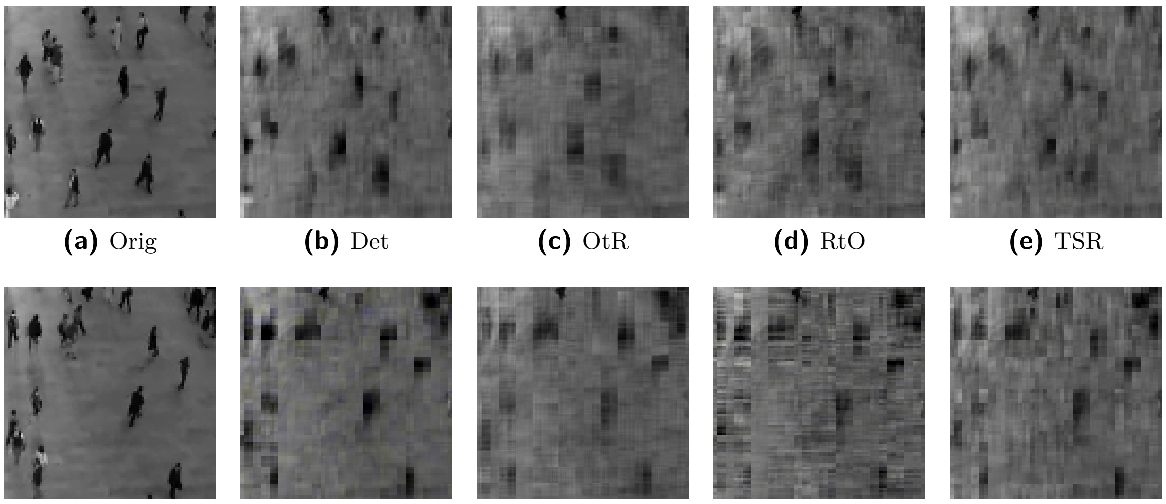

(a) Orig **(b)** Det **(c)** OtR **(d)** RtO **(e)** TSR

Figure 5-11: Moving part of reconstructed frames 1 (top row) and 30 (bottom row) using all rounding algorithms. The first frame of both rows is the original frame. Grand Central Station data set.

	Det	**OtR**	**RtO**	**TSR**
Relative error frame 1	8.62e-02	1.13e-01	1.01-e01	1.03e-01
Relative error frame 30	1.12e-01	1.18e-01	1.31e-01	1.26e-01

Table 5-4: Relative error of full reconstructed frames 1 and 30 using all rounding algorithms. Grand Central Station data set.

Contrary to the first simulation, it is observed that already the first reconstructed frames differ in reconstruction quality. Deterministic rounding seems to be able to reconstruct the silhouettes of the persons relatively well, whereas the randomized rounding algorithms give more distortion. When looking at the 30th reconstructed frames in the bottom row, we can still determine the position of certain people in the frame which is harder for randomized rounding. The frames of the Grand Central Station video contain more small moving parts, as many persons are walking on the station square. The randomization introduced in the rounding of the state TT prevents the TNKF to reconstruct these parts of the frame with the same quality as for deterministic rounding. Since a large part of the frame is covered by the moving 'foreground', the differences in the rounding of the state TT translate to more difference in relative error. The full reconstructed frames of this analysis can be found in Figure B-4 listed in Appendix B.

5-3-2 State rank influence

In the previous section, it was shown that for the Grand Central Station data set, the randomized rounding algorithms compromise the performance. Now, we attempt to improve the performance by increasing the desired state rank $R_{\mathbf{x}}$. This value was set to $R_{\mathbf{x}} = 30$ by default and increasing could improve the performance. As earlier explained, this does slow down computations since carrying out operations with larger state ranks is more computationally expensive. To keep the amount of state rounding function calls similar to the original situation, we increase the maximum state rank with the same amount $R_{\mathbf{x}^{\max}} = R_{\mathbf{x}} + 20$. The results are plotted in Figure 5-12.

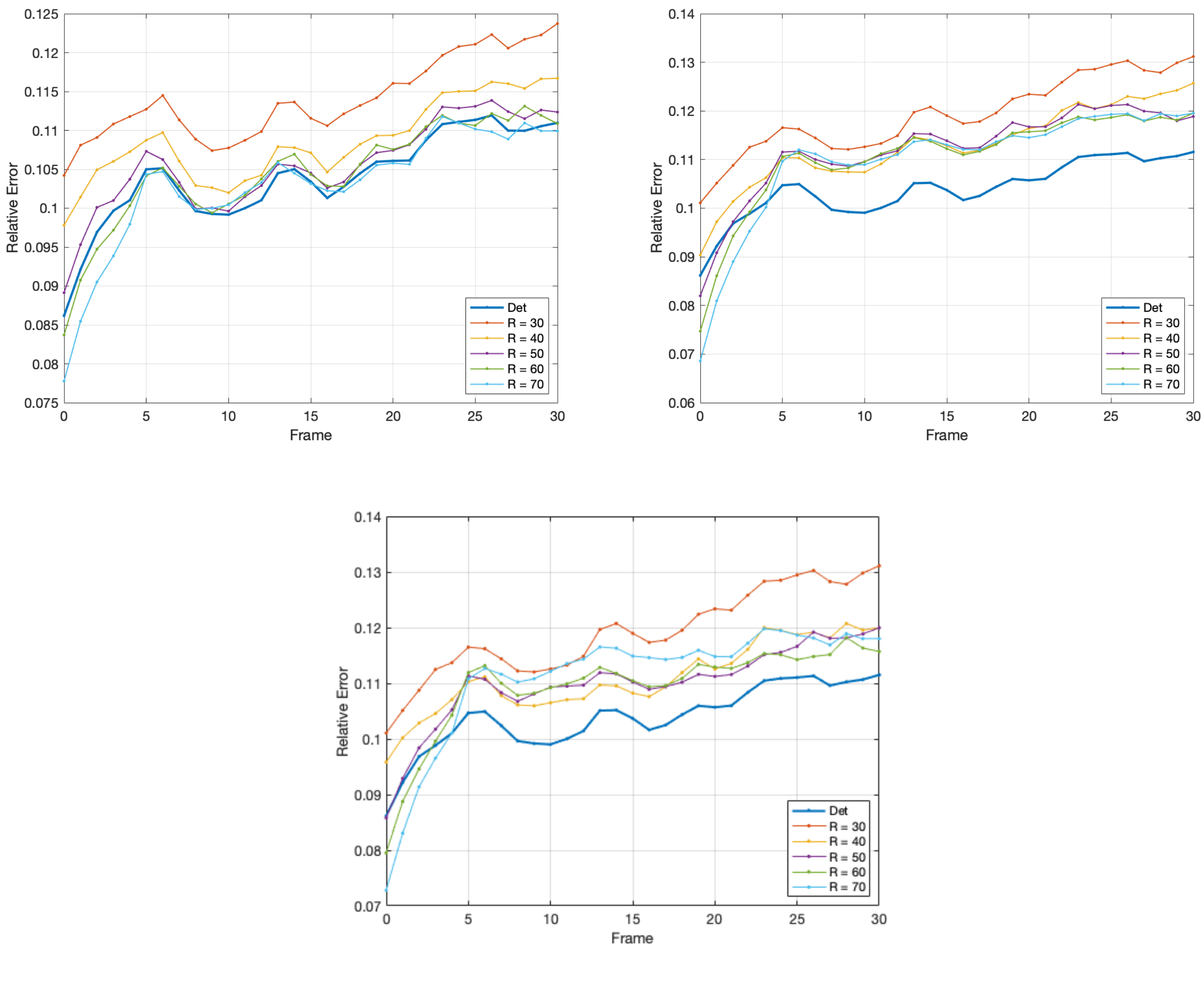


Figure 5-12: Relative error of reconstructed frames using randomized rounding algorithms for different desired state rank $R_{\mathbf{x}}$. Top left: OtR, top right: RtO, bottom: TSR. Block Update TNKF, Grand Central Station data set.

For all algorithms, it is shown that increasing the desired state rank improves the performance.

For the RtO and TSR algorithm, the performance stops improving at a value of $R_{\mathbf{x}} = 40$. When further increasing this rank, the performance only improves for the first five reconstructed frames. For the OtR algorithm however, the rank increase results in a performance that is similar to deterministic rounding for this data set and settings.

5-3-3 State rounding: long simulation

During the simulations that have been run, the randomized rounding algorithms perform relatively well in the first reconstructed frames. After some time, the relative error per frame of randomized rounding algorithms seems to rise more quickly than deterministic rounding. It could be the case that the randomization adds more and more distortion to the state as the reconstruction progresses. To see what the longer term effects are of using randomized rounding, a sequence of 100 corrupted frames is reconstructed for the Grand Central Station data set corresponding to more than 4 seconds of video footage (frame rate = 24 fps). The results are shown in Figure 5-13.

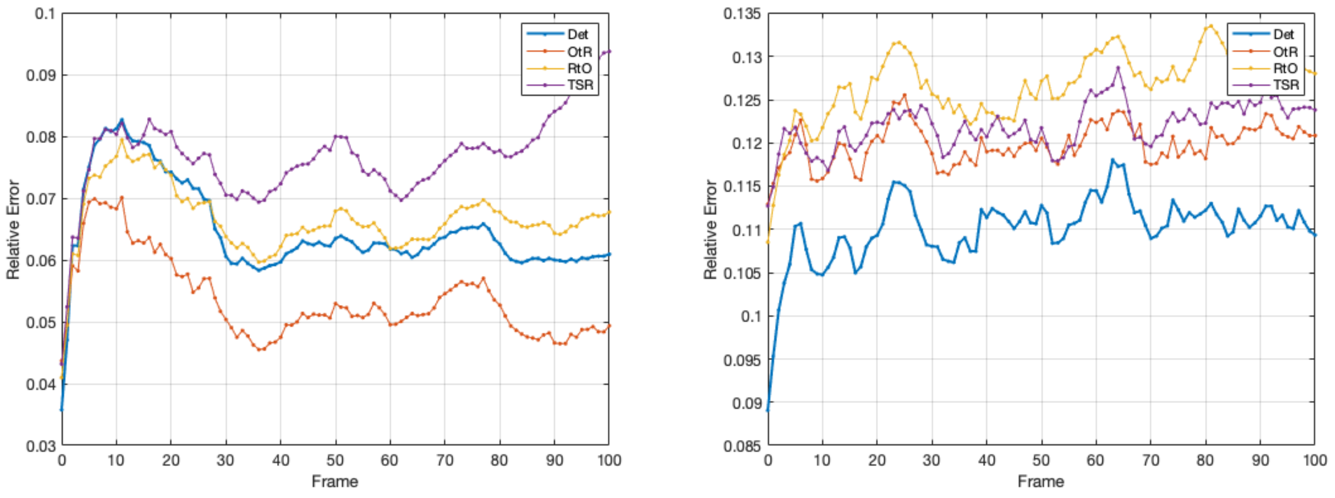


Figure 5-13: Relative error for deterministic and randomized rounding algorithms for a 100 frame reconstructed sequence. Block Update TNKF, Campus (left) and Grand Central Station (right) data sets.

From the plot, it is clear that there is a difference between the performance of randomized and deterministic rounding, as earlier illustrated. This difference is not largely influenced by the length of the reconstruction, as the relative errors seem to stabilize over time for all rounding algorithms. During this whole sequence, persons are walking through the frame, which the algorithms are able to reconstruct to a certain extent. In the long term, the RtO algorithm seems to perform the worst out of all algorithms for this data set. The OtR and TSR algorithms have similar performance, but after 100 reconstructed frames, there is still a substantial difference with deterministic rounding.

5-3-4 TSR algorithm: oversampling

In chapter 4, it is elaborated that the TSR algorithm requires choosing an oversampling parameter which determines the ranks $\boldsymbol{\rho} = [\rho_0, \ldots, \rho_d]$ of the Gaussian TT-tensor $\boldsymbol{\mathcal{R}}$. To determine this parameter for state rounding with the TSR algorithm, we vary the maximum rank of the random TT $\boldsymbol{\mathcal{R}}$ ($R^{\max}_{\boldsymbol{\mathcal{R}},\mathbf{x}}$) and assess the performance of the algorithm for rounding the state TT. The resulting relative errors per frame for the Campus data set are depicted in Figure 5-14.

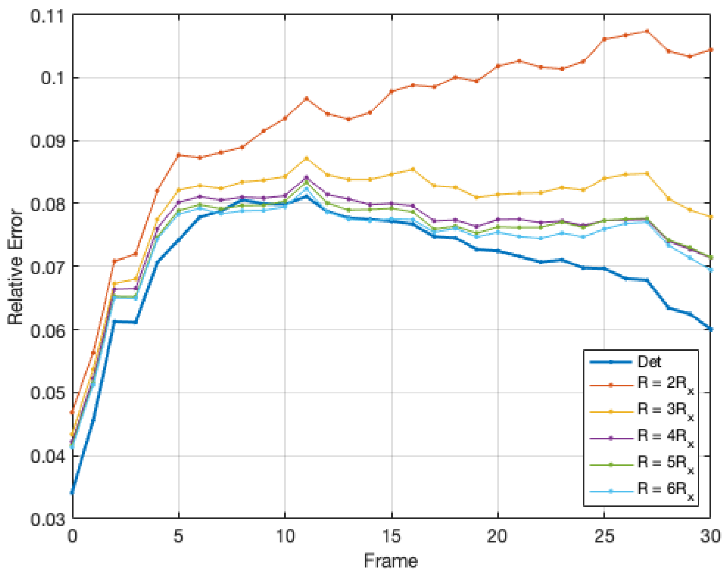


Figure 5-14: Relative error per frame for different state rounding oversampling parameters $R^{\max}_{\boldsymbol{\mathcal{R}},\mathbf{x}}$ using the TSR algorithm. Block Update TNKF, Campus data set.

Figure 5-14 indicates that choosing relatively low values for the oversampling $R^{\max}_{\boldsymbol{\mathcal{R}},\mathbf{x}} = 2R_{\mathbf{x}}$ results in bad performance as the completion seems to become unstable. Clearly the suggested oversampling of $\rho_n = [1.5\ell_n]$ by [23] does not give satisfactory results for this specific application. Increasing this oversampling parameter increases the computational load of the algorithm. Since the time spent on rounding the state TT is only a small percentage of the total computational time, this increase is relatively little however. An oversampling parameter $R^{\max}_{\boldsymbol{\mathcal{R}},\mathbf{x}} = 4R_{\mathbf{x}}$ seems to be a good choice, since higher values do not significantly improve performance for this data set as shown in Figure 5-14. For the remainder of this analysis, this oversampling parameter will be chosen. In Figure 5-15 two reconstructed frames are shown for a low oversampling parameter (left, $R^{\max}_{\boldsymbol{\mathcal{R}},\mathbf{x}} = 2R_{\mathbf{x}}$) and a high oversampling parameter (right, $R^{\max}_{\boldsymbol{\mathcal{R}},\mathbf{x}} = 6R_{\mathbf{x}}$) for rounding the state TT with the TSR algorithm.

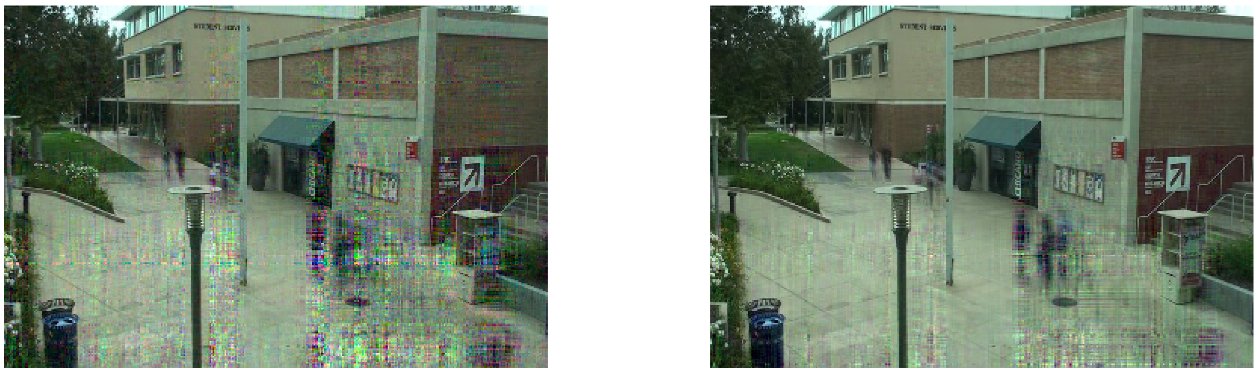


Figure 5-15: Reconstructed frames using the TSR randomized rounding algorithm with a low oversampling parameter (left, $R^{\text{max}}_{\boldsymbol{\mathcal{R}},\mathbf{x}} = 2R_{\mathbf{x}}$) and a high oversampling parameter (right, $R^{\text{max}}_{\boldsymbol{\mathcal{R}},\mathbf{x}} = 6R_{\mathbf{x}}$). Campus data set.

From Figure 5-15 the importance of selecting a large enough oversampling parameter is highlighted, as the left plot with a low oversampling parameter is much more distorted than the right plot with a high oversampling parameter. Interestingly, this distortion mainly occurs in the columns where moving parts are located. After determining the oversampling parameter for rounding the state, the same can be done for the rounding of the covariance. After some tests, it was found that the choice of the oversampling parameter for the covariance does not influence the performance significantly. It must be chosen large enough to ensure stability. For the remainder of this paper, it is chosen to be $R^{\text{max}}_{\boldsymbol{\mathcal{R}},\mathbf{P}} = 3R_{\mathbf{P}} = 3$.

5-3-5 Covariance rounding: instability

Although the randomized rounding algorithms significantly speed up computations for the state TT, there is a shortcoming of using randomized rounding for the streaming video completion problem. By adding randomization to reduce the computational load of the rounding procedure of the covariance matrix $\mathbf{P}$ (in TTm format), the positive definite property of the corresponding matrix can be lost. This can lead to the completion of the TNKF becoming unstable: entries in the state vector and covariance matrix blow up to very large positive and negative numbers. An example of a reconstructed frame where this instability occurs is depicted in Figure 5-16, where the OtR rounding algorithm was used.

The reason for these instability issues with randomized rounding algorithms is that the error between the 'full' covariance matrix and the rounded covariance matrix is larger by adding a randomization step. Since the covariance matrix is rounded to a rank 1 TTm, the difference in error can result in the $\mathbf{P}$ matrix losing its positive definiteness, thus destabilizing the TNKF. In order to tackle this, introducing an oversampling term p to the desired covariance TTm rank $\ell = r + p$ can be introduced. For the OtR algorithm, this can be done by simply increasing the desired covariance TTm rank. The problem is that this dramatically slows down computations as the rounding function now must be called more often and computations with larger ranks

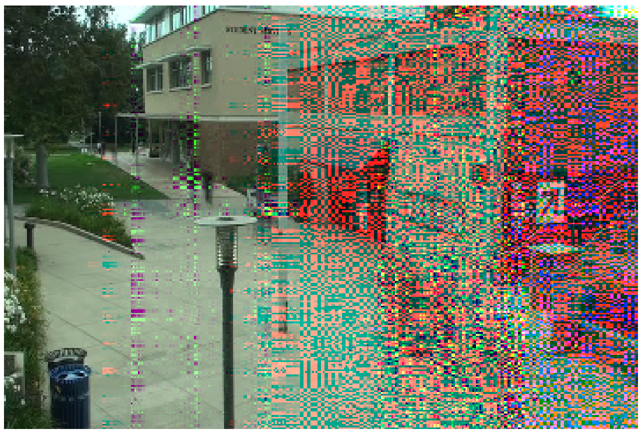


Figure 5-16: Example of instability in the TNKF caused by randomized rounding with the OtR algorithm. Campus data set.

such as addition and multiplication have a higher computational load. Because this research aims to speed up the TNKF, the OtR algorithm will not be considered for further analysis.

The RtO and TSR algorithm seem to trigger this instability less since the randomization is induced in a less direct manner by computing the left and right partial contraction matrices with random TT's first (Algorithm 8, line 2 and Algorithm 9, lines 3 and 4). If this instability does occur, the RtO algorithm does not have an easy oversampling solution since the same covariance oversampling issue slows down the rounding procedure considerably. For the TSR algorithm, the oversampling can be done by increasing the ranks of the random TT $\boldsymbol{\mathcal{R}}$, as explained in the previous section. During testing, it was found that the choice of oversampling can tackle the instability issues for both data sets. Aside from oversampling, another measure to counter this instability was found for the RtO and TSR rounding algorithms. Instability is less of an issue when the entries of the random TT's $\boldsymbol{\mathcal{R}}$ and $\boldsymbol{\mathcal{L}}$ are chosen to be uniformly distributed random numbers (`rand` function in MATLAB) instead of random numbers drawn from a zero-mean Gaussian distribution (`randn` function) for the rounding of the covariance matrix. As the cores of the rounded covariance matrix in TTm format are now multiplied with only positive values, the chances of losing its positive definiteness seem lower. During experimentation it was found that making this change does influence the performance of the TNKF. For the remainder of this analysis, this measure will be implemented for rounding the covariance matrix with both the RtO and TSR algorithm to ensure stability of the reconstruction.

5-3-6 State-of-the-art comparison

In this section, the randomized rounding of the state and covariance is compared to deterministic rounding. To assess the speedup that can be achieved, we compare the rounding algorithms in the Element Update TNKF for both data sets. Firstly, the Campus data set is considered where the settings are chosen as default. The resulting relative error and average computation time per frame are shown in Figure 5-17.

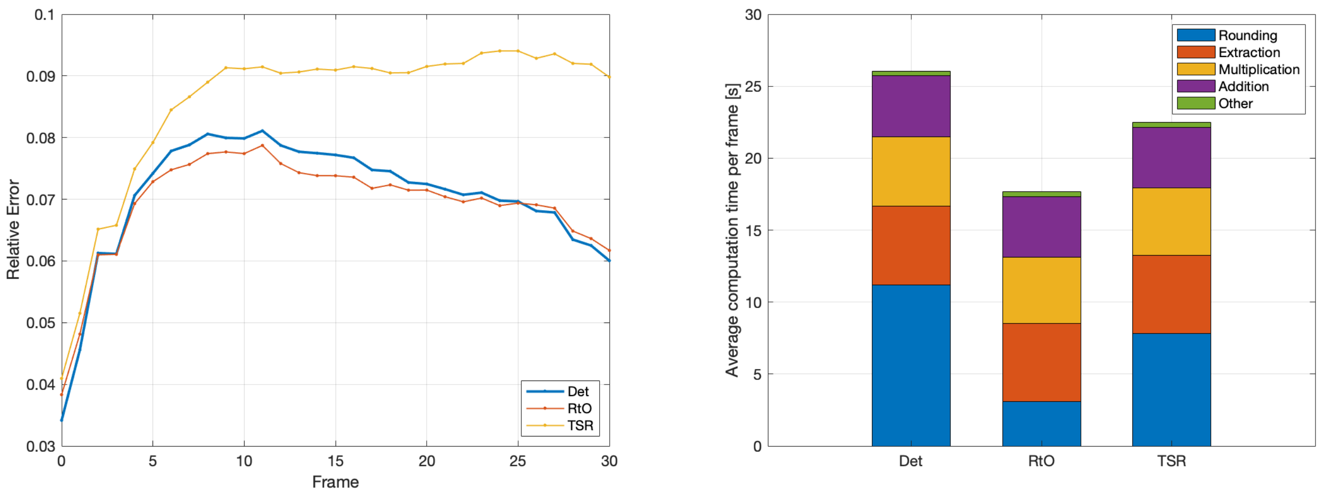


Figure 5-17: Comparison of rounding algorithms for rounding of the state and covariance. Left plot: relative error per frame, right plot: average computation time per frame for main operations. Element Update TNKF, Campus data set.

The relative error plot in Figure 5-17 shows that deterministic rounding and RtO have comparable performance. The TSR rounding algorithm performs the worst of all algorithms. The reason for this is the influence its randomization has on the covariance matrix. In the first simulation in this section (Figure 5-8), it was observed that using the TSR algorithm for the rounding of the state does not result in a significant drop in performance. However, when using the algorithm for the rounding of the covariance matrix, the performance drops considerably. This drop in performance cannot be solved by selecting a higher oversampling parameter for the rounding of the covariance matrix, unfortunately. The achieved total speedups are +**47**% for RtO and +**16**% for TSR. Interestingly, the RtO algorithm does not compromise much on performance while having the largest speedup. Next, the speedup is assessed for the Grand Central Station data set. The results are plotted in Figure 5-18.

From the relative error plot, it is clear that both randomized rounding algorithms seem to result in an instability of the completion. Although a speedup has been achieved of +**42**% for RtO and +**11**% for TSR, the influence of randomization on the covariance matrix dramatically influences the performance. Clearly, the covariance matrix has a strong role in the quality of the reconstruction. Injecting too much randomization into this matrix results in poor performance. In Table 5-5, the cumulative relative error difference (over 30 frames) between the considered rounding algorithm and deterministic rounding is listed with the achieved speedup for all simulations.

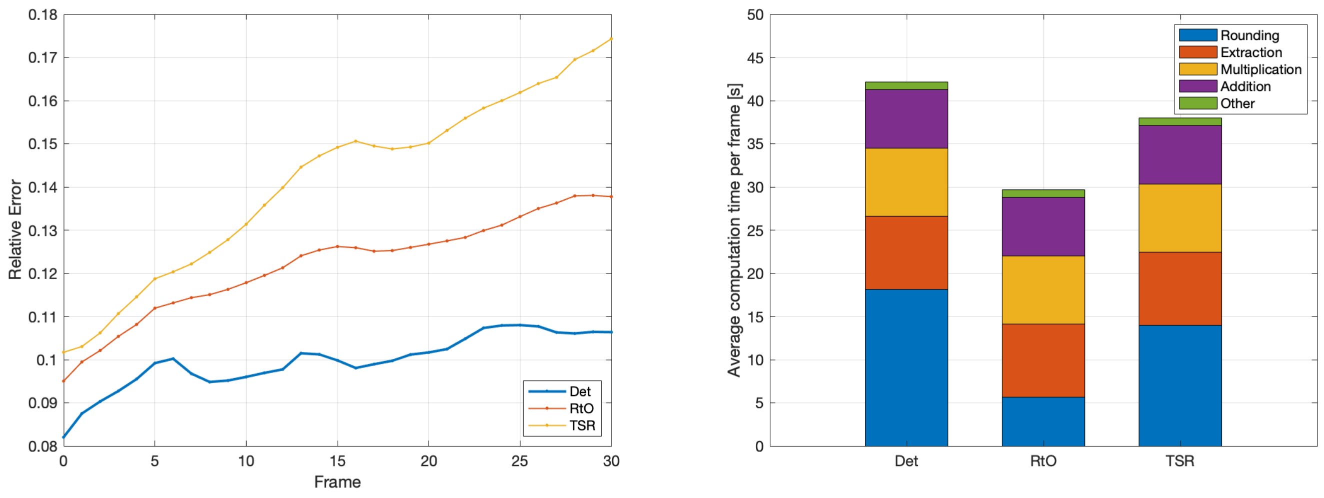


Figure 5-18: Comparison of rounding algorithms for rounding the state and covariance. Left plot: relative error per frame, right plot: average computation time per frame for main operations. Element Update TNKF, Grand Central Station data set.

Data set (Frame #)	**Rounding algorithm**	**Average relative error difference**	**Speedup percentage**
Campus (50 - 80)	RtO	1.1e-03	**+47 ± 2.1%**
Campus (50 - 80)	TSR	-1.5e-02	**+16 ± 1.6%**
Campus (200 - 230)	RtO	1.9e-04	**+41 ± 4.6%**
Campus (200 - 230)	TSR	-8.3e-03	**+9 ± 0.8%**
Grand Central Station (50 - 80)	RtO	-2.2e-02	**+42 ± 3.6%**
Grand Central Station (50 - 80)	TSR	-4.2e-02	**+11 ± 1.7%**
Grand Central Station (200 - 230)	RtO	-2.4e-02	**+41 ± 1.8%**
Grand Central Station (200 - 230)	TSR	-3.7e-02	**+8.3 ± 1.4%**

Table 5-5: Average relative error differences and speedups from randomized rounding compared to deterministic rounding. Element Update TNKF.

As can be concluded from the last column in Table 5-5, the usage of randomized rounding algorithms increases the computational speed of the TNKF algorithm. The RtO rounding algorithm shows total speed ups ranging from **+41%** to **+47%**, for the TSR these speedups range from **+8.3%** to **+11%**. The average relative error difference shows that generally, the usage of randomized rounding algorithms negatively influences performance. For data sets with limited moving parts that are not too small (Campus video), there is not much loss in performance when using the RtO algorithm for the rounding of both the state TT and the covariance TTm. For data sets with more and smaller moving parts, the speedup obtained by randomization comes with a significant performance loss, especially when using randomized rounding on the covariance matrix. In the next section, the effect of the covariance matrix on the TNKF is further researched.

5-4 Simplification of the TNKF update

This section investigates the possibilities for further simplification of the TNKF update step. Firstly, the principal angles between the orthonormal bases of the state and covariance are computed, as described in the previous chapter. This subsection answers the fifth research question by investigating to what extent the TT(m) cores of the state and covariance change through TNKF operations. Subsequently, the performance and speed of the Element Update TNKF are shown for the reconstruction of frames when the covariance matrix is fixed to the predicted covariance matrix. Here, conclusions are drawn about the importance of the covariance matrix for the performance of the algorithm. This answers the sixth and last research question.

5-4-1 Orthonormal bases check

After computing the singular values of the orthonormal bases of the state with its summed term $\mathbf{U}^T_{\mathbf{x}_{j-1}}\mathbf{U}_{\mathbf{k}_j v_j}$ and the covariance with its subtracted term $\mathbf{U}^T_{\mathbf{P}_{j-1}}\mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j}$, we check to what extent the relations in (4-12) hold. To obtain the angles, the singular value vector **s** is computed using the `svd` command in MATLAB. These values are then summed and divided by the length of the vector (`sum(s)/length(s)`) to obtain the average singular value. This is done for the first 100 iterations of the reconstruction of a frame using the Element Update TNKF. The results are plotted in Figure 5-19.

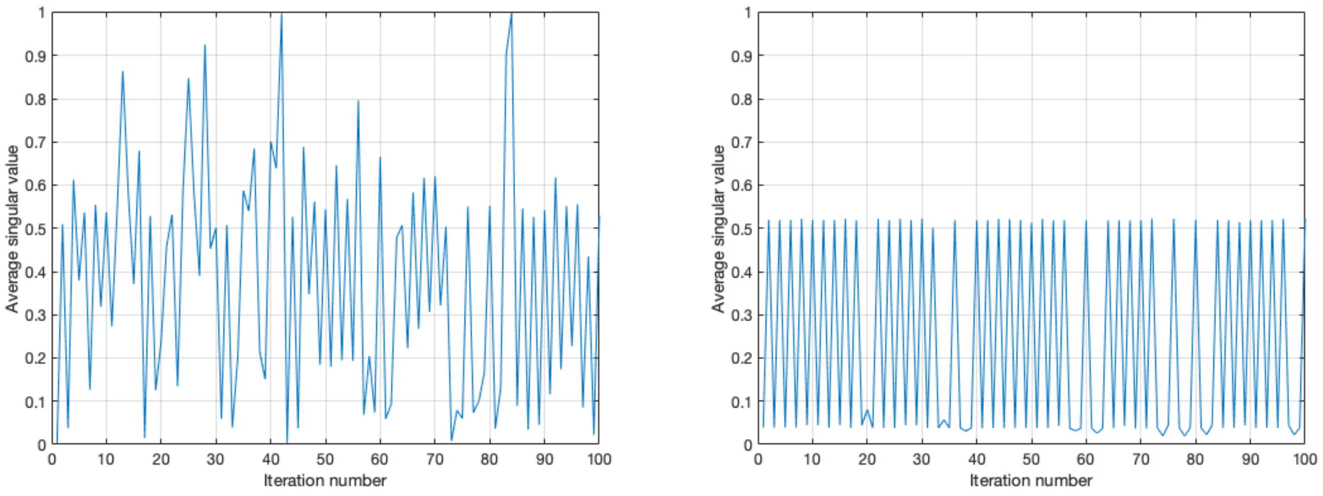


Figure 5-19: The average singular values of $\mathbf{U}^T_{\mathbf{x}_{j-1}}\mathbf{U}_{\mathbf{k}_j v_j}$ (left) and $\mathbf{U}^T_{\mathbf{P}_{j-1}}\mathbf{U}_{\mathbf{k}_j s_j \mathbf{k}_j}$ (right). Element Update TNKF, Campus data set.

From both plots, it is clear that the direction of the added or subtracted terms differs from the direction of the state vector and covariance matrix. For the state, the average singular value seems to rise to 1 on some occasions. This indicates that during that iteration, the principal angles between both orthonormal bases are zero, meaning these cores do not change in direction. Generally, the average singular value is relatively low, showing that the angles between the state and added term are non-zero.

For the covariance, the average singular value seems to jump from close to zero to about 0.5, which is repeated through almost all iterations. In the Element Update TNKF, the ranks of the covariance and the subtracted term jump between values of 1 and 2 for these settings. When the ranks of both TTm's have a value of 1, we see a low average singular value. When the ranks of both TTm's have a value of 2, we notice about half of the singular values to be equal to 1, indicating these principal angles are zero. To see to what extent these summations influence the directions, we compute the average singular values of $\mathbf{U}_{\mathbf{x}_{j-1}}^T \mathbf{U}_{\mathbf{x}_j}$ and $\mathbf{U}_{\mathbf{P}_{j-1}}^T \mathbf{U}_{\mathbf{P}_j}$ during 100 TNKF iterations. The results are plotted in Figure 5-21.

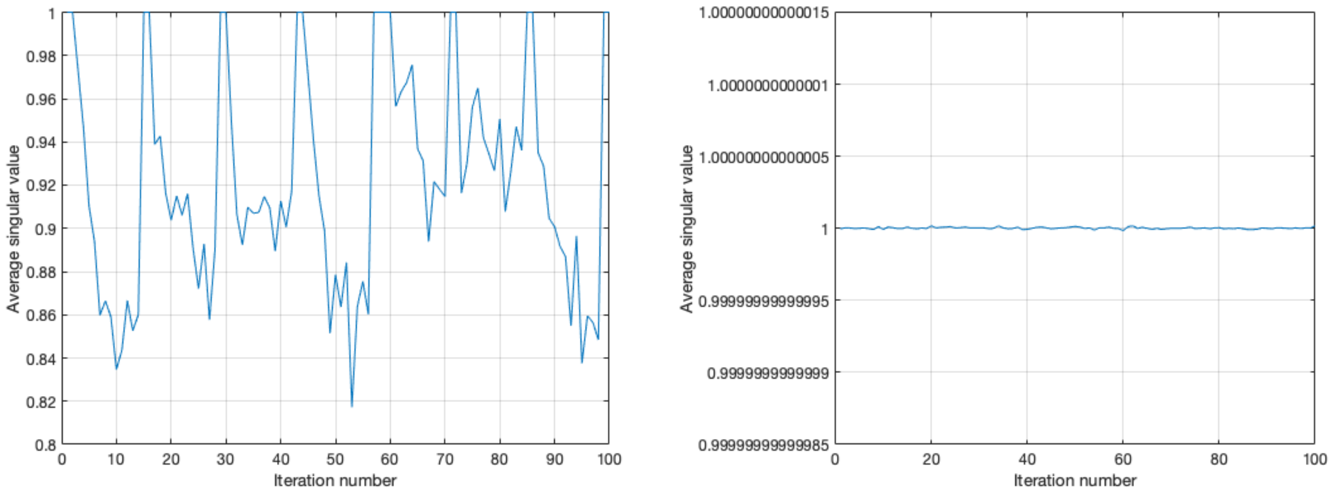


Figure 5-20: The singular values of $\mathbf{U}_{\mathbf{x}_{j-1}}^T \mathbf{U}_{\mathbf{x}_j}$ (left) and $\mathbf{U}_{\mathbf{P}_{j-1}}^T \mathbf{U}_{\mathbf{P}_j}$ (right) of 100 TNKF iterations. Element Update TNKF, Campus data set.

From the left plot, it is observed that the addition seems to influence the direction of the orthonormal bases of the state, as the average singular value varies. Interestingly, this amount seems to stay at a relatively high level as `sum(s)/length(s)` does not drop to low values. This indicates that although the direction of the orthonormal bases changes, this change seems relatively minor. Furthermore, after each rounding step, the average singular value jumps to 1, showing that the principle angles are then zero. Although the direction of the added term differs significantly from the state vector itself, as shown in the left plot of Figure 5-19, this does not seem to influence the direction of the state vector considerably.

In the right plot of Figure 5-21, there seems to be no single zero value present in the singular value vectors during all iterations. The fluctuations that are seen are results of machine precision in MATLAB. This means that although the covariance matrix changes due to the subtraction, this does not seem to influence the direction of the orthonormal bases. This could mean that the update of the covariance matrix can be simplified. To see the direction of the state and covariance matrix throughout the reconstruction of a whole frame, we plot the singular values for all iterations. Results are shown in Figure 5-21.

Again, although the state has drops in the singular values, the covariance matrix stays at 1. This means that updating the covariance matrix using the measured pixels does not influence the direction of the considered cores of the orthogonalized TTm. Since we computed the

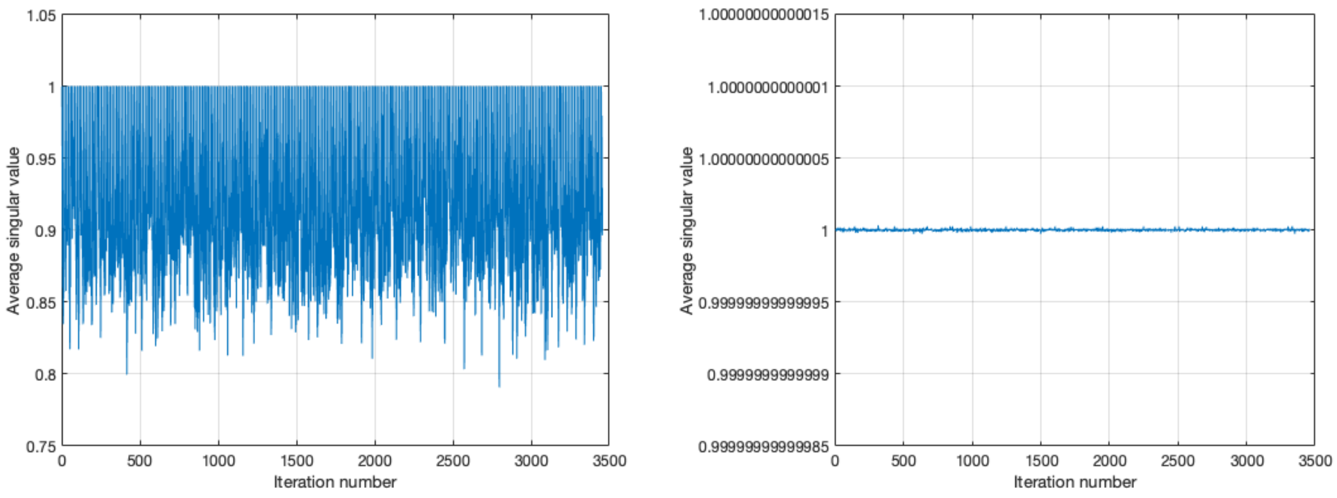


Figure 5-21: The singular values of $\mathbf{U}_{\mathbf{x}_{j-1}}^T \mathbf{U}_{\mathbf{x}_j}$ (left) and $\mathbf{U}_{\mathbf{P}_{j-1}}^T \mathbf{U}_{\mathbf{P}_j}$ (right) of all TNKF iterations of one reconstructed frame. Element Update TNKF, Campus data set.

orthonormal bases of all but the last core of the TTm, it could still be the case that the last core does contain a change in direction. This could mean that the first cores do not have to be accounted for when subtracting the $\mathbf{k}_j s_j \mathbf{k}_j$ term from the covariance, as explained in the previous chapter. When this change in the last core of the covariance TTm is minimal, we could consider skipping the update of the covariance matrix and do the TNKF computations for the state with the predicted covariance matrix. In the next section, this will be further investigated.

5-4-2 Skipping the covariance matrix update

To see what influence the update of the covariance matrix has, we can skip the update step and use the predicted covariance matrix during computations. For this analysis, the Element Update TNKF is used since the covariance matrix can be kept more exact as explained in the first section of this chapter. If this influence turns out to be minimal, we can drastically reduce the computational load. This is because then, the computation of the term $\mathbf{k}_j s_j \mathbf{k}_j^T$ can be omitted and the covariance matrix does not have to be rounded during the update phase of the TNKF. In Figure 5-22, the performance is compared between a fixed (predicted) covariance matrix and an updated covariance matrix for the Campus data set.

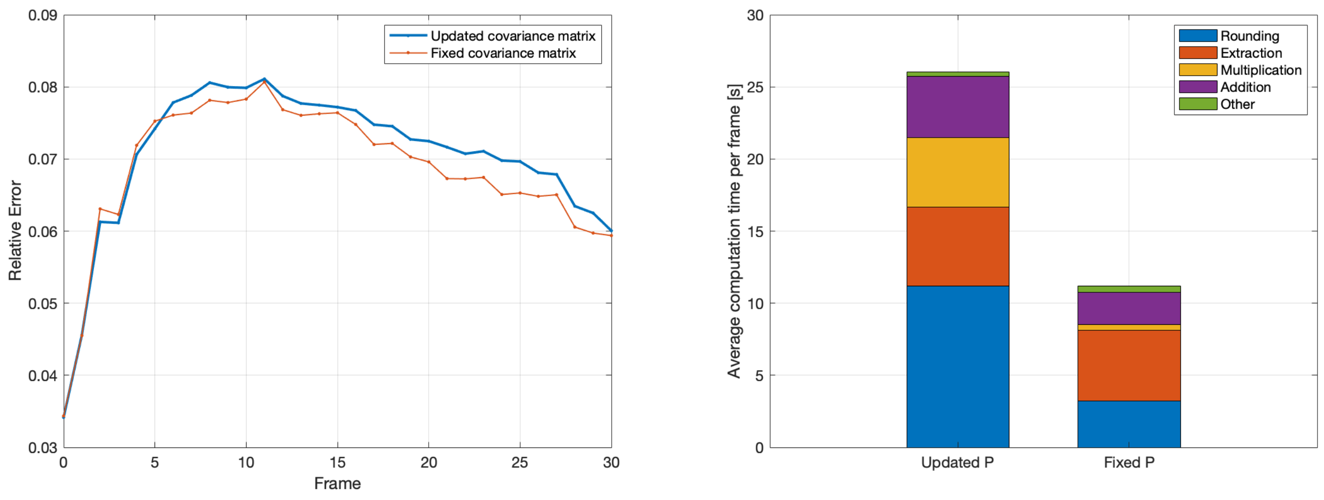
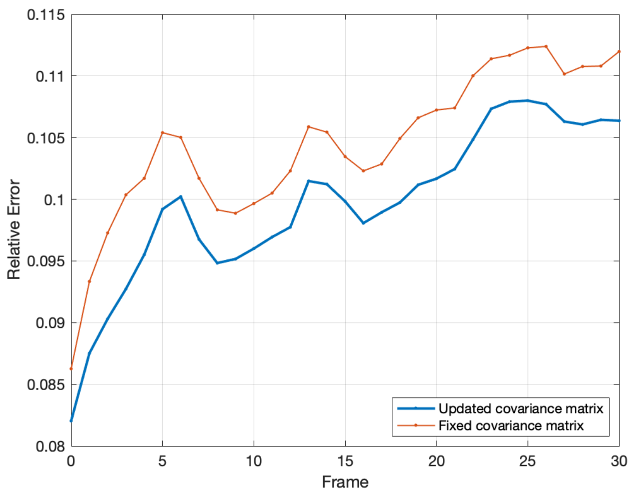


Figure 5-22: Relative error (left) and average computation time per frame (right) of reconstructed frames for fixed and updated covariance matrix. Element Update TNKF, Campus data set.

It is immediately observed that for these settings and sequences the relative error is not compromised when skipping the covariance update and doing the computations using the predicted covariance. Updating the covariance matrix does not result in a better performance for this data set. This could be because there are not many moving parts in the frame, and the missing pixel percentage is high (99%). This means the amount of measured pixels that change over time is low, which does not change the covariance matrix significantly during the update in the TNKF. Looking at the right plot of Figure 5-22, we see that the rounding, multiplication and addition operations have all reduced in computation time. The achieved speedup is +**132**%. To see if skipping the covariance matrix update can be done for a video with more moving parts, we carry out the same test for the Grand Central Station data set. The results are plotted in Figure 5-23.

For this data set, a speedup of +**121**% is realized. The difference in relative error is now more substantial, which could be the result of a lower missing pixel percentage (95%). This results in more measurements, which change the covariance matrix through the update iterations. Further lowering this missing pixel percentage makes this difference in relative error larger.

Interesting conclusions can be drawn when comparing these results with the performance



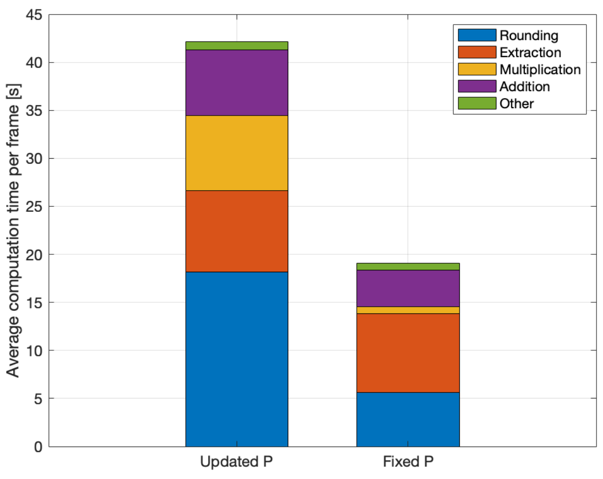


Figure 5-23: Relative error (left) and average computation time per frame (right) of reconstructed frames for fixed and updated covariance matrix. Element Update TNKF, Grand Central Station data set.

when using randomized rounding algorithms. The state-of-the-art comparison in the previous section shows that using randomized rounding for the rank truncation of the covariance matrix gives poor results. In fact, it is more beneficial to skip the update of the covariance matrix instead of using randomized rounding on the covariance matrix. This results in a drop in performance error that is smaller compared to the drop from randomized rounding and gives a much larger speedup.

Since the frame rate in both data sets is high, assuming the covariance to be equal to the predicted matrix works well. This is the case as there is not much change between two frames. Apparently, equalling the change in covariance of the next frame to the predicted covariance $\mathbf{P}_0 = \mathbf{P}[k-1] + \mathbf{W}[k]$ gives relatively good results. When the frame rate is lower or when objects are moving faster through the frame, however, skipping the update of the covariance matrix could not be beneficial as the measured pixels will have a stronger influence on changing the covariance matrix. This is also the case for lower percentages of missing pixels. As more measured pixels are located in moving parts of the frame, the covariance matrix becomes more and more important.

In Table 5-6, the achieved speedups by skipping the covariance matrix update are summarized for all simulations. The percentages, ranging from **+120%** to **+132%**, show that skipping the computation of the covariance update yields a very significant speedup. Again, this does result in a compromise in performance. When the performance is not of the highest importance, skipping the covariance update could be a solution in speeding up computations considerably.

Data set (Frame #)	**Missing pixel percentage**	**Average relative error difference**	**Speedup percentage**
Campus (50 - 80)	99%	1.8e-03	**+132** ± **4.5%**
Campus (200 - 230)	99%	-2.7e-03	**+124** ± **7.6%**
Grand Central Station (50 - 80)	95%	-4.8e-03	**+121** ± **3.9%**
Grand Central Station (200 - 230)	99%	-5.1e-03	**+120** ± **3.5%**

Table 5-6: Average relative error differences and speedups for a fixed covariance matrix compared to an updated covariance matrix. Element Update TNKF.

Chapter 6

Conclusions & Recommendations for Future Research

From the previous chapter, it is clear that all proposed methods can (significantly) speed up computations. In this chapter, conclusions are drawn on each speedup method and recommendations are made for further research into improving the computational efficiency of the TNKF algorithm.

6-1 Conclusions

Block Update TNKF

By reducing the TNKF update iterations with a factor of the block size B, the Block Update TNKF reduces the computational load of operations such as TT(m) multiplication and addition. The fastest block size for the video data and settings is concluded to be $B = 6$. The value of this fastest block size depends on how the other TNKF parameters are chosen but is always bounded as a consequence of the rounding procedure. With the block size $B = 6$, speedups of $\mathbf{+13\%}$ t0 $\mathbf{+21\%}$ were achieved. Further truncating the extracted covariance columns ranks can result in even larger speedups. Using this structure for the computation of the TNKF update equations has disadvantages, however. Since the ranks of the covariance matrix increase with a number of B^2 each iteration, these must be rounded back to rank 1 in the same iteration. Aside from increasing the computational load per rounding step, this overshoot of the ranks also means more information from the covariance is discarded. For data sets with many moving parts, this results in a considerable difference in performance compared to the Element Update TNKF. The usage of a Block Update TNKF raises a trade-off between computational speed and performance. There could be cases where performance is not of the highest priority. For example, when the percentage of missing pixels is lower, and the amount of moving parts in the frame is limited. In these cases, the Block Update TNKF can significantly reduce the computational load while still being able to reconstruct the frames. The Element Update TNKF does not have the option to shift this trade-off more towards improved computational speed.

Randomized rounding algorithms

By reducing the computational load of each rounding function call, randomized rounding algorithms can speed up the most computationally expensive process of the TNKF. When rounding the state TT with the fastest randomized rounding algorithm, RtO, total speedups of **+10%** to **+15%** can be achieved. If the moving objects in the to-be-reconstructed video are not too small, the randomized rounding algorithms can achieve a similar performance compared to deterministic rounding. When rounding the covariance matrix with these algorithms, problems can occur. This matrix can lose its positive definite property due to the randomization, destabilizing the TNKF. When stability is preserved, the randomization can increase the error of the covariance matrix to a level that dramatically influences the performance. When this is the case, the predicted covariance matrix is generally a better estimate. Again, the usage of randomized rounding algorithms comes down to a trade-off between speed and performance. Randomized rounding algorithms could be an option for rounding the state TT when the observer settles for a slight drop in performance.

Simplification of TNKF update

Since the TNKF updates the state and covariance for each measured pixel, simplifying the TNKF update equations could result in large speedups. After inspecting the directions of the orthogonal bases of the state and covariance, it is concluded that state TT changes throughout all TNKF iterations. The covariance matrix shows no change in the principal angles of its orthogonal bases, hinting that the update of the covariance matrix can be further simplified. Skipping the covariance matrix update drastically increases the computational speed of the algorithm. During experiments, the speedups ranged from **+120%** to **+132%**. When the amount of measured pixels in moving parts of the frame is relatively low, this measure can be taken without compromising on performance. When this amount is higher, there will be a larger difference in relative error. These characteristics of the video feed, including the frame rate, decide if skipping the covariance matrix update can be considered.

All methods considered in this research show a speedup compared to the state-of-the-art. Generally, these speedups also come with a decrease in reconstruction accuracy. As mentioned in the introduction of this research, the goal is to reduce the computational load of the TNKF to a level that makes real-time reconstruction possible with a regular computer. With the implementation of these methods, the average computation time per frame reduces but remains too large to reach this goal. From an efficiency point of view, the proposed methods are a step in the right direction. By reducing the energy consumption of the hardware used to run the TNKF algorithm, costs and carbon emissions are also reduced.

6-2 Recommendations for future research

Addressed topics

Because of the large rank increase of the covariance TTm, the Block Update TNKF has limitations on its performance and speed. If the extracted columns from the covariance TTm could be constructed in such a way that they can be represented by a rank 1 TTm, this problem would not occur. During the update, the ranks of the covariance TTm will then increase by the same number compared to the Element Update TNKF. This eliminates rank overshoot problem of the Block Update TNKF, giving it a similar performance as the Element Update TNKF. A possible direction for further research could be in representing the extracted columns of the covariance matrix more efficiently.

The implementation of randomized rounding algorithms turned out to be problematic for rounding the covariance matrix in TTm format. Since this TTm must be rounded to rank 1 to ensure efficient operations, the randomization 'distorts' the covariance that results in bad performance or even TNKF instability. In this thesis, some solutions are shown to tackle this instability. In future work, more investigation can be done into how this instability can be solved and how randomized rounding can be used for the covariance. The randomized rounding of the state showed promising results in terms of computational speed, but did compromise the performance. One of the causes of this was the distortion in the columns of moving parts of the frame. In future research, this column distortion could be prevented by fixing the values of the pixels located in the 'background' of the frame. This could bring the performance of the randomized rounding of the state closer to deterministic rounding. The algorithms presented in [10] are generalizations of randomized low-rank matrix approximation algorithms. In future work, more of these algorithms could be developed. The results in this thesis indicate that implementing these for the rounding of the state can be relevant.

The last section of Chapter 5 shows that the principal angles between the orthogonal bases of the covariance matrix do not change during TNKF iterations. This means that it might be possible to drastically simplify the update of the covariance matrix: $\mathbf{P}_j = \mathbf{P}_{j-1} - \mathbf{k}_j s_j \mathbf{k}_j^T$. Using these orthogonal bases, it could be possible to perform the subtraction of both terms in TTm-format element-wise. How this could be done is explained in Chapter 4. This element-wise subtraction does not increase the ranks of the covariance TTm, meaning rounding of the covariance matrix is unnecessary. As mentioned earlier, rounding the covariance matrix is the bottleneck in the TNKF algorithm. Finding a way to eliminate the need for this process could speed up computations considerably. The rounding of the covariance matrix is also the bottleneck in the Block Update TNKF algorithm. If the ranks of the covariance matrix in TTm format do not increase, this also eliminates the discussed rank overshoot problem. Then, using a Block Update TNKF will further speed up the operations without compromising on performance. When the performance is not of the highest priority, randomized rounding of the state TT can then also be combined with these methods. This combination could result in speedups that bring the algorithm closer to the goal of real-time reconstruction.

Additional topics

Aside from further researching the topics addressed in this thesis, other topics can be considered that could speed up TNKF operations. One of these topics is the usage of Graphics Processing Unit (GPU) arrays [29]. Currently, all computations are done using a computer's Central Processing Unit (CPU). In modern computers, this unit consists of multiple processor cores which can perform computational tasks in parallel. A GPU, however, consists of hundreds of smaller processor cores. In [9], it is shown that operations in TT-format such as rounding, addition and multiplication are parallelizable, and this can result in significant speedups. Using the GPU cores and parallelizing the TNKF code could thus speed up computations considerably. A downside to this approach could be that the access to memory is more computationally expensive when using the GPU. Data must be sent and retrieved between the CPU and GPU during computations, so the achievable speedup is limited by the amount of data transfer in the algorithm. Since the TNKF update requires a lot of this memory access during the update, it could be that the computational costs of data transfer prevent significant speedups.

Appendix A

Algorithms

Algorithm 10: Selection matrices TT [19]

```
Data: $d$-dimensional tensor $\boldsymbol{\mathcal{A}}$ in TT-format $\boldsymbol{\mathcal{A}} = \langle\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$
      with ranks $\mathbf{r} = [R_0, \ldots, R_d]$, value index vector $\mathbf{v} \in \mathbb{R}^K$
Result: Selection matrices $\{\mathbf{S}_1, \ldots, \mathbf{S}_d\}$
$\mathbf{i} = [I_1, I_2, \ldots, I_n]^T$
for $n = 1 : K$ do
    $\mathbf{C}$ = ind2sub($\mathbf{i}, \mathbf{v}(n)$) ;    // Convert linear index to multidimensional index
end
for $k = 1 : d$ do
    $m = \mathbf{i}(k)$
    $\mathbf{B}$ = eye($m$)
    for $l = 1 : K$ do
        $\mathbf{E}(:, k) = \mathbf{B}(:, \mathbf{C}(l, k))$
    end
    $\mathbf{S}_k = \mathbf{E}$
end
```

Algorithm 11: Selection matrices TTm [19]

```
Data: $d$-dimensional tensor $\boldsymbol{\mathcal{A}}$ in TTm-format $\boldsymbol{\mathcal{A}} = \langle\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\rangle$ (where
      $\boldsymbol{\mathcal{A}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ for $n = 1, \ldots, d$) with ranks $\mathbf{r} = [R_0, \ldots, R_d]$, value index
      vector $\mathbf{v} \in \mathbb{R}^K$
Result: Selection matrices $\{\mathbf{S}_1, \ldots, \mathbf{S}_d\}$
1  $\{\mathbf{S}_1, \ldots, \mathbf{S}_d\}$ = SelectionMatricesTTm($\boldsymbol{\mathcal{A}}, \mathbf{v}$) ;                 // Algorithm 9
2  $\mathbf{j} = [J_1, J_2, \ldots, J_n]^T$
3  for $n = 1 : K$ do
4      $\mathbf{C}$ = ind2sub($\mathbf{j}, \mathbf{v}(n)$) ;    // Convert linear index to multidimensional index
5  end
6  for $k = 1 : d$ do
7      $m = \mathbf{j}(k)$
8      $\mathbf{B}$ = eye($m$)
9      for $l = 1 : K$ do
10         $\mathbf{E}(:, k) = \mathbf{B}(:, \mathbf{C}(l, k))$
11     end
12     $\mathbf{S}_k = \mathbf{E}$
13 end
```

Algorithm 12: TT value extraction

```
Data: $d$-dimensional tensor $\boldsymbol{\mathcal{A}}$ in TT-format $\boldsymbol{\mathcal{A}} = \langle\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\rangle \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$
      with ranks $\mathbf{r} = [R_0, \ldots, R_d]$, value index vector $\mathbf{v} \in \mathbb{R}^K$
Result: Vector $\mathbf{b}$ with desired extracted values
1  $\{\mathbf{S}_1, \ldots, \mathbf{S}_d\}$ = SelectionMatricesTT($\boldsymbol{\mathcal{A}}, \mathbf{v}$) ;                 // Algorithm 9
2  $\mathbf{X} = \mathbf{S}_1^T \boldsymbol{\mathcal{A}}^{(1)}$
3  for $n = 2, 3, \ldots, d$ do
4      $\mathbf{Y} = \mathbf{S}_n \odot \mathbf{X}^T$
5      $\mathbf{Z}$ = reshape($\boldsymbol{\mathcal{A}}^{(n)}, R_n I_n, []$)
6      $\mathbf{X} = \mathbf{Y}^T \mathbf{Z}$
7  end
8  $\mathbf{b} = \mathbf{X}$
```

Algorithm 13: TTm value extraction

```
Data: $d$-dimensional tensor $\boldsymbol{\mathcal{A}}$ in TTm-format $\boldsymbol{\mathcal{A}} = \langle\!\langle \boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \dots, \boldsymbol{\mathcal{A}}^{(d)} \rangle\!\rangle$ (where
      $\boldsymbol{\mathcal{A}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ for $n = 1, \dots, d$) with ranks $\mathbf{r} = [R_0, \dots, R_d]$, value index
      vector $\mathbf{v} \in \mathbb{R}^K$
Result: Matrix $\mathbf{B}$ with desired extracted values
1  $\{\mathbf{S}_1, \dots, \mathbf{S}_d\}$ = SelectionMatricesTTm($\boldsymbol{\mathcal{A}}, \mathbf{v}$) ;                    // Algorithm 9
2  $\mathbf{C} = \mathbf{S}_1 \otimes \mathbf{S}_1$
3  $\mathbf{X}$ = permute($\boldsymbol{\mathcal{A}}^{(1)}, [1, 4, 2, 3]$)
4  $\mathbf{Y}$ = (reshape($\mathbf{X}, R_1 R_2, I_1 J_1$))$\mathbf{C}$
5  for $n = 2, 3, \dots, d-1$ do
6      $\mathbf{C} = \mathbf{S}_n \otimes \mathbf{S}_n$
7      $\mathbf{Z} = \mathbf{C} \odot \mathbf{Y}$
8      $\mathbf{X}$ = permute($[4, 1, 2, 3]$)
9      $\mathbf{Y}$ = (reshape($\mathbf{X}, R_{n+1}, R_n I_n J_n$))$\mathbf{Z}$
10 end
11 $\mathbf{C} = \mathbf{S}_d \otimes \mathbf{S}_d$
12 $\mathbf{Z} = \mathbf{C} \odot \mathbf{Y}$
13 $\mathbf{X} = \mathbf{Z}^T$(reshape($\boldsymbol{\mathcal{A}}^{(d)}, [], 1$))
14 $\mathbf{B}$ = reshape($\mathbf{X}, K, K$)
```

Algorithm 14: TTm column extraction

```
Data: $d$-dimensional tensor $\mathcal{A}$ in TTm-format $\mathcal{A} = \langle\langle \mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \dots, \mathcal{A}^{(d)} \rangle\rangle$ (where
      $\mathcal{A}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$ for $n = 1, \dots, d$) with ranks $\mathbf{r} = [R_0, \dots, R_d]$, column index
      vector $\mathbf{v} \in \mathbb{R}^K$, maximum extracted column rank $R_c^{\max}$
Result: Matrix with desired extracted columns $\mathcal{B}$ in TTm-format
        $\mathcal{B} = \langle\langle \mathcal{B}^{(1)}, \mathcal{B}^{(2)}, \dots, \mathcal{B}^{(d)} \rangle\rangle$ with ranks $\mathbf{h} = [H_0, \dots, H_d]$
1  $\{\mathbf{S}_1, \dots, \mathbf{S}_d\}$ = SelectionMatricesTTm($\mathcal{A}, \mathbf{v}$) ;          // Algorithm 9
2  $\mathbf{X}$ = permute($\mathcal{A}^{(1)}, [1, 2, 4, 3]$) ;                      // Compute first core
3  $\mathbf{Y}$ = reshape($\mathbf{X}, [], J_1$)$\mathbf{S}_1$
4  $\mathbf{Z} = \mathbf{S}_2 \odot \mathbf{Y}$
5  Permute and reshape $\mathbf{Z}$ to $\mathbf{Z} \in \mathbb{R}^{I_1 K \times R_1 J_2}$
6  $\mathbf{X}$ = permute($\mathcal{A}^{(1)}, [1, 3, 2, 4]$)
7  $\mathbf{Y} = \mathbf{Z}$(reshape($\mathbf{X}, R_1 J_2, []$))
8  Permute and reshape $\mathbf{Y}$ to $\mathbf{Y} \in \mathbb{R}^{I_1 \times I_2 R_2 K}$
9  $[\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}]$ = svd($\mathbf{Y}$, 'econ')
10 Truncate ranks if necessary: $\mathbf{U} = \mathbf{U}(:, 1 : R_c^{\max})$, $\mathbf{\Sigma} = \mathbf{\Sigma}(1 : R_c^{\max}, 1 : R_c^{\max})$,
    $\mathbf{V} = \mathbf{V}(:, 1 : R_c^{\max})$
11 $H_1$ = size($\mathbf{U}, 2$)
12 $\mathcal{B}^{(1)}$ = reshape($\mathbf{U}, 1, I_1, 1, H_1$)
13 $\mathbf{Z} = \mathbf{\Sigma}\mathbf{V}^T$
14 for $n = 2, 3, \dots, d-1$ do
15     $\mathbf{Z}$ = reshape($\mathbf{Z}, H_{n-1}, I_n, R_n, K$) ;                  // Compute middle cores
16     $\mathbf{X}$ = reshape($\mathbf{Z}, [], K$)
17     $\mathbf{Y} = \mathbf{S}_{n+1} \odot \mathbf{X}$
18     Reshape and permute $\mathbf{Y}$ to $\mathbf{Y} \in \mathbb{R}^{H_{n-1} I_n K \times R_n J_{n+1}}$
19     $\mathbf{Z}$ = permute($\mathcal{A}^{(n+1}, [1, 3, 2, 4]$)
20     $\mathbf{X} = \mathbf{Y}$(reshape($\mathbf{Z}, R_n J_{n+1}, []$))
21     Reshape and permute $\mathbf{X}$ to $\mathbf{X} \in \mathbb{R}^{H_{n-1} I_n \times I_{n+1} R_{n+1} K}$
22     $[\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}]$ = svd($\mathbf{X}$, 'econ')
23     Truncate ranks if necessary: $\mathbf{U} = \mathbf{U}(:, 1 : R_c^{\max})$, $\mathbf{\Sigma} = \mathbf{\Sigma}(1 : R_c^{\max}, 1 : R_c^{\max})$,
        $\mathbf{V} = \mathbf{V}(:, 1 : R_c^{\max})$
24     $H_n$ = size($\mathbf{U}, 2$)
25     $\mathcal{B}^{(n)}$ = reshape($\mathbf{U}, H_{n-1}, I_n, 1, H_n$)
26     $\mathbf{Z} = \mathbf{\Sigma}\mathbf{V}^T$
27 end
28 $\mathcal{B}^{(d)}$ = reshape($\mathbf{Z}, H_{d-1}, I_d, K, 1$) ;                  // Compute last core
```

Algorithm 15: Left-to-right partial contraction [10]

```
Data: Tensors $\mathcal{A}$ and $\mathcal{B}$ with consistent dimensions in TT format and ranks
      $\mathbf{r}_{\mathcal{A}} = [R_0, \dots, R_d]$ and $\mathbf{r}_{\mathcal{B}} = [S_0, \dots, S_d]$
Result: Partial contraction matrices $\{\mathbf{W}_1, \dots, \mathbf{W}_{d-1}\}$
1 $\{\mathbf{W}_1, \dots, \mathbf{W}_{d-1}\} = \texttt{PartialContractionsLR}(\mathcal{A}, \mathcal{B})$
2 $\mathbf{W}_1 = \mathcal{A}^{(1)^T} \mathcal{B}^{(1)}$
3 for $n = 2, \dots, d-1$ do
4     $\mathbf{X} = \mathbf{W}_n(\texttt{reshape}(\mathcal{B}^{(n)}, S_n, []))$
5     $\mathbf{W}_n = (\texttt{reshape}(\mathcal{A}^{(n)}, [], R_{n+1}))^T(\texttt{reshape}(\mathbf{X}, [], S_{n+1}))$
6 end
```

Algorithm 16: Right-to-left partial contraction [10]

```
Data: Tensors $\mathcal{A}$ and $\mathcal{B}$ with consistent dimensions in TT format and ranks
      $\mathbf{r}_{\mathcal{A}} = [R_0, \dots, R_d]$ and $\mathbf{r}_{\mathcal{B}} = [S_0, \dots, S_d]$
Result: Partial contraction matrices $\{\mathbf{W}_1, \dots, \mathbf{W}_{d-1}\}$
1 $\{\mathbf{W}_1, \dots, \mathbf{W}_{d-1}\} = \texttt{PartialContractionsRL}(\mathcal{A}, \mathcal{B})$
2 $\mathbf{W}_{d-1} = \mathcal{A}^{(d)} \mathcal{B}^{(d)^T}$
3 for $n = d-1, \dots, 2$ do
4     $\mathbf{X} = \texttt{reshape}(\mathcal{A}^{(n)}, [], R_{n+1})\mathbf{W}_n$
5     $\mathbf{W}_{n-1} = (\texttt{reshape}(\mathbf{X}, R_n, []))(\texttt{reshape}(\mathcal{B}^{(n)}, S_n, []))^T$
6 end
```

Appendix B

Video Frames

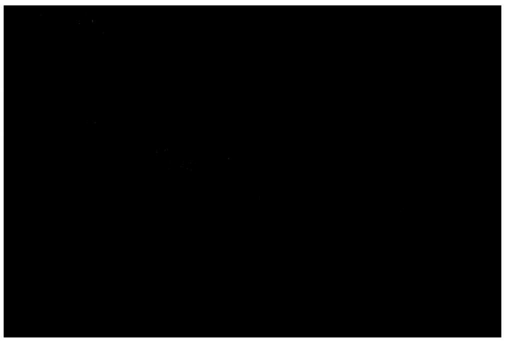


Figure B-1: Corrupted frame of the Campus data set, $\gamma = 0.99$.

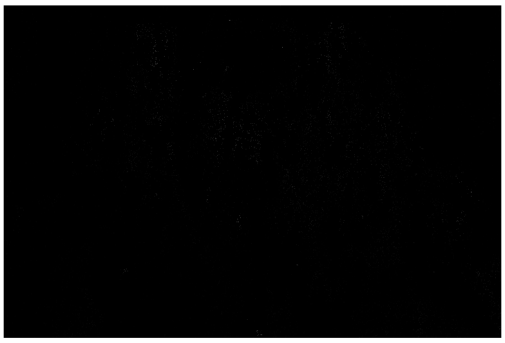
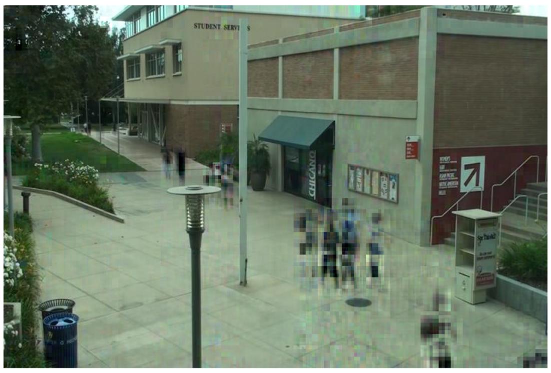
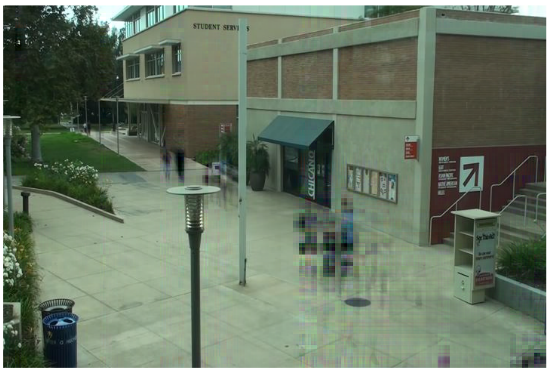


Figure B-2: Corrupted frame of the Grand Central Station data set, $\gamma = 0.95$.

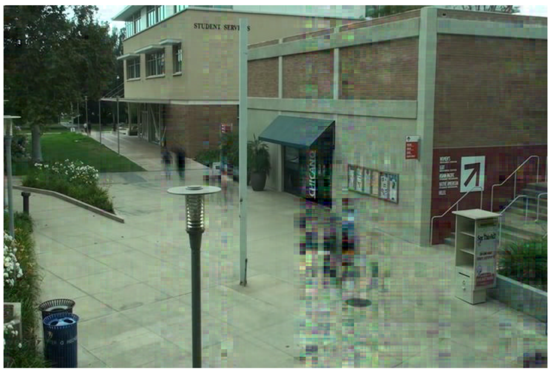
(a) Original frame



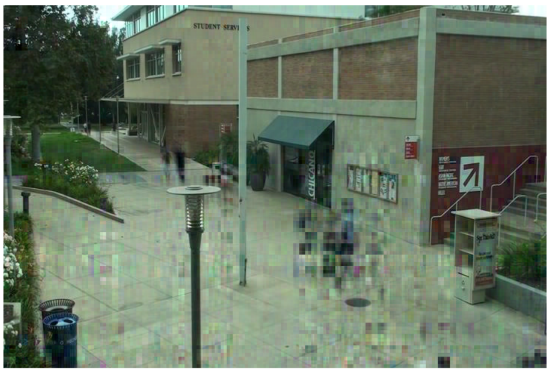
(b) Det, $\varepsilon = 5.95\text{e}-02$



(c) OtR, $\varepsilon = 5.34\text{e}-02$



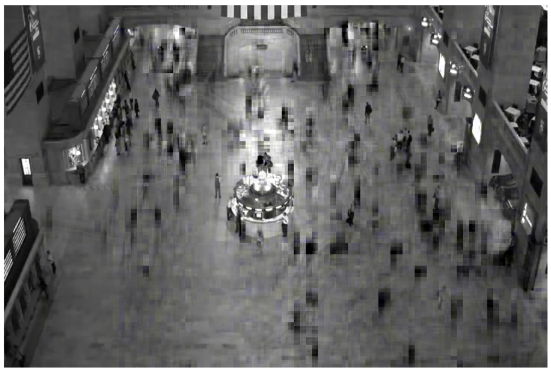
(d) RtO, $\varepsilon = 6.68\text{e}-02$



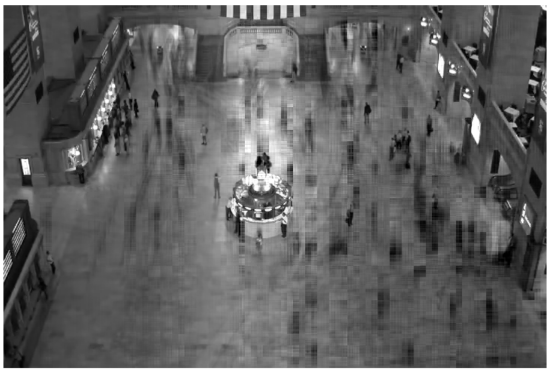
(e) TSR, $\varepsilon = 6.85\text{e}-02$

Figure B-3: 30th reconstructed frames using all rounding algorithms for rounding the state TT, including the corresponding relative errors. Block Update TNKF, deterministic covariance rounding, default settings, Campus data set.

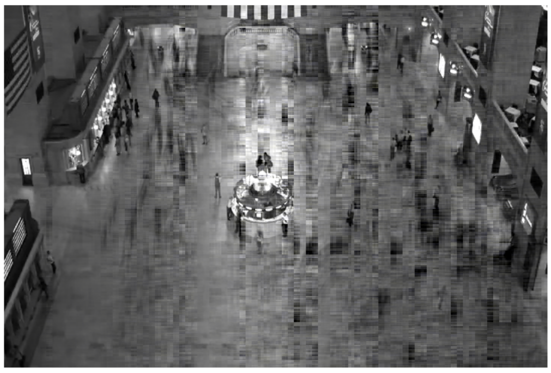
(a) Original frame



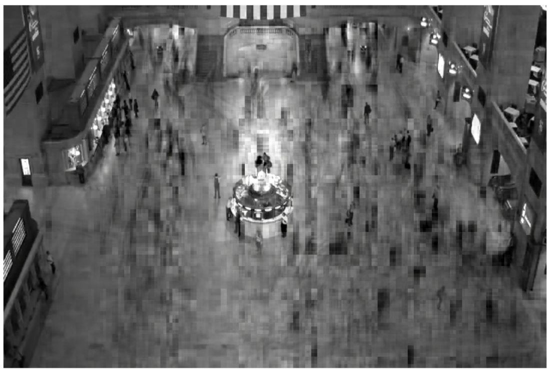
(b) Det, $\varepsilon = 1.12\text{e}-01$



(c) OtR, $\varepsilon = 1.18\text{e}-01$



(d) RtO, $\varepsilon = 1.31\text{e}-01$



(e) TSR, $\varepsilon = 1.26\text{e}-01$

Figure B-4: 30th reconstructed frames using all rounding algorithms for rounding the state TT, including the corresponding relative errors. Block Update TNKF, deterministic covariance rounding, default settings, Grand Central Station data set.

Bibliography

[1] Salman Ahmadi-Asl, Andrzej Cichocki, Anh Huy Phan, Maame G Asante-Mensah, Mirfarid Musavian Ghazani, Toshihisa Tanaka, and Ivan Oseledets. “Randomized algorithms for fast computation of low rank tensor ring model”. In: *Machine Learning: Science and Technology* 2 (1 Dec. 2020).

[2] Kim Batselier, Zhongming Chen, and Ngai Wong. A Tensor Network Kalman filter with an application in recursive MIMO Volterra system identification. 2017.

[3] Kim Batselier, Wenjian Yu, Luca Daniel, and Ngai Wong. “Computing low-rank approximations of large-scale matrices with the tensor network randomized SVD”. In: *SIAM Journal on Matrix Analysis and Applications* 39 (3 2018), pp. 1221–1244.

[4] Ben Benfold and Ian Reid. “Stable Multi-Target Tracking in Real-Time Surveillance Video”. In: *CVPR* (2011), pp. 3457–3464.

[5] Maolin Che and Yimin Wei. “Randomized algorithms for the approximations of Tucker and the tensor train decompositions”. In: *Advances in Computational Mathematics* 45 (1 Feb. 2019), pp. 395–428.

[6] Yilun Chen, Yuantao Gu, and Alfred O. Hero III. “Sparse LMS for System Identification”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing* (Apr. 2009), pp. 3125–3128.

[7] A. Cichocki, N. Lee, I. V. Oseledets, A. -H. Phan, Q. Zhao, and D. Mandic. “Low-Rank Tensor Networks for Dimensionality Reduction and Large-Scale Optimization Problems: Perspectives and Challenges: Part 1”. In: *Foundations and Trends in Machine Learning* 9 (4-5 Sept. 2016), pp. 249–429.

[8] Pau Closas, Carles Fernández-Prades, Jordi Vilà-Valls, Jordi Vilà-Valls Multiple Quadrature Kalman Filtering IEEE, and Senior Member. “Multiple Quadrature Kalman Filtering”. In: *IEEE Transactions on Signal Processing* 60 (12 2012), pp. 6125–6137.

[9] Hussam Al Daas, Grey Ballard, and Peter Benner. Parallel Algorithms for Tensor Train Arithmetic. Nov. 2020.

[10] Hussam Al Daas, Grey Ballard, Paul Cazeaux, Eric Hallman, Agnieszka Miedlar, Mirjeta Pasha, Tim W. Reid, and Arvind K. Saibaba. Randomized algorithms for rounding in the Tensor-Train format. Oct. 2021.

[11] 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. 2019.

[12] M. Fannes, B. Nachtergaele, and R. F. Werner. "Finitely Correlated States on Quantum Spin Chains". In: *Communications in Mathematical Physics* 144 (1992), pp. 443–490.

[13] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288.

[14] Simon Hinterholzer and Ralph Hintemann. Energy consumption of data centers worldwide How will the Internet become green? 2020.

[15] IPVM. *Average Frame Rate Video Surveillance Statistics*. 2021. URL: https://ipvm.com/reports/average-frame-rate-video-surveillance-2021.

[16] C. G. Khatri and C. R. Rao. "Solutions to some Functional Equations and their Applications to Characterization of Probability Distributions". In: *Sankhya: The Indian Journal of Statistics* (1968), pp. 167–180.

[17] Boris N. Khoromskij. "$O(d \log N)$-Quantics Approximation of N-d Tensors in High-Dimensional Numerical Modeling". In: *Constructive Approximation* 34 (2 Oct. 2011), pp. 257–280.

[18] P. Van Klaveren. Tensor-Networked Square-Root Kalman Filter for Online Video Completion. 2021.

[19] Ching-Yun Ko, Kim Batselier, Luca Daniel, Senior Member, Wenjian Yu, and Ngai Wong. Fast and Accurate Tensor Completion with Total Variation Regularized Tensor Trains. 2015.

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

[21] A.P. van Koppen. Improving the Computational Speed of a Tensor-Networked Kalman Filter for Streaming Video Completion - Literature Study. 2022.

[22] MATLAB. *Version R2020b*. The Mathworks Inc., Natick, Massachusetts, 2019.

[23] Yuji Nakatsukasa. Fast and stable randomized low-rank matrix approximation. Sept. 2020.

[24] I. V. Oseledets. "Approximation of 2d $\times$ 2d matrices using tensor decomposition". In: *SIAM Journal on Matrix Analysis and Applications* 31 (4 2009), pp. 2130–2145.

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

[26] R Penrose. "Applications of Negative Dimensional Tensors". In: *Combinatorial Mathematics and its Applications* (1971), pp. 221–244.

[27] Matti Raitoharju and Robert Piche. "On Computational Complexity Reduction Methods for Kalman Filter Extensions". In: *IEEE Aerospace and Electronic Systems Magazine* 34 (10 Oct. 2019), pp. 2–19.

[28] Matti Raitoharju, Robert Piché, Juha Ala-Luhtala, and Simo Ali-Löytty. Partitioned Update Kalman Filter. Mar. 2015.

[29] Jill Reese and Sarah Zaranek. *GPU Programming in MATLAB*. 2011. URL: https://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html.

[30] S. J. S. De Rooij. Streaming Video Completion using a Tensor-Networked Kalman Filter. 2020.

[31] Ulrich Schollwöck. "The density-matrix renormalization group in the age of matrix product states". In: *Annals of Physics* 326 (1 Jan. 2011), pp. 96–192.

[32] Ruchi Tripathi, Boda Mohan, and Ketan Rajawat. "Adaptive low-rank matrix completion". In: *IEEE Transactions on Signal Processing* 65 (14 July 2017), pp. 3603–3616.

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

[34] VIRAT. *VIRAT Video Dataset*. 2011. URL: https://viratdata.org/.

[35] Rui Zhao, Hengyu Li, Jingyi Liu, Huayan Pu, Shaorong Xie, and Jun Luo. "A video inpainting method for unmanned vehicle based on fusion of time series optical flow information and spatial information". In: *International Journal of Advanced Robotic Systems* 18 (5 2021).

[36] Peizhen Zhu and Andrew V. Knyazev. "Principal angles between subspaces and their tangents". In: *Journal of Numerical Mathematics* 21 (4 Sept. 2013), pp. 325–240.

Glossary

List of Acronyms

TNKF	Tensor Networked Kalman Filter
TT	Tensor-Train
TTm	Tensor-Train matrix
fps	frames per second
HD	High-Definition
RGB	red, green and blue
SVD	singular value decomposition
PLMS	Proximal Least Mean Squares
MPS	matrix product states
MGS	Modified Gram-Schmidt
PUKF	Partitioned Update Kalman Filter
OtR	Orthogonalize-then-Randomize

RtO	Randomize-then-Orthogonalize
TSR	Two-Sided-Randomization
flops	floating-point operations