Approximation von Tensoren

Katrin Almon
Geboren am 08.08.1987 in Konstanz

Barbara Fuchs
Geboren am 26.05.1988 in Bad Honnef

1. September 2010

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Mario Bebendorf

INSTITUT FÜR NUMERISCHE SIMULATION

MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT DER
RHEINISCHEN FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Inhaltsverzeichnis

1. Einleitung

In der vorliegenden Bachelorarbeit wird anhand mehrerer Methoden versucht, dreidimensionale Tensoren zu approximieren. Zum einen wird dabei die von L. R. Tucker Mitte der 1960er Jahre in [12] entwickelte *Tucker-Zerlegung* behandelt, zum anderen der kurze Zeit später von J. Douglas Carroll und Jih-Jie Chang in [7] vorgestellte *Alternating Least Squares Algorithmus* (ALS). Beide Methoden basieren auf einer Zerlegung des gegebenen Tensors in sogenannte *Faktormatrizen*.

Dabei stellt der Tucker Algorithmus eine Art Verallgemeinerung der Singulärwertzerlegung zweidimensionaler Matrizen für höherdimensionale Tensoren dar, während der ALS Algorithmus einen iterativen Charakter besitzt und versucht, *Least Squares Lösungen* für die Faktoren zu finden.

Ursprünglich stammt die Idee der Zerlegung höherdimensionaler Tensoren in einzelne Faktoren aus den Bereichen der Psychometrie und Chemometrie [7, 12], aber im Laufe der letzten Jahrzehnte hat sie ihre Anwendung in zahlreichen wissenschaftlichen Gebieten wie zum Beispiel in der Signalverarbeitung oder Ökonomie gefunden und auch auf mathematischer Ebene hat es wesentliche Fortschritte und Erkenntnisse zu diesem Thema gegeben, siehe [11].

Im Kontrast dazu steht die weitaus jüngere *Adaptive Cross Approximation* (ACA aus [4]). Bei dieser muss nicht auf alle Einträge des ursprünglichen Tensors zugegriffen werden, um eine Approximation desselbigen zu erhalten, sondern vielmehr werden rekursiv Approximationen hergeleitet, die auf der Struktur des Tensors beruhen. Aus diesem Grund ist die ACA nur für multivariate Funktionen geeignet und nicht für zufällige Datenstrukturen, wie sie zum Beispiel bei Bildern gegeben sind.

Das Ziel ist es, die vorgestellten Algorithmen zu vergleichen, sowie die Verwendbarkeit von dem Tucker Algorithmus und dem ALS Algorithmus für die Kompression von Video- und Bilddateien zu überprüfen.

Für die Arbeit wurden die beiden zuerst genannten Methoden sowohl für eine Anwendung auf trivariate Funktionen als auch für Bilder und kurze Videoausschnitte programmiert. Der Quellcode dazu befindet sich im Anhang.

Nach einer Einführung in die Notation mehrdimensionaler Tensoren und einigen wichtigen Eigenschaften werden zunächst der Tucker Algorithmus und der Alternating Least Squares Algorithmus im Detail vorgestellt.

Dabei wird unter anderem die Anwendung auf eine Bild-Sequenz behandelt. Die wesentlichen Schritte des daraus entstandenen Videoausschnittes sind in Kapitel 3.9 und 4.4 abgebildet. Es folgt ein Vergleich der beiden Methoden in Bezug auf die Laufzeiten und Komplexität, sowie eine Gegenüberstellung mit dem ACA Algorithmus. Hierbei wird der ACA Algorithmus in der Form verwendet, wie er in [4] für trivariate Funktionen vorgestellt wurde.

Abschließend werden in einem kurzen Fazit die erhaltenen Erkenntnisse zusammengefasst und beurteilt.

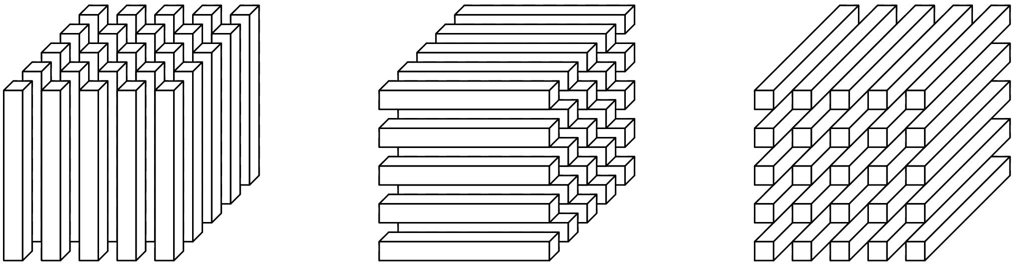
2. Grundlagen

2.1. Notation

Ein *Tensor* ist ein mehrdimensionales Array. Die *Ordnung* eines Tensors gibt die Anzahl seiner Dimensionen oder Modi an. Ein Tensor erster Ordnung ist ein Vektor und ein Tensor zweiter Ordnung ist eine Matrix. Besitzt ein Tensor die Ordnung drei oder größer, so wird er *Tensor höherer Ordnung* genannt.

Vektoren werden mit fettgedruckten Kleinbuchstaben bezeichnet, z.B. $\boldsymbol{a}$, Matrizen mit fettgedruckten Großbuchstaben, z.B. $\boldsymbol{A}$. Tensoren höherer Ordnung werden mit fettgedruckten Eulerskriptbuchstaben bezeichnet, z.B. $\boldsymbol{\mathcal{X}}$. Für Skalare werden wie gewöhnlich Kleinbuchstaben verwendet, z.B. a.

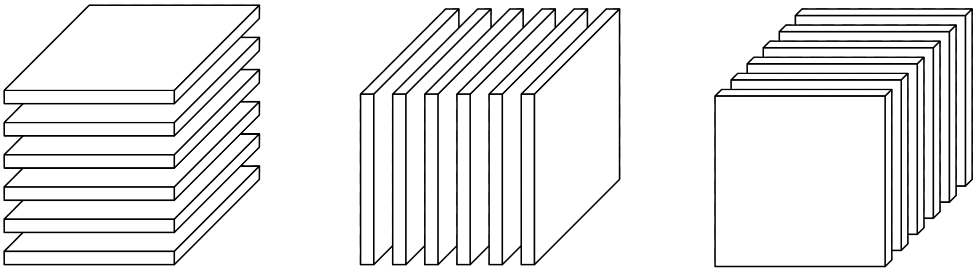
Der i-te Eintrag eines Vektors $\mathbf{a}$ wird mit a_i bezeichnet, Eintrag (i, j) einer Matrix $\boldsymbol{A}$ mit a_{ij} und Eintrag (i, j, k) eines Tensors $\boldsymbol{\mathcal{X}}$ dritter Ordnung mit x_{ijk}. Die Indizes laufen immer von 1 bis zu ihrem Großbuchstaben, z.B. $i = 1, \ldots, I$.



(a) Fasern 1. Art $x_{:jk}$ (b) Fasern 2. Art $x_{i:k}$ (c) Fasern 3. Art $x_{ij:}$

Abbildung 1: Fasern eines Tensors dritter Ordnung.

Für Tensoren dritter oder höherer Ordnung sind *Fasern* das Analogon zu Zeilen und Spalten einer Matrix. Eine Faser (Abbildung 1) wird dadurch definiert, dass bis auf einen Index alle anderen Indizes fixiert werden. Ist beispielsweise der erste Index i frei, so heißen diese Fasern Fasern erster Art oder Fasern des ersten Modus.



(a) horizontale Scheiben $\boldsymbol{X}_{i::}$ (b) laterale Scheiben $\boldsymbol{X}_{:j:}$ (c) frontale Scheiben $\boldsymbol{X}_{::k}$

Abbildung 2: Scheiben eines Tensors dritter Ordnung.

Scheiben sind zweidimensionale Teile eines Tensors, die man erhält, indem man alle bis auf

zwei Indizes fixiert. Abbildung 2 zeigt die horizontalen, lateralen und frontalen Scheiben eines Tensors $\boldsymbol{\mathcal{X}}$ dritter Ordnung. Insbesondere werden die einzelnen frontalen Scheiben mit $\boldsymbol{X}_k$ bezeichnet.
Diese Notation wird häufig verwendet, wie zum Beispiel in [10].

2.2. Transformation eines Tensors in eine Matrix

Die Elemente eines Tensors können so umsortiert werden, dass sie eine Matrix bilden. Beispielsweise kann ein $3 \times 4 \times 2$ Tensor in Matrizen der Größe 2×12, 3×8 und 4×6 umsortiert werden. Die Transformation eines Tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ in eine Matrix n-ter Art wird mit $\boldsymbol{X}_{(n)}$ bezeichnet und dabei werden die Fasern n-ter Art so sortiert, dass sie die Spalten der resultierenden Matrix bilden. Dabei wird das Tensorelement $(i_1, i_2, \ldots, i_N)$ auf ein Matrixelement (i_n, j) abgebildet mit

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) J_k \qquad \text{und} \qquad J_k = \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m .$$

Beispiel 1. *Die frontalen Scheiben von* $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{3 \times 4 \times 2}$ *seien*

$$\boldsymbol{X}_1 = \begin{bmatrix} 4 & 1 & 5 & 3 \\ 6 & 5 & 1 & 7 \\ 7 & 2 & 4 & 8 \end{bmatrix} \qquad \textit{und} \qquad \boldsymbol{X}_2 = \begin{bmatrix} 2 & 1 & 3 & 8 \\ 5 & 7 & 4 & 9 \\ 9 & 2 & 5 & 6 \end{bmatrix} .$$

Dann sind die drei Matrizen n-ter Art

$$\boldsymbol{X}_{(1)} = \begin{bmatrix} 4 & 1 & 5 & 3 & 2 & 1 & 3 & 8 \\ 6 & 5 & 1 & 7 & 5 & 7 & 4 & 9 \\ 7 & 2 & 4 & 8 & 9 & 2 & 5 & 6 \end{bmatrix} ,$$

$$\boldsymbol{X}_{(2)} = \begin{bmatrix} 4 & 6 & 7 & 2 & 5 & 9 \\ 1 & 5 & 2 & 1 & 7 & 2 \\ 5 & 1 & 4 & 3 & 4 & 5 \\ 3 & 7 & 8 & 8 & 9 & 6 \end{bmatrix} ,$$

$$\boldsymbol{X}_{(3)} = \begin{bmatrix} 4 & 6 & 7 & 1 & 5 & \ldots & 4 & 3 & 7 & 8 \\ 2 & 5 & 9 & 1 & 7 & \ldots & 5 & 8 & 9 & 6 \end{bmatrix} .$$

Außerdem ist

$$\operatorname{vec}(\boldsymbol{\mathcal{X}}) = \begin{bmatrix} 4 & 6 & 7 & 1 & ... & 4 & 5 & 8 & 9 & 6 \end{bmatrix}^T .$$

In manchen Artikeln, wie zum Beispiel in [12], werden andere Sortierungen für die Spalten einer Matrix n-ter Art benutzt. Die Ordnung der Spalten ist nicht von Bedeutung, solange sie in allen Berechnungen konsistent ist.

2.3. Multiplikation von Tensoren

Tensoren können miteinander multipliziert werden, allerdings sind Notation und Symbole komplexer als bei Matrixmultiplikationen. Wir betrachten hier nur das Produkt eines Tensors mit einer Matrix (bzw. einem Vektor).

Definition 1 (Produkt n-ter Art)**.** *Das Produkt n-ter Art eines Tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ mit einer Matrix $\boldsymbol{U} \in \mathbb{R}^{J \times I_n}$ wird mit $\mathcal{X} \times_n \mathbf{U}$ bezeichnet und hat die Größe $I_1 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N$. Elementweise gilt*

$$(\mathcal{X} \times_n \boldsymbol{U})_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} u_{j i_n}.$$

Wie man leicht sehen kann, ist es möglich, dieses Produkt auch durch ein Matrixprodukt darzustellen.

$$\mathcal{Y} = \mathcal{X} \times_n \boldsymbol{U} \qquad \Leftrightarrow \qquad \boldsymbol{Y}_{(n)} = \boldsymbol{U}\boldsymbol{X}_{(n)}.$$

Beispiel 2. *Sei $\mathcal{X} \in \mathbb{R}^{3\times4\times2}$ wie im vorherigen Beispiel und $\boldsymbol{U} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2\times3}$. Dann ist das Produkt $\mathcal{Y} = \mathcal{X} \times_1 \boldsymbol{U} \in \mathbb{R}^{2\times4\times2}$ gleich*

$$\boldsymbol{Y}_1 = \begin{bmatrix} 37 & 17 & 19 & 41 \\ 88 & 41 & 49 & 95 \end{bmatrix}, \qquad \boldsymbol{Y}_2 = \begin{bmatrix} 39 & 21 & 26 & 44 \\ 87 & 51 & 62 & 113 \end{bmatrix}.$$

Führt man solche Produkte mehrmals hintereinander aus, dann gelten folgende Rechenregeln:

Satz 1. *Für die Produkte eines Tensors $\mathcal{X}$ mit Matrizen $\boldsymbol{A}$ und $\boldsymbol{B}$ gilt*

$$(\mathcal{X} \times_m \boldsymbol{A}) \times_n \boldsymbol{B} = (\mathcal{X} \times_n \boldsymbol{B}) \times_m \boldsymbol{A} \qquad \textit{für } m \neq n \tag{1}$$

$$(\mathcal{X} \times_n \boldsymbol{A}) \times_n \boldsymbol{B} = \mathcal{X} \times_n (\boldsymbol{B}\boldsymbol{A}). \tag{2}$$

Beweis von (2).

$$\begin{array}{rrcll} & \mathcal{Y} & = & (\mathcal{X} \times_n \boldsymbol{A}) \times_n \boldsymbol{B} & \\ \Leftrightarrow & \boldsymbol{Y}_{(n)} & = & \boldsymbol{B}\left(\boldsymbol{A}\boldsymbol{X}_{(n)}\right) & = \; (\boldsymbol{B}\boldsymbol{A})\,\boldsymbol{X}_{(n)} \\ \Leftrightarrow & \mathcal{Y} & = & \mathcal{X} \times_n (\boldsymbol{B}\boldsymbol{A}) & \end{array}$$

Beweis von (1). Da bei endlichen Summen die Reihenfolge der Summanden vertauscht werden darf, gilt:

$$\begin{aligned}
((\mathcal{X} \times_m \boldsymbol{A}) \times_n \boldsymbol{B})_{i_1 \dots i_{m-1} j i_{m+1} \dots i_{n-1} k i_{n+1} \dots i_N} &= \sum_{i_n=1}^{I_n} \left(\sum_{i_m=1}^{I_m} x_{i_1 \dots i_N}\, a_{j i_m} \right) b_{k i_n} \\
&= \sum_{i_m=1}^{I_m} \left(\sum_{i_n=1}^{I_n} x_{i_1 \dots i_N}\, b_{k i_n} \right) a_{j i_m} \\
&= ((\mathcal{X} \times_n \boldsymbol{B}) \times_m \boldsymbol{A})_{i_1 \dots i_{m-1} j i_{m+1} \dots i_{n-1} k i_{n+1} \dots i_N} \\
&= (\mathcal{X} \times_n \boldsymbol{B} \times_m \boldsymbol{A})_{i_1 \dots i_{m-1} j i_{m+1} \dots i_{n-1} k i_{n+1} \dots i_N}\,.
\end{aligned}$$

□

2.4. Norm und inneres Produkt von Tensoren

Definition 2 (Norm)**.** *Die Norm eines Tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ ist die Wurzel aus der Summe der Quadrate der einzelnen Einträge, also*

$$\| \mathcal{X} \| = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \dots \sum_{i_N=1}^{I_N} x_{i_1 \dots i_N}^2}\,.$$

Diese Definition ist analog zur Frobeniusnorm einer Matrix $\boldsymbol{A}$, die mit $\|\mathbf{A}\|$ bezeichnet wird. Daher ist auch die Norm eines Tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ gleich der Frobeniusnorm seiner Transformationen n-ter Art in Matrizen:

$$\| \mathcal{X} \| = \left\| \boldsymbol{X}_{(n)} \right\| \qquad \text{für } n = 1, 2, \dots, N.$$

Definition 3 (Inneres Produkt)**.** *Das innere Produkt zweier Tensoren gleicher Größe $\mathcal{X}$, $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ ist die Summe der Produkte ihrer Einträge, also*

$$\langle \mathcal{X}, \mathcal{Y} \rangle = \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \dots \sum_{i_N=1}^{I_N} x_{i_1 i_2 \dots i_N}\, y_{i_1 i_2 \dots i_N}.$$

Es folgt direkt, dass $\langle \mathcal{X}, \mathcal{X} \rangle = \| \mathcal{X} \|^2$ gilt.

Satz 2. *Seien $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_{n-1} \times J \times I_{n+1} \times \dots \times I_N}$, $\mathcal{Y} \in \mathbb{R}^{I_1 \times \dots \times I_{n-1} \times K \times I_{n+1} \times \dots \times I_N}$ und $\boldsymbol{A} \in \mathbb{R}^{J \times K}$. Dann gilt*

$$\langle \mathcal{X}, \mathcal{Y} \times_n \boldsymbol{A} \rangle = \left\langle \mathcal{X} \times_n \boldsymbol{A}^T, \mathcal{Y} \right\rangle. \tag{3}$$

Beweis. Als erstes werden die Produkte n-ter Art ausgeführt:

$$\begin{aligned}
(\mathcal{Y} \times_n \boldsymbol{A})_{i_1 \dots i_{n-1} j i_{n+1} \dots i_N} &= \sum_{k=1}^{K} y_{i_1 \dots i_{n-1} k i_{n+1} \dots i_N} a_{jk} \\
\left(\mathcal{X} \times_n \boldsymbol{A}^T\right)_{i_1 \dots i_{n-1} k i_{n+1} \dots i_N} &= \sum_{j=1}^{J} x_{i_1 \dots i_{n-1} j i_{n+1} \dots i_N} a_{jk}.
\end{aligned}$$

Damit folgt

$$\begin{aligned}
\langle \mathcal{X}, \mathcal{Y} \times_n \boldsymbol{A} \rangle &= \sum_{i_1=1}^{I_1} \cdots \sum_{i_{n-1}=1}^{I_{n-1}} \sum_{j=1}^{J} \sum_{i_{n+1}=1}^{I_{n+1}} \cdots \sum_{i_N=1}^{I_N} x_{i_1 \dots i_{n-1} j i_{n+1} \dots i_N} \left(\sum_{k=1}^{K} y_{i_1 \dots i_{n-1} k i_{n+1} \dots i_N} a_{jk} \right) \\
&= \sum_{i_1=1}^{I_1} \cdots \sum_{i_{n-1}=1}^{I_{n-1}} \sum_{k=1}^{K} \sum_{i_{n+1}=1}^{I_{n+1}} \cdots \sum_{i_N=1}^{I_N} y_{i_1 \dots i_{n-1} k i_{n+1} \dots i_N} \left(\sum_{j=1}^{J} x_{i_1 \dots i_{n-1} j i_{n+1} \dots i_N} a_{jk} \right) \\
&= \left\langle \mathcal{X} \times_n \boldsymbol{A}^T, \mathcal{Y} \right\rangle,
\end{aligned}$$

da bei endlichen Summen die Summationsreihenfolge beliebig vertauscht werden darf. □

2.5. Spezielle Vektor- und Matrixprodukte

Definition 4 (Äußeres Produkt). *Das äußere Produkt von N Vektoren $\boldsymbol{a}_{i_1} \in \mathbb{R}^{I_1}, \dots, \boldsymbol{a}_{i_N} \in \mathbb{R}^{I_N}$ ist ein elementweise definierter N-dimensionaler Tensor der Form:*

$$x_{i_1, \dots, i_N} = a_{i_1} \cdot \ldots \cdot a_{i_N} \qquad \textit{mit } i_n = 1, \dots, I_n \quad \forall\, 1 \leq n \leq N.$$

Um die Tucker-Zerlegung in Matrixschreibweise auszudrücken, wird das *Kronecker-Produkt* benötigt.

Definition 5 (Kronecker-Produkt). *Das Kronecker-Produkt zweier Matrizen $\boldsymbol{A} \in \mathbb{R}^{I \times J}$ und $\boldsymbol{B} \in \mathbb{R}^{K \times L}$ wird mit $\boldsymbol{A} \otimes \boldsymbol{B}$ bezeichnet. Das Ergebnis ist eine Matrix der Größe $(IK) \times (JL)$ und ist definiert durch*

$$\begin{aligned}
\boldsymbol{A} \otimes \boldsymbol{B} &= \begin{bmatrix} a_{11}\boldsymbol{B} & a_{12}\boldsymbol{B} & \dots & a_{1J}\boldsymbol{B} \\ a_{21}\boldsymbol{B} & a_{22}\boldsymbol{B} & \dots & a_{2J}\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\boldsymbol{B} & a_{I2}\boldsymbol{B} & \dots & a_{IJ}\boldsymbol{B} \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \boldsymbol{a}_1 \otimes \boldsymbol{b}_2 & \boldsymbol{a}_1 \otimes \boldsymbol{b}_3 & \dots & \boldsymbol{a}_J \otimes \boldsymbol{b}_{L-1} & \boldsymbol{a}_J \otimes \boldsymbol{b}_L \end{bmatrix}.
\end{aligned}$$

Für die Implementierung des ALS sind abgesehen von dem Kronecker-Produkt zwei weitere Matrixprodukte interessant:

Definition 6 (Hadamard-Produkt)**.** *Bei dem Hadamard-Produkt werden zwei Matrizen* $\boldsymbol{A} \in \mathbb{R}^{K \times R}$ *und* $\boldsymbol{B} \in \mathbb{R}^{K \times R}$ *elementweise multipliziert, so dass*

$$\boldsymbol{A} * \boldsymbol{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1R}b_{1R} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2R}b_{2R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K1}b_{K1} & a_{K2}b_{K2} & \dots & a_{KR}b_{KR} \end{bmatrix} = \boldsymbol{C} \in \mathbb{R}^{K \times R}.$$

Definition 7 (Khatri-Rao-Produkt)**.** *Das Khatri-Rao-Produkt wird mit zwei Matrizen* $\boldsymbol{A} \in \mathbb{R}^{K \times R}$ *und* $\boldsymbol{B} \in \mathbb{R}^{J \times R}$ *ausgeführt und liefert eine Matrix* $\boldsymbol{C} \in \mathbb{R}^{(KJ) \times R}$*:*

$$\begin{aligned} \boldsymbol{A} \odot \boldsymbol{B} &= \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1R}b_{1R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{J1} & a_{12}b_{J2} & \dots & a_{1R}b_{JR} \\ a_{21}b_{11} & a_{22}b_{12} & \dots & a_{2R}b_{1R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K1}b_{J1} & a_{K2}b_{J2} & \dots & a_{KR}b_{JR} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \boldsymbol{a}_2 \otimes \boldsymbol{b}_2 & \dots & \boldsymbol{a}_R \otimes \boldsymbol{b}_R \end{bmatrix}. \end{aligned}$$

Bemerkung: Falls $R = 1$ gilt, sind $\boldsymbol{A}$ und $\boldsymbol{B}$ Vektoren und es gilt: $\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a} \odot \boldsymbol{b}$.

Im Folgenden sei $\boldsymbol{A}^{\dagger}$ die Moore-Penrose Pseudoinverse von $\boldsymbol{A}$.
Mithilfe der soeben definierten Matrixprodukte erhält man

Lemma 1.

$$(\boldsymbol{A} \odot \boldsymbol{B}) \odot \boldsymbol{C} = \boldsymbol{A} \odot (\boldsymbol{B} \odot \boldsymbol{C}) \tag{4}$$

$$(\boldsymbol{A}^T \boldsymbol{A}) * (\boldsymbol{B}^T \boldsymbol{B}) = (\boldsymbol{A} \odot \boldsymbol{B})^T (\boldsymbol{A} \odot \boldsymbol{B}) \tag{5}$$

$$\left[(\boldsymbol{A} \odot \boldsymbol{B})^T\right]^{\dagger} = (\boldsymbol{A} \odot \boldsymbol{B})(\boldsymbol{A}^T \boldsymbol{A} * \boldsymbol{B}^T \boldsymbol{B})^{\dagger}. \tag{6}$$

Beweis von (4). Sei $\boldsymbol{A} \odot \boldsymbol{B} = \boldsymbol{G} = (g_{sr}) \in \mathbb{R}^{IJ \times R}$ und $g_{sr} = a_{ir}b_{jr}$ mit $s = J(i-1) + j$.

Dann gilt für $(\boldsymbol{A} \odot \boldsymbol{B}) \odot \boldsymbol{C} = \boldsymbol{G} \odot \boldsymbol{C} = \boldsymbol{E} = (e_{tr}) \in \mathbb{R}^{IJK \times R}$:

$$\begin{aligned} e_{tr} &= g_{sr} c_{kr} \\ &= a_{ir} b_{jr} c_{kr} \end{aligned}$$

für $t = K(s-1) + k = K(J(i-1) + (j-1)) + k$.
Ebenso sei $\boldsymbol{B} \odot \boldsymbol{C} = (g'_{s'r}) = \boldsymbol{G'} \in \mathbb{R}^{JK \times R}$ und $g'_{s'r} = b_{jr} c_{jkr}$ mit $s' = K(j-1) + k$ und so ist $\boldsymbol{A} \odot (\boldsymbol{B} \odot \boldsymbol{C}) = \boldsymbol{A} \odot \boldsymbol{G'} = \boldsymbol{E'} = (e_{t'r}) \in \mathbb{R}^{IJK \times R}$, wobei $e'_{t'r} = a_{ir} g'_{s'r}$ für $t' = KJ(i-1) + s'$. Dies ist aber gerade

$$e'_{t'r} = a_{ir} b_{jr} c_{kr}$$

für $t' = K(s-1) + k = K(J(i-1) + (j-1)) + k$. Es ergibt sich $t = t'$ und so auch $e'_{t'r} = e_{tr}$. Dies ist die Behauptung.

Beweis von (5). Man betrachte $(\boldsymbol{A}^T \boldsymbol{A}) * (\boldsymbol{B}^T \boldsymbol{B}) = \boldsymbol{F} = (f_{r_1 r_2}) \in \mathbb{R}^{R \times R}$ mit

$$\begin{aligned} f_{r_1 r_2} &= \left(\sum_{i=1}^{I} a_{ir_1} a_{ir_2} \right) \left(\sum_{j=1}^{J} b_{jr_1} b_{jr_2} \right) \\ &= \sum_{i=1}^{I} \sum_{j=1}^{J} a_{ir_1} b_{jr_1} a_{ir_2} b_{jr_2} \end{aligned}$$

für $r_1,\ r_2 = 1, \ldots, R$ und außerdem $(\boldsymbol{A} \odot \boldsymbol{B})^T = \boldsymbol{F'} = (f'_{sr}) \in \mathbb{R}^{S \times R}$ für $S = IJ$ und $s = I(i-1) + j$ mit $f'_{sr} = a_{ir} b_{jr}$. Es folgt für $\boldsymbol{F'}^T \boldsymbol{F'} = \boldsymbol{T} = (t_{r_1 r_2}) \in \mathbb{R}^{R \times R}$ mit $r_1,\ r_2 = 1, \ldots, R$:

$$\begin{aligned} t_{r_1 r_2} &= \sum_{s=1}^{S} g_{sr_1} g_{sr_2} \\ &= \sum_{i=1}^{I} \sum_{j=1}^{J} a_{ir_1} b_{jr_1} a_{ir_2} b_{jr_2} \end{aligned}$$

$\Rightarrow$ $\boldsymbol{T} = \boldsymbol{F}$ und somit ergibt sich die Behauptung.

Beweis von (6). Sei nun $\boldsymbol{C} := (\boldsymbol{A} \odot \boldsymbol{B})$.
Dann folgt mithilfe von (5), dass $(\boldsymbol{A}^T \boldsymbol{A}) * (\boldsymbol{B}^T \boldsymbol{B}) = \boldsymbol{C}^T \boldsymbol{C}$ gilt.
Zu zeigen ist: $\left[\boldsymbol{C}^T\right]^\dagger = \boldsymbol{C}(\boldsymbol{C}^T \boldsymbol{C})^\dagger$
Sei weiter $\boldsymbol{C} = \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{V}^T$ eine Singulärwertzerlegung von $\boldsymbol{C}$ mit $\boldsymbol{U}$, $\boldsymbol{V}$ orthogonal und $\boldsymbol{\Sigma} = \operatorname{diag}(\sigma_i)$, wobei σ_i die Singulärwerte von $\boldsymbol{C}$ sind (und $\boldsymbol{\Sigma}$ nicht unbedingt quadratisch ist).

Dann gilt:

$$\begin{aligned}
\left[\boldsymbol{C}^T\right]^\dagger &= \boldsymbol{U}\boldsymbol{\Sigma}^\dagger\boldsymbol{V}^T \\
\boldsymbol{C}(\boldsymbol{C}^T\boldsymbol{C})^\dagger &= \boldsymbol{C}(\boldsymbol{V}\boldsymbol{\Sigma}^T\underbrace{\boldsymbol{U}^T\boldsymbol{U}}_{=\boldsymbol{I}}\boldsymbol{\Sigma}\boldsymbol{V}^T)^\dagger \\
&= \boldsymbol{C}(\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\boldsymbol{V}^T)^\dagger \\
&= \boldsymbol{U}\boldsymbol{\Sigma}\underbrace{\boldsymbol{V}^T\boldsymbol{V}}_{=\boldsymbol{I}}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^\dagger\boldsymbol{V}^T \\
&= \boldsymbol{U}\boldsymbol{\Sigma}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^\dagger\boldsymbol{V}^T
\end{aligned}$$

Weiter folgt:
$\boldsymbol{\Sigma}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^\dagger = \operatorname{diag}(\sigma_i)(\operatorname{diag}(\sigma_i^2))^\dagger = \operatorname{diag}(\sigma_i)\operatorname{diag}(\frac{1}{\sigma_i^2}) = \operatorname{diag}(\frac{1}{\sigma_i}) = \boldsymbol{\Sigma}^\dagger$
Also gilt auch $\left[\boldsymbol{C}^T\right]^\dagger = \boldsymbol{C}(\boldsymbol{C}^T\boldsymbol{C})^\dagger$. □

2.6. Rang eines Tensors

Um den *Rang* eines Tensors zu definieren, ist es hilfreich, zunächst einen Spezialfall zu betrachten:

Definition 8 (Rang-1 Tensoren)**.** *Ein N-dimensionaler Tensor $\mathcal{X} \in \mathbb{R}^{I_1\times\ldots\times I_N}$ ist ein Rang-*1 *Tensor, wenn er als äußeres Produkt von N Vektoren geschrieben werden kann:*

$$\mathcal{X} = \boldsymbol{a}^{(I_1)} \circ \ldots \circ \boldsymbol{a}^{(I_N)}.$$

Daraus erhält man die

Definition 9 (Rang)**.** *Der Rang R eines Tensors $\mathcal{X} \in \mathbb{R}^{I_1\times\ldots\times I_N}$ ist die minimale Anzahl an Rang-*1 *Tensoren, deren Summe $\mathcal{X}$ erzeugt.*

Bemerkung: Es ist zu beachten, dass im Gegensatz zum Rang einer Matrix der Rang R eines N-dimensionalen Tensors durchaus größer als die kleinste Dimension seiner Richtungen sein kann, das heißt $R > \min\{I_n : n = 1, \ldots, N\}$.
Weiterhin ist das Problem der Bestimmung des Rangs eines gegebenen Tensors NP-schwer [7]. Es gibt nähere Erkenntnisse zu typischen (mit Wahrscheinlichkeit größer Null auftretenden) und maximalen Rängen, hierauf wird jedoch im Verlauf der Arbeit nicht näher eingegangen.

3. Tucker-Zerlegung

3.1. Das Modell

Die Tucker-Zerlegung wurde von Tucker [12] im Zusammenhang mit der *Faktor-Analysis* entwickelt. Dabei wird ein Tensor in einen *Kerntensor* zerlegt, der mit jeweils einer Matrix pro Modus multipliziert wird. Für einen dreidimensionalen Tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ erhält man folgende Zerlegung:

$$\mathcal{X} = \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} = \sum_{m=1}^{M} \sum_{p=1}^{P} \sum_{q=1}^{Q} g_{mpq}\, \boldsymbol{a}_m \circ \boldsymbol{b}_p \circ \boldsymbol{c}_q. \tag{7}$$

Dabei heißt der Tensor $\mathcal{G} \in \mathbb{R}^{M \times P \times Q}$ Kerntensor und an seinen Einträgen kann man ablesen, wie stark die Zusammenhänge zwischen den einzelnen Komponenten sind. Die meistens orthogonalen Matrizen $\boldsymbol{A} \in \mathbb{R}^{I \times M}$, $\boldsymbol{B} \in \mathbb{R}^{J \times P}$ und $\boldsymbol{C} \in \mathbb{R}^{K \times Q}$ heißen *Faktormatrizen* oder *Koeffizientenmatrizen.*

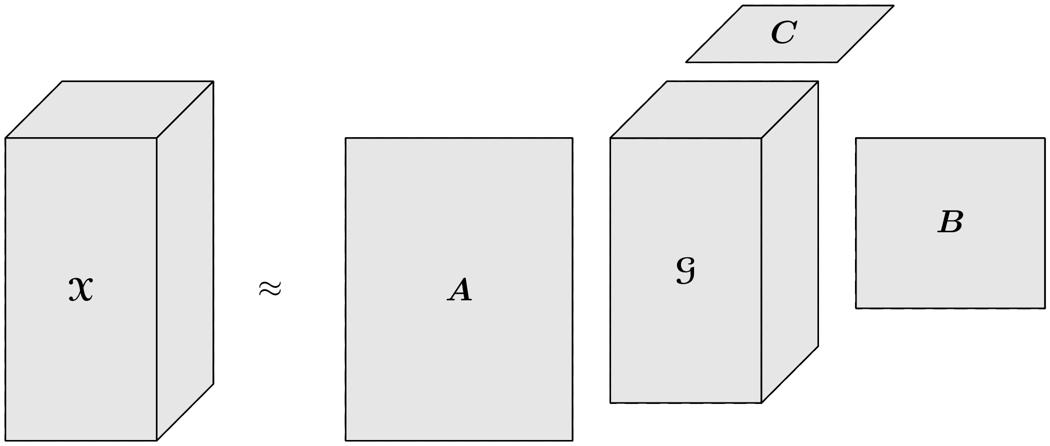


Abbildung 3: Tucker-Zerlegung eines Tensors dritter Ordnung.

Elementweise gilt für die Tucker-Zerlegung in (7)

$$x_{ijk} = \sum_{m=1}^{M} \sum_{p=1}^{P} \sum_{q=1}^{Q} g_{mpq}\, a_{im}\, b_{jp}\, c_{kq} \tag{8}$$

für $i = 1, \ldots, I,\ j = 1, \ldots, J,\ k = 1, \ldots, K$. Hierbei sind M, P und Q die Anzahl der Komponenten bzw. Zeilen in den entsprechenden Faktormatrizen $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$. Sind M, P und Q kleiner als I, J und K, dann kann der Kerntensor $\mathcal{G}$ als komprimierte Version des Tensors $\mathcal{X}$ betrachtet werden.

Neben der Darstellung (7) für den Tensor $\mathcal{X}$ gibt es noch eine weitere äquivalente Darstellung:

Satz 3. *Für die drei Transformationen des Tensors $\mathcal{X}$ in Matrizen gilt:*

$$\begin{aligned} \boldsymbol{X}_{(1)} &= \boldsymbol{A}\boldsymbol{G}_{(1)}(\boldsymbol{C} \otimes \boldsymbol{B})^T, \\ \boldsymbol{X}_{(2)} &= \boldsymbol{B}\boldsymbol{G}_{(2)}(\boldsymbol{C} \otimes \boldsymbol{A})^T, \\ \boldsymbol{X}_{(3)} &= \boldsymbol{C}\boldsymbol{G}_{(3)}(\boldsymbol{B} \otimes \boldsymbol{A})^T. \end{aligned} \tag{9}$$

Beweis. Zuerst müssen die Tensoren $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ und $\mathcal{G} \in \mathbb{R}^{M\times P\times Q}$ in Matrizen $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I\times JK}$ und $\boldsymbol{G}_{(1)} \in \mathbb{R}^{M\times PQ}$ 1-ter Art transformiert werden. Dabei wird ein Tensorelement x_{ijk} auf ein Matrixelement x_{is} abgebildet mit $s = (k-1)J + j$ und ein Tensorelement g_{mpq} auf das Matrixelement g_{mr} mit $r = (q-1)Q + p$.
Als nächstes werden die Produkte $\boldsymbol{A}\boldsymbol{G}_{(1)}$ und $(\boldsymbol{C} \otimes \boldsymbol{B})^T = (\boldsymbol{C}^T \otimes \boldsymbol{B}^T)$ berechnet:

$$\begin{aligned} (\boldsymbol{A}\boldsymbol{G}_{(1)})_{ir} &= \sum_{m=1}^{M} a_{im}\, g_{mr} \\ (\boldsymbol{C}^T \otimes \boldsymbol{B}^T)_{rs} &= c_{kq}\, b_{jp}. \end{aligned}$$

Damit folgt dann

$$\begin{aligned} x_{is} &= (\boldsymbol{A}\boldsymbol{G}_{(1)}(\boldsymbol{C}^T \otimes \boldsymbol{B}^T))_{is} \\ &= \sum_{r=1}^{R} \left(\sum_{m=1}^{M} a_{im}\, g_{mr} \right) (c_{kq}\, b_{jp}) \\ &= \sum_{q=1}^{Q} \sum_{p=1}^{P} \sum_{m=1}^{M} a_{im}\, g_{mpq}\, c_{kq}\, b_{jp} \\ &= \sum_{m=1}^{M} \sum_{p=1}^{P} \sum_{q=1}^{Q} g_{mpq}\, a_{im}\, b_{jp}\, c_{kq} \\ &= x_{ijk}. \end{aligned}$$

Die Darstellungen für $\boldsymbol{X}_{(2)}$ und $\boldsymbol{X}_{(3)}$ lassen sich analog zeigen. □

Natürlich kann die Darstellung durch die Tucker-Zerlegung auch für beliebige N-dimensionale Tensoren verallgemeinert werden:

$$\mathcal{X} = \mathcal{G} \times_1 \boldsymbol{A}^{(1)} \times_2 \boldsymbol{A}^{(2)} \times_3 \ldots \times_N \boldsymbol{A}^{(N)}$$

oder elementweise

$$x_{i_1 i_2 \ldots i_N} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_N=1}^{R_N} g_{r_1 r_2 \ldots r_N}\, a^{(1)}_{i_1 r_1}\, a^{(2)}_{i_2 r_2} \cdots a^{(N)}_{i_N r_N}$$

für $i_n = 1, \dots, I_n, \quad n = 1, \dots, N$.
Für diese N-dimensionalen Tensoren erhält man die Darstellung

$$\boldsymbol{X}_{(n)} = \boldsymbol{A}^{(n)} \boldsymbol{G}_{(n)} \left(\boldsymbol{A}^{(N)} \otimes \dots \otimes \boldsymbol{A}^{(n+1)} \otimes \boldsymbol{A}^{(n-1)} \otimes \dots \otimes \boldsymbol{A}^{(1)} \right)^T$$

in Matrixschreibweise.

3.2. Eigenschaften der Tucker-Zerlegung

Im Folgenden seien die Spalten von $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$ jeweils paarweise orthonormal, das heißt es gilt $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}$, $\boldsymbol{B}^T\boldsymbol{B} = \boldsymbol{I}$ und $\boldsymbol{C}^T\boldsymbol{C} = \boldsymbol{I}$. Damit ergibt sich für den Kerntensor die nützliche Eigenschaft:

Satz 4. *Die Norm des Tensors* $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ *ist gleich der Norm des Kerntensors* $\mathcal{G} \in \mathbb{R}^{M \times P \times Q}$*:*

$$\| \mathcal{X} \| = \| \mathcal{G} \| . \tag{10}$$

Beweis.

$$\begin{aligned}
\sum_{m=1}^{M} \sum_{p=1}^{P} \sum_{q=1}^{Q} g_{mpq}^2 &= \| \mathcal{G} \|^2 = \| \boldsymbol{G}_{(1)} \|^2 \\
&\overset{(*)}{=} \| \boldsymbol{A}\boldsymbol{G}_{(1)} \|^2 =: \| \boldsymbol{V}_{(1)} \|^2 = \| \boldsymbol{V}_{(2)} \|^2 \\
&\overset{(*)}{=} \| \boldsymbol{B}\boldsymbol{V}_{(2)} \|^2 =: \| \boldsymbol{W}_{(2)} \|^2 = \| \boldsymbol{W}_{(3)} \|^2 \\
&\overset{(*)}{=} \| \boldsymbol{C}\boldsymbol{W}_{(3)} \|^2 = \| \boldsymbol{X}_{(3)} \|^2 = \| \mathcal{X} \|^2 \\
&= \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} x_{ijk}^2 ,
\end{aligned}$$

wobei die Gleichungen $(*)$ jeweils aus der Invarianz der Frobeniusnorm unter Orthogonaltransformationen folgen. □

Möchte man die Norm des Tensors $\mathcal{X}$ berechnen, so ist diese Eigenschaft von Vorteil, da der Kerntensor $\mathcal{G}$ meistens deutlich kleiner ist, als der eigentliche Tensor $\mathcal{X}$.

Satz 5. *Für den Kerntensor gilt*

$$\mathcal{G} = \mathcal{X} \times_1 \boldsymbol{A}^T \times_2 \boldsymbol{B}^T \times_3 \boldsymbol{C}^T . \tag{11}$$

Beweis. Nach Satz 1 darf die Reihenfolge der Produkte n-ter Art vertauscht werden und für gleiches n dürfen diese zusammengefasst werden. Damit folgt

$$\begin{aligned} & \mathcal{X} &=& \ \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \\ \Leftrightarrow & \mathcal{X} \times_3 \boldsymbol{C}^T &=& \ \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \times_3 \boldsymbol{C}^T = \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 (\underbrace{\boldsymbol{C}^T\boldsymbol{C}}_{\boldsymbol{I}}) \\ \Leftrightarrow & \mathcal{X} \times_3 \boldsymbol{C}^T \times_2 \boldsymbol{B}^T &=& \ \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_2 \boldsymbol{B}^T = \mathcal{G} \times_1 \boldsymbol{A} \times_2 (\underbrace{\boldsymbol{B}^T\boldsymbol{B}}_{\boldsymbol{I}}) \\ \Leftrightarrow & \mathcal{X} \times_3 \boldsymbol{C}^T \times_2 \boldsymbol{B}^T \times_1 \boldsymbol{A}^T &=& \ \mathcal{G} \times_1 \boldsymbol{A} \times_1 \boldsymbol{A}^T = \mathcal{G} \times_1 (\underbrace{\boldsymbol{A}^T\boldsymbol{A}}_{\boldsymbol{I}}) \\ \Leftrightarrow & \mathcal{X} \times_1 \boldsymbol{A}^T \times_2 \boldsymbol{B}^T \times_3 \boldsymbol{C}^T &=& \ \mathcal{G}. \end{aligned}$$

□

Korollar 1. *Der Kerntensor $\mathcal{G}$ kann berechnet werden durch*

$$\begin{aligned} \boldsymbol{A}^T\boldsymbol{X}_{(1)} &=: \boldsymbol{V}_{(1)} \\ \boldsymbol{B}^T\boldsymbol{V}_{(2)} &=: \boldsymbol{W}_{(2)} \\ \boldsymbol{C}^T\boldsymbol{W}_{(3)} &= \ \boldsymbol{G}_{(3)}. \end{aligned} \tag{12}$$

Beweis. Folgt direkt mit der Eigenschaft, dass ein Produkt n-ter Art auch als Matrixmultiplikation dargestellt werden kann. □

Die Tucker-Zerlegung ist nicht eindeutig. Man kann den Kerntensor $\mathcal{G}$ mit Hilfe von nicht-singulären Matrizen $\boldsymbol{U} \in \mathbb{R}^{M\times M}$, $\boldsymbol{V} \in \mathbb{R}^{P\times P}$ und $\boldsymbol{W} \in \mathbb{R}^{Q\times Q}$ modifizieren, ohne das Ergebnis der Tucker-Zerlegung zu verändern. Dabei muss nur die jeweils inverse Modifikation auf die Faktormatrizen $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$ angewandt werden.

Satz 6. *Sei $\mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C}$ die Tucker-Zerlegung des Tensors $\mathcal{X}$ wie in (7). Dann gilt*

$$\mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} = (\mathcal{G} \times_1 \boldsymbol{U} \times_2 \boldsymbol{V} \times_3 \boldsymbol{W}) \times_1 (\boldsymbol{A}\boldsymbol{U}^{-1}) \times_2 (\boldsymbol{B}\boldsymbol{V}^{-1}) \times_3 (\boldsymbol{C}\boldsymbol{W}^{-1})$$

für $\boldsymbol{U} \in \mathbb{R}^{M\times M}$, $\boldsymbol{V} \in \mathbb{R}^{P\times P}$ und $\boldsymbol{W} \in \mathbb{R}^{Q\times Q}$.

Beweis. Nach Satz 1 darf man die Reihenfolge der Produkte n-ter Art beliebig vertauschen und für ein festes n zusammenfassen:

$$\begin{aligned} & (\mathcal{G} \times_1 \boldsymbol{U} \times_2 \boldsymbol{V} \times_3 \boldsymbol{W}) \times_1 (\boldsymbol{A}\boldsymbol{U}^{-1}) \times_2 (\boldsymbol{B}\boldsymbol{V}^{-1}) \times_3 (\boldsymbol{C}\boldsymbol{W}^{-1}) \\ = \ & \mathcal{G} \times_1 \boldsymbol{U} \times_1 \boldsymbol{A}\boldsymbol{U}^{-1} \times_2 \boldsymbol{V} \times_2 \boldsymbol{B}\boldsymbol{V}^{-1} \times_3 \boldsymbol{W} \times_3 \boldsymbol{C}\boldsymbol{W}^{-1} \\ = \ & \mathcal{G} \times_1 (\boldsymbol{A}\boldsymbol{U}^{-1}\boldsymbol{U}) \times_2 (\boldsymbol{B}\boldsymbol{V}^{-1}\boldsymbol{V}) \times_3 (\boldsymbol{C}\boldsymbol{W}^{-1}\boldsymbol{W}) \\ = \ & \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C}. \end{aligned}$$

□

Dieser Satz ermöglicht es, die Transformationen so auszuwählen, dass die Struktur des Kerntensors $\mathcal{G}$ vereinfacht wird. Beispielsweise kann $\mathcal{G}$ so transformiert werden, dass möglichst viele Einträge gleich Null oder sehr klein werden. Dadurch werden die Zusammenhänge zwischen den einzelnen Komponenten verkleinert.

3.3. Der Algorithmus

Tucker [12] stellte drei Methoden vor, um die Zerlegung zu berechnen. Die erste Methode war ursprünglich nur für kleine Tensoren geeignet, da sie mehr Speicherplatz benötigt als Methode 2 und 3. Heutzutage reicht aber auch die Kapazität des Computers für größere Tensoren, weswegen nur Methode 1 vorgestellt wird. Sie versucht die Komponenten zu finden, welche die Variation in einem Modus n am besten erfassen, unabhängig von den anderen Modi. Tucker formulierte Methode 1 wie folgt:

- Berechne die Produktmatrizen $\boldsymbol{M}$, $\boldsymbol{P}$ und $\boldsymbol{Q}$ mit Hilfe des gegebenen Tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$:

$$\begin{aligned} \boldsymbol{M} &= \boldsymbol{X}_{(1)}\, \boldsymbol{X}_{(1)}^T \ \in \mathbb{R}^{I \times I} \\ \boldsymbol{P} &= \boldsymbol{X}_{(2)}\, \boldsymbol{X}_{(2)}^T \ \in \mathbb{R}^{J \times J} \\ \boldsymbol{Q} &= \boldsymbol{X}_{(3)}\, \boldsymbol{X}_{(3)}^T \ \in \mathbb{R}^{K \times K}. \end{aligned}$$

- Berechne die Eigenwerte und Eigenvektoren dieser Produktmatrizen.
- Verwende jeweils die M, P und Q Eigenwerte ungleich Null und stelle die Koeffizientenmatrizen $\boldsymbol{A} \in \mathbb{R}^{I \times M}$, $\boldsymbol{B} \in \mathbb{R}^{J \times P}$ und $\boldsymbol{C} \in \mathbb{R}^{K \times Q}$ auf, die die entsprechenden normierten Eigenvektoren als Spalten enthalten.
- Berechne den Kerntensor $\mathcal{G}$ mit einer der umgestellten Formeln aus (9), zum Beispiel

$$\boldsymbol{G}_{(1)} = \boldsymbol{A}^T \boldsymbol{X}_{(1)} (\boldsymbol{C} \otimes \boldsymbol{B}),$$

  oder mit (12)

$$\begin{aligned} \boldsymbol{A}^T \boldsymbol{X}_{(1)} &= \boldsymbol{V}_{(1)} \\ \boldsymbol{B}^T \boldsymbol{V}_{(2)} &= \boldsymbol{W}_{(2)} \\ \boldsymbol{C}^T \boldsymbol{W}_{(3)} &= \boldsymbol{G}_{(3)}. \end{aligned}$$

Letzteres hat eine kürzere Berechnungszeit, da das Kronecker-Produkt nicht benötigt wird und somit große Matrizen als Zwischenergebnisse vermieden werden. Stattdessen müssen Matrizen umsortiert werden, was sich schneller realisieren lässt, da dabei keine neuen Berechnungen durchgeführt werden müssen. Näheres dazu findet sich in [6] und Kapitel 3.7.

Heutzutage ist diese Methode auch als *Singulärwert-Zerlegung höherer Ordnung* (HOSVD) bekannt.

Algorithmus 1 zeigt die Tucker-Zerlegung in Kurzform.

Algorithmus 1: Tucker-Zerlegung($\mathcal{X}$)

```
$A \leftarrow M$ führende Eigenvektoren von $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$
$B \leftarrow P$ führende Eigenvektoren von $\boldsymbol{X}_{(2)}\boldsymbol{X}_{(2)}^T$
$C \leftarrow Q$ führende Eigenvektoren von $\boldsymbol{X}_{(3)}\boldsymbol{X}_{(3)}^T$
$\mathcal{G} \leftarrow \mathcal{X} \times_1 \boldsymbol{A}^T \times_2 \boldsymbol{B}^T \times_3 \boldsymbol{C}^T$
return $\mathcal{G}$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$
```

Betrachte die Singulärwertzerlegung $\boldsymbol{X}_{(1)} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ von $\boldsymbol{X}_{(1)}$ mit den Singulärwerten $\sigma_1^{(1)} \geq \sigma_2^{(1)} \geq \ldots \geq \sigma_M^{(1)} > \sigma_{M+1}^{(1)} = \ldots = \sigma_I^{(1)} = 0$,

$$\boldsymbol{\Sigma} := \left[\begin{array}{ccc|ccc} \sigma_1 & & & & \vdots & \\ & \ddots & & \cdots & 0 & \cdots \\ & & \sigma_M & & \vdots & \\ \hline & \vdots & & & \vdots & \\ \cdots & 0 & \cdots & \cdots & 0 & \cdots \\ & \vdots & & & \vdots & \end{array}\right] \in \mathbb{R}^{I \times JK}$$

und $\boldsymbol{U} \in \mathbb{R}^{I \times I}$, $\boldsymbol{V} \in \mathbb{R}^{JK \times JK}$ orthogonal. Dann sind die Eigenwerte von $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$ genau die Quadrate der Singulärwerte $\sigma_i^{(1)}$ und die Eigenvektoren entsprechen den Spalten der Matrix $\boldsymbol{U}$. Mit Hilfe der Singulärwerte kann man die Norm von $\mathcal{X}$ bestimmen:

Satz 7. *Seien $\boldsymbol{X}_{(1)}$, $\boldsymbol{X}_{(2)}$ und $\boldsymbol{X}_{(3)}$ die drei Transformationen eines Tensors $\mathcal{X}$ in Matrizen und $\sigma_i^{(1)}$, $\sigma_j^{(2)}$, $\sigma_k^{(3)}$ die dazugehörigen Singulärwerte. Dann gilt*

$$\begin{aligned} \| \mathcal{X} \|^2 &= \sum_{i=1}^{I} \left(\sigma_i^{(1)}\right)^2 = \sum_{i=1}^{M} \left(\sigma_i^{(1)}\right)^2 \\ &= \sum_{j=1}^{J} \left(\sigma_j^{(2)}\right)^2 = \sum_{j=1}^{P} \left(\sigma_j^{(2)}\right)^2 \\ &= \sum_{k=1}^{K} \left(\sigma_k^{(3)}\right)^2 = \sum_{k=1}^{Q} \left(\sigma_k^{(3)}\right)^2. \end{aligned}$$

Beweis.

$$\begin{aligned}\| \mathcal{X} \|^2 &= \| \boldsymbol{X}_{(1)} \|^2 &&= \operatorname{tr}(\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T)\\ &= \operatorname{tr}((\boldsymbol{U\Sigma V}^T)(\boldsymbol{U\Sigma V}^T)^T) &&= \operatorname{tr}(\boldsymbol{U\Sigma V}^T\boldsymbol{V\Sigma}^T\boldsymbol{U}^T)\\ &= \operatorname{tr}(\boldsymbol{U\Sigma\Sigma}^T\boldsymbol{U}^T) &&= \operatorname{tr}(\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{U\Sigma})\\ &= \operatorname{tr}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}) &&= \operatorname{tr}(\boldsymbol{\Sigma\Sigma}^T)\\ &= \sum_{i=1}^{I}\left(\sigma_i^{(1)}\right)^2\end{aligned}$$

Da $\sigma_{M+1}^{(1)} = \sigma_{M+2}^{(1)} = \ldots = \sigma_I^{(1)} = 0$ gilt, reicht es aus, die letzte Summe nur bis M laufen zu lassen.

Ebenso folgt der Beweis für $\| \mathcal{X} \|^2 = \sum_{j=1}^{J}\left(\sigma_j^{(2)}\right)^2$ und $\| \mathcal{X} \|^2 = \sum_{k=1}^{K}\left(\sigma_k^{(3)}\right)^2$. □

Definition 10 (n-Rang)**.** *Sei* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ *ein Tensor* N*-ter Ordnung. Dann ist der* n*-Rang von* $\mathcal{X}$ *der Spaltenrang von* $\boldsymbol{X}_{(n)}$ *und wird mit* $\operatorname{rank}_n(\mathcal{X})$ *bezeichnet.*

Der n-Rang ist also die Dimension des Vektorraums, der durch die Fasern n-ter Art aufgespannt wird. Setzt man $R_n = \operatorname{rank}_n(\mathcal{X})$ für $n = 1, \ldots, N$, dann kann man $\mathcal{X}$ als Rang $(R_1, R_2, \ldots, R_N)$ - Tensor bezeichnen. Dies darf aber nicht mit dem Rang eines Tensors in Kapitel 2.6 verwechselt werden.

Mit oben genanntem Algorithmus ist es einfach, eine exakte Tucker-Zerlegung eines Tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ vom Rang (M, P, Q) mit $M = \operatorname{rank}_1(\mathcal{X})$, $P = \operatorname{rank}_2(\mathcal{X})$ und $Q = \operatorname{rank}_3(\mathcal{X})$ zu berechnen. Berechnet man hingegen eine Tucker-Zerlegung mit $R_n < \operatorname{rank}_n(\mathcal{X})$ für mindestens ein n, dann ist diese nicht mehr exakt. Auf diese Art und Weise erhält man die *abgeschnittene Tucker-Zerlegung*:

3.4. Die abgeschnittene Tucker-Zerlegung

Möchte man die Datenmenge eines Tensors $\mathcal{X}$ komprimieren, so kann man ihn mit der abgeschnittenen Tucker-Zerlegung approximieren. Hierbei verwendet man nur die größten M, P und Q Eigenwerte der Produktmatrizen und erhält mit den zugehörigen Eigenvektoren kleinere Koeffizientenmatrizen $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$. Dadurch verkleinert sich natürlich auch die Größe des Kerntensors $\mathcal{G}$.

Die abgeschnittene Tucker-Zerlegung ist nicht die beste Rang (M, P, Q) - Approximation an den Tensor $\mathcal{X}$, aber die Matrizen $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$ können gut als Startwerte für einen iterativen Algorithmus benutzt werden, wie zum Beispiel der *Higher-Order Orthogonal Iteration* (HOOI) in [11].

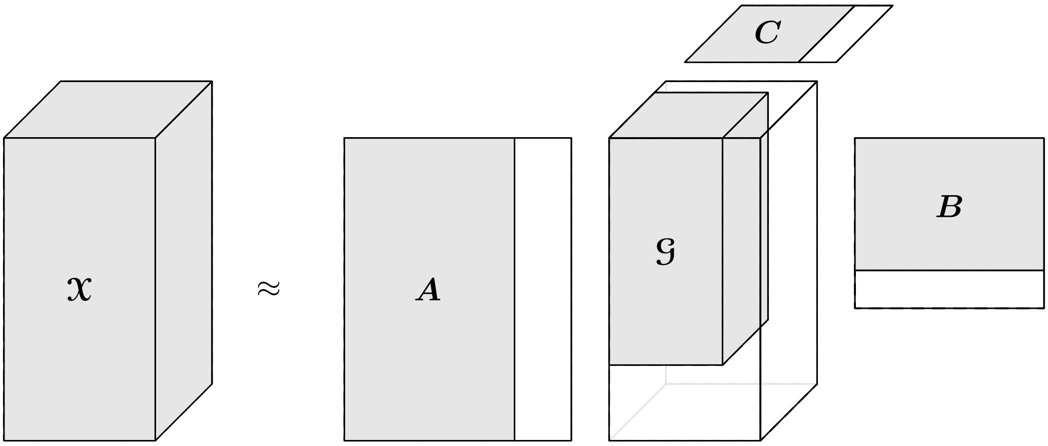


Abbildung 4: abgeschnittene Tucker-Zerlegung eines Tensors dritter Ordnung.

Beispiel 3. *Für den Tensor $\mathcal{X}$ aus Beispiel 1 erhält man folgende abgeschnittene Tucker-Zerlegung vom Rang* $(2, 2, 1)$*:*

$$\boldsymbol{A} = \begin{bmatrix} 0.40791 & 0.19876 \\ 0.64690 & -0.75168 \\ 0.64430 & 0.62887 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} 0.55690 & 0.53901 \\ 0.31758 & -0.58419 \\ 0.35068 & 0.45487 \\ 0.68266 & -0.40160 \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} -0.64932 \\ -0.76052 \end{bmatrix} \quad \boldsymbol{G} = \begin{bmatrix} 24.94869 & -0.09724 \\ -0.08623 & -5.14788 \end{bmatrix}.$$

Damit erhält man den approximierten Tensor $\widetilde{\mathcal{X}}$ *mit*

$$\widetilde{\boldsymbol{X}}_1 = \begin{bmatrix} -4.05819 & 1.69894 & 2.63512 & 4.24146 \\ 4.48035 & 4.75870 & 2.53590 & 8.11790 \\ 6.98720 & 2.07414 & 4.64723 & 6.28874 \end{bmatrix}$$

$$\widetilde{\boldsymbol{X}}_2 = \begin{bmatrix} -4.75318 & 1.98989 & 3.08640 & 4.96784 \\ 5.24763 & 5.57365 & 2.97019 & 9.50814 \\ 8.18380 & 2.42934 & 5.44309 & 7.36573 \end{bmatrix}$$

und es gilt für den Fehler

$$\begin{aligned} \| \mathcal{X} - \widetilde{\mathcal{X}} \| &= 6.40685 \\ &= 0.24390 \, \| \mathcal{X} \| . \end{aligned}$$

Führt man danach noch zwei Schritte des iterativen Algorithmus 2 durch, so erhält man

Algorithmus 2: HOOI($\mathcal{X}$, $R_1, \dots, R_N$)

```
initialisiere $\boldsymbol{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ für $n = 1, \dots, N$ mit der abgeschnittenen Tucker-Zerlegung
repeat
  for $n = 1, \dots, N$ do
    $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \boldsymbol{A}^{(1)T} \dots \times_{n-1} \boldsymbol{A}^{(n-1)T} \times_{n+1} \boldsymbol{A}^{(n+1)T} \dots \times_N \boldsymbol{A}^{(N)T}$
    $\boldsymbol{A}^{(n)} \leftarrow R_n$ führende Linkssingulärvektoren von $\boldsymbol{Y}_{(n)}$
  end for
until Fehlerschranke oder maximale Anzahl an Iterationen erreicht
$\mathcal{G} \leftarrow \mathcal{X} \times_1 \boldsymbol{A}^{(1)T} \times_2 \boldsymbol{A}^{(2)T} \dots \times_N \boldsymbol{A}^{(N)T}$
return $\mathcal{G}$, $\boldsymbol{A}^{(1)}$, $\boldsymbol{A}^{(2)}$, ..., $\boldsymbol{A}^{(N)}$
```

das bessere Ergebnis

$$\boldsymbol{A} = \begin{bmatrix} 0.40685 & 0.24347 \\ 0.64383 & -0.76017 \\ 0.64804 & 0.60238 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} -0.56193 & 0.36352 \\ -0.31729 & -0.80977 \\ -0.35146 & 0.43604 \\ -0.67826 & -0.14831 \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} -0.65683 \\ -0.75404 \end{bmatrix} \quad \boldsymbol{G} = \begin{bmatrix} 24.95146 & -0.00016 \\ -0.00347 & -5.51672 \end{bmatrix}$$

mit

$$\widetilde{\boldsymbol{X}}_1 = \begin{bmatrix} 4.06802 & 1.40105 & 2.72854 & 4.39198 \\ 4.92724 & 5.57734 & 2.50709 & 7.56400 \\ 6.76255 & 1.60217 & 4.68528 & 6.88066 \end{bmatrix}$$

$$\widetilde{\boldsymbol{X}}_2 = \begin{bmatrix} 4.67009 & 1.60841 & 3.13236 & 5.04199 \\ 5.65647 & 6.40278 & 2.87814 & 8.68346 \\ 7.76340 & 1.83928 & 5.37870 & 7.89898 \end{bmatrix}$$

und dem Fehler

$$\begin{aligned} \| \mathcal{X} - \widetilde{\mathcal{X}} \| &= 6.08177 \\ &= 0.23153 \, \| \mathcal{X} \| . \end{aligned}$$

3.5. Fehlerabschätzungen

Für die folgende Fehlerabschätzung benötigen wir ein Lemma:

Lemma 2. *Seien* $\sigma_i^{(1)}$, $\sigma_j^{(2)}, \sigma_k^{(3)}$ *die Singulärwerte der Matrizen* $\boldsymbol{X}_{(1)}$, $\boldsymbol{X}_{(2)}$, $\boldsymbol{X}_{(3)}$. *Dann gilt*

$$\begin{aligned} \left(\sigma_i^{(1)}\right)^2 &= \sum_{j=1}^{J}\sum_{k=1}^{K} g_{ijk}^2 \\ \left(\sigma_j^{(2)}\right)^2 &= \sum_{i=1}^{I}\sum_{k=1}^{K} g_{ijk}^2 \\ \left(\sigma_k^{(3)}\right)^2 &= \sum_{i=1}^{I}\sum_{j=1}^{J} g_{ijk}^2. \end{aligned} \tag{13}$$

Beweis. Sei $\boldsymbol{X}_{(1)} \approx \widetilde{\boldsymbol{X}}_{(1)} = \boldsymbol{U\Sigma V}^T$ die abgeschnittene Singulärwertzerlegung von $\boldsymbol{X}_{(1)}$ mit $\boldsymbol{U} \in \mathbb{R}^{I\times M}$, $\boldsymbol{V} \in \mathbb{R}^{JK\times M}$ und $\boldsymbol{\Sigma} = \operatorname{diag}(\sigma_1^{(1)}, \sigma_2^{(1)}, \ldots, \sigma_M^{(1)}) \in \mathbb{R}^{M\times M}$. Dann folgt

$$\begin{array}{rcccl} \| \mathcal{G} \|^2 &=& \| \widetilde{\mathcal{X}} \|^2 &=& \| \widetilde{\boldsymbol{X}}_{(1)} \|^2 \\ &=& \| \boldsymbol{U\Sigma V}^T \|^2 &=& \operatorname{tr}(\boldsymbol{U\Sigma V}^T\boldsymbol{V\Sigma}^T\boldsymbol{U}^T) \\ &=& \operatorname{tr}(\boldsymbol{U\Sigma\Sigma}^T\boldsymbol{U}^T) &=& \operatorname{tr}(\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{U\Sigma}) \\ &=& \operatorname{tr}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}) &=& \operatorname{tr}(\boldsymbol{\Sigma\Sigma}^T) \\ &=& \| \boldsymbol{\Sigma} \|^2 &=& \sum_{i=1}^{M} \left(\sigma_i^{(1)}\right)^2. \end{array}$$

Also gilt

$$\sum_{i=1}^{M}\sum_{j=1}^{J}\sum_{k=1}^{K} g_{ijk}^2 = \sum_{i=1}^{M} \left(\sigma_i^{(1)}\right)^2.$$

Da dies für alle $M \in \{1, \ldots, I\}$ gilt, müssen die einzelnen Summanden schon gleich sein:

$$\left(\sigma_i^{(1)}\right)^2 = \sum_{j=1}^{J}\sum_{k=1}^{K} g_{ijk}^2.$$

Gleiches gilt für $\left(\sigma_i^{(2)}\right)^2$ und $\left(\sigma_i^{(3)}\right)^2$. □

Damit erhält man für die Fehlerabschätzung nach oben:

Satz 8. *Sei* $\mathcal{X} \approx \widetilde{\mathcal{X}} = \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C}$ *die abgeschnittene Tucker-Zerlegung des Tensors* $\mathcal{X}$*. Dann gilt für den Fehler:*

$$\left\| \mathcal{X} - \widetilde{\mathcal{X}} \right\|^2 \leq \sum_{i=M+1}^{I} \left(\sigma_i^{(1)}\right)^2 + \sum_{j=P+1}^{J} \left(\sigma_j^{(2)}\right)^2 + \sum_{k=Q+1}^{K} \left(\sigma_k^{(3)}\right)^2. \tag{14}$$

Beweis.

$$\begin{aligned}
\| \mathcal{X} - \widetilde{\mathcal{X}} \|^2 &= \| \mathcal{X} \|^2 - 2 \langle \mathcal{X}, \widetilde{\mathcal{X}} \rangle + \| \widetilde{\mathcal{X}} \|^2 \\
&= \| \mathcal{X} \|^2 - 2 \langle \mathcal{X}, \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \rangle + \| \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \|^2 \\
&\overset{(3)}{=} \| \mathcal{X} \|^2 - 2 \langle \mathcal{X} \times_1 \boldsymbol{A}^T \times_2 \boldsymbol{B}^T \times_3 \boldsymbol{C}^T, \mathcal{G} \rangle + \| \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \|^2 \\
&\overset{(10)}{=} \| \mathcal{X} \|^2 - 2 \langle \mathcal{G}, \mathcal{G} \rangle + \| \mathcal{G} \|^2 \\
&= \| \mathcal{X} \|^2 - \| \mathcal{G} \|^2 \\
&= \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} g_{ijk}^2 - \sum_{i=1}^{M} \sum_{j=1}^{P} \sum_{k=1}^{Q} g_{ijk}^2 \\
&\leq \sum_{i=M+1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} g_{ijk}^2 + \sum_{i=1}^{I} \sum_{j=P+1}^{J} \sum_{k=1}^{K} g_{ijk}^2 + \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=Q+1}^{K} g_{ijk}^2 \\
&\overset{(13)}{=} \sum_{i=M+1}^{I} \left(\sigma_i^{(1)}\right)^2 + \sum_{j=P+1}^{J} \left(\sigma_j^{(2)}\right)^2 + \sum_{k=Q+1}^{K} \left(\sigma_k^{(3)}\right)^2.
\end{aligned}$$

Vergleiche auch mit [11] und [8]. □

Ähnlich erhält man eine Fehlerabschätzung nach unten:

Satz 9. *Sei* $\mathcal{X} \approx \widetilde{\mathcal{X}} = \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C}$ *die abgeschnittene Tucker-Zerlegung des Tensors* $\mathcal{X}$*. Dann gilt für den Fehler:*

$$\begin{aligned}
\left\| \mathcal{X} - \widetilde{\mathcal{X}} \right\|^2 &\geq \sum_{i=M+1}^{I} \left(\sigma_i^{(1)}\right)^2 \\
\left\| \mathcal{X} - \widetilde{\mathcal{X}} \right\|^2 &\geq \sum_{j=P+1}^{J} \left(\sigma_j^{(2)}\right)^2 \\
\left\| \mathcal{X} - \widetilde{\mathcal{X}} \right\|^2 &\geq \sum_{k=Q+1}^{K} \left(\sigma_k^{(3)}\right)^2.
\end{aligned}$$

Beweis. Die Aussage lässt sich ähnlich beweisen wie (14).

$$\begin{aligned}
\| \mathcal{X} - \widetilde{\mathcal{X}} \|^2 &= \| \mathcal{X} \|^2 - 2\, \langle \mathcal{X}, \widetilde{\mathcal{X}} \rangle + \| \widetilde{\mathcal{X}} \|^2 \\
&= \| \mathcal{X} \|^2 - 2\, \langle \mathcal{X}, \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \rangle + \| \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \|^2 \\
&\overset{(3)}{=} \| \mathcal{X} \|^2 - 2\, \langle \mathcal{X} \times_1 \boldsymbol{A}^T \times_2 \boldsymbol{B}^T \times_3 \boldsymbol{C}^T, \mathcal{G} \rangle + \| \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \|^2 \\
&\overset{(10)}{=} \| \mathcal{X} \|^2 - 2\, \langle \mathcal{G}, \mathcal{G} \rangle + \| \mathcal{G} \|^2 \\
&= \| \mathcal{X} \|^2 - \| \mathcal{G} \|^2 \\
&= \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} g_{ijk}^2 - \sum_{i=1}^{M} \sum_{j=1}^{P} \sum_{k=1}^{Q} g_{ijk}^2 \\
&\geq \sum_{i=M+1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} g_{ijk}^2 \\
&\overset{(13)}{=} \sum_{i=M+1}^{I} \left(\sigma_i^{(1)} \right)^2 .
\end{aligned}$$

Analog folgen $\left\| \mathcal{X} - \widetilde{\mathcal{X}} \right\|^2 \geq \sum_{j=P+1}^{J} \left(\sigma_j^{(2)} \right)^2$ und $\left\| \mathcal{X} - \widetilde{\mathcal{X}} \right\|^2 \geq \sum_{k=Q+1}^{K} \left(\sigma_k^{(3)} \right)^2$. □

Bemerkung: Nach Satz 4 folgt für die abgeschnittene Tucker-Zerlegung $\| \mathcal{G} \| = \| \widetilde{\mathcal{X}} \|$, aber nicht $\| \mathcal{G} \| = \| \mathcal{X} \|$, da im Allgemeinen $\boldsymbol{A}\boldsymbol{A}^T \neq \boldsymbol{I}$ gilt.

3.6. Der Algorithmus zur abgeschnittenen Tucker-Zerlegung

Da der Fehler des approximierten Tensors direkt vom Fehler der einzelnen Faktormatrizen abhängt (14), kann man eine abgeschnittene Tucker-Zerlegung zu einer vorgegebenen Fehlerschranke $\epsilon > 0$ berechnen. Man erhält folgenden Algorithmus:

- Berechne die Produktmatrizen $\boldsymbol{M}$, $\boldsymbol{P}$ und $\boldsymbol{Q}$ mit Hilfe des gegebenen Tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$:

$$\begin{aligned}
\boldsymbol{M} &= \boldsymbol{X}_{(1)}\, \boldsymbol{X}_{(1)}^T \;\in\; \mathbb{R}^{I \times I} \\
\boldsymbol{P} &= \boldsymbol{X}_{(2)}\, \boldsymbol{X}_{(2)}^T \;\in\; \mathbb{R}^{J \times J} \\
\boldsymbol{Q} &= \boldsymbol{X}_{(3)}\, \boldsymbol{X}_{(3)}^T \;\in\; \mathbb{R}^{K \times K}.
\end{aligned}$$

- Berechne die Eigenwerte und Eigenvektoren dieser Produktmatrizen.
- Verwende jeweils die M, P und Q größten Eigenwerte $\left(\sigma_i^{(1)} \right)^2$, $\left(\sigma_j^{(2)} \right)^2$ und $\left(\sigma_k^{(3)} \right)^2$,

so dass

$$\begin{aligned}
\sum_{i=M+1}^{I}\left(\sigma_i^{(1)}\right)^2 &\leq \frac{\varepsilon^2}{3}\sum_{i=1}^{I}\left(\sigma_i^{(1)}\right)^2 = \frac{\varepsilon^2}{3}\|\,\boldsymbol{X}_{(1)}\,\|^2 = \frac{\varepsilon^2}{3}\|\,\mathcal{X}\,\|^2,\\
\sum_{j=P+1}^{J}\left(\sigma_j^{(2)}\right)^2 &\leq \frac{\varepsilon^2}{3}\sum_{j=1}^{J}\left(\sigma_j^{(2)}\right)^2 = \frac{\varepsilon^2}{3}\|\,\boldsymbol{X}_{(2)}\,\|^2 = \frac{\varepsilon^2}{3}\|\,\mathcal{X}\,\|^2 \text{ und}\\
\sum_{k=Q+1}^{K}\left(\sigma_k^{(3)}\right)^2 &\leq \frac{\varepsilon^2}{3}\sum_{k=1}^{K}\left(\sigma_k^{(3)}\right)^2 = \frac{\varepsilon^2}{3}\|\,\boldsymbol{X}_{(3)}\,\|^2 = \frac{\varepsilon^2}{3}\|\,\mathcal{X}\,\|^2
\end{aligned}$$

und stelle die Koeffizientenmatrizen $\boldsymbol{A} \in \mathbb{R}^{I\times M}$, $\boldsymbol{B} \in \mathbb{R}^{J\times P}$ und $\boldsymbol{C} \in \mathbb{R}^{K\times Q}$ auf, die die entsprechenden normierten Eigenvektoren als Spalten enthalten.

- Berechne den Kerntensor $\mathcal{G}$ mit einer der umgestellten Formeln aus (9), zum Beispiel

$$\boldsymbol{G}_{(1)} = \boldsymbol{A}^T\boldsymbol{X}_{(1)}(\boldsymbol{C}\otimes\boldsymbol{B}),$$

oder mit (12)

$$\begin{aligned}
\boldsymbol{A}^T\boldsymbol{X}_{(1)} &= \boldsymbol{V}_{(1)}\\
\boldsymbol{B}^T\boldsymbol{V}_{(2)} &= \boldsymbol{W}_{(2)}\\
\boldsymbol{C}^T\boldsymbol{W}_{(3)} &= \boldsymbol{G}_{(3)}.
\end{aligned}$$

Dieser Algorithmus liefert dann mit (14) eine abgeschnittene Tucker-Zerlegung mit Fehler

$$\begin{aligned}
\left\|\mathcal{X}-\tilde{\mathcal{X}}\right\|^2 &\leq \sum_{i=M+1}^{I}\left(\sigma_i^{(1)}\right)^2 + \sum_{j=P+1}^{J}\left(\sigma_j^{(2)}\right)^2 + \sum_{k=Q+1}^{K}\left(\sigma_k^{(3)}\right)^2\\
&\leq \frac{\varepsilon^2}{3}\|\,\mathcal{X}\,\|^2 + \frac{\varepsilon^2}{3}\|\,\mathcal{X}\,\|^2 + \frac{\varepsilon^2}{3}\|\,\mathcal{X}\,\|^2\\
&= \varepsilon^2\|\,\mathcal{X}\,\|^2.
\end{aligned}$$

Algorithmus 3 zeigt die abgeschnittene Tucker-Zerlegung in Kurzform.

3.7. Aufwand der Implementierung

Der Algorithmus wurde in C++ implementiert, siehe dazu auch Anhang A.1. Im folgenden habe der zu approximierende Tensor $\mathcal{X}$ die Größe $n\times n\times n$. Es wird zuerst exemplarisch der Aufwand für die Berechnung einer Faktormatrix $\boldsymbol{A}$ betrachtet:

- Matrixmultiplikation $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$ mit BLAS [1]:
  Da $\boldsymbol{X}_{(1)} \in \mathbb{R}^{n\times n^2}$ gilt, benötigt die Multiplikation $\mathcal{O}(n^4)$ Rechenoperationen.

```
Algorithmus 3: abgeschnittene Tucker-Zerlegung($\mathcal{X}$, $\varepsilon$)

$A \leftarrow M$ führende Eigenvektoren von $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$, so dass $\sum_{i=M+1}^{I} \left(\sigma_i^{(1)}\right)^2 \leq \frac{\varepsilon^2}{3} \sum_{i=1}^{I} \left(\sigma_i^{(1)}\right)^2$
$B \leftarrow P$ führende Eigenvektoren von $\boldsymbol{X}_{(2)}\boldsymbol{X}_{(2)}^T$, so dass $\sum_{j=P+1}^{J} \left(\sigma_j^{(2)}\right)^2 \leq \frac{\varepsilon^2}{3} \sum_{j=1}^{J} \left(\sigma_j^{(2)}\right)^2$
$C \leftarrow Q$ führende Eigenvektoren von $\boldsymbol{X}_{(3)}\boldsymbol{X}_{(3)}^T$, so dass $\sum_{k=Q+1}^{K} \left(\sigma_k^{(3)}\right)^2 \leq \frac{\varepsilon^2}{3} \sum_{k=1}^{K} \left(\sigma_k^{(3)}\right)^2$
$\mathcal{G} \leftarrow \mathcal{X} \times_1 \boldsymbol{A}^T \times_2 \boldsymbol{B}^T \times_3 \boldsymbol{C}^T$
return $\mathcal{G}$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$
```

- Berechnung der Eigenwerte von $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$ mit LAPACK [3]:
  Die Matrix $\boldsymbol{M} = \boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$ besitzt die Größe $n \times n$, die Eigenwerte können also in $\mathcal{O}(n^3)$ Operationen berechnet werden, vgl. [9].

- Bestimmung der Anzahl k der benötigten Eigenwerte, um die Fehlerschranke einzuhalten:
  Es werden anfangs alle n Eigenwerte addiert und dann wird eine while-Schleife in konstanter Zeit maximal n-mal durchlaufen. Das ergibt eine Laufzeit von $\mathcal{O}(n)$.

- Berechnung der k zugehörigen Eigenvektoren mit LAPACK:
  Dies kann wie die Berechnung der Eigenwerte in $\mathcal{O}(n^3)$ Operationen durchgeführt werden.

Nun wird der Aufwand für die Berechnung des Kerntensors $\mathcal{G}$ betrachtet:

- Matrixmultiplikation $\boldsymbol{A}^T\boldsymbol{X}_{(1)} = \boldsymbol{V}_{(1)}$ mit BLAS:
  Es gilt $\boldsymbol{A}^T \in \mathbb{R}^{k \times n}$ und $\boldsymbol{X}_{(1)} \in \mathbb{R}^{n \times n^2}$, es werden also $\mathcal{O}(kn^3)$ Rechenoperationen für die Matrixmultiplikation benötigt.

- Umsortieren von $\boldsymbol{V}_{(1)}$ nach $\boldsymbol{V}_{(2)}$:
  Für das Umsortieren müssen kn^2 Werte kopiert, aber keine Berechnungen durchgeführt werden. Die Laufzeit kann maximal $\mathcal{O}(kn^2)$ betragen.

- Matrixmultiplikation $\boldsymbol{B}^T\boldsymbol{V}_{(2)} = \boldsymbol{W}_{(2)}$ mit BLAS:
  Da $\boldsymbol{B}^T \in \mathbb{R}^{k \times n}$ und $\boldsymbol{V}_{(2)} \in \mathbb{R}^{n \times kn}$ gilt, wird das Produkt $\boldsymbol{W}_{(2)}$ mit $\mathcal{O}(k^2n^2)$ Operationen berechnet.

- Umsortieren von $\boldsymbol{W}_{(2)}$ nach $\boldsymbol{W}_{(3)}$:
  Kopieren von k^2n Werten ergibt eine Laufzeit von höchstens $\mathcal{O}(k^2n)$.

- Matrixmultiplikation $\boldsymbol{C}^T\boldsymbol{W}_{(3)} = \boldsymbol{G}_{(3)}$ mit BLAS:
  Für $\boldsymbol{C}^T \in \mathbb{R}^{k \times n}$ und $\boldsymbol{W}_{(3)} \in \mathbb{R}^{n \times k^2}$ wird die Matrixmultiplikation in $\mathcal{O}(k^3n)$ Rechenoperationen durchgeführt.

Damit ergibt sich eine Gesamtlaufzeit der abgeschnittenen Tucker-Zerlegung von $\mathcal{O}(n^4)$ Operationen. Diese Laufzeit wird von der Matrixmultiplikation zu Beginn mit $\mathcal{O}(n^4)$ Rechenoperationen dominiert. Für nicht-symmetrische Tensoren muss diese dreimal ausgeführt werden. Der Kerntensor $\mathcal{G}$ wurde mit (12) in $\mathcal{O}(kn^3)$ Rechenoperationen bestimmt. Würde man diesen mit einer der umgestellten Formeln (9), z.B.

$$\boldsymbol{G}_{(1)} = \boldsymbol{A}^T \boldsymbol{X}_{(1)} (\boldsymbol{C} \otimes \boldsymbol{B}),$$

berechnen, so hätte schon das Produkt von $\boldsymbol{X}_{(1)}$ mit $\boldsymbol{C} \otimes \boldsymbol{B}$ Komplexität $\mathcal{O}(k^2 n^3)$.

Der benötigte Speicherplatz während allen Berechnungen ist $\mathcal{O}(n^3)$, da der Tensor $\mathcal{X}$ explizit aufgestellt werden muss und am Ende für die Berechnung des Kerntensors $\mathcal{G}$ benötigt wird.

3.8. Laufzeitanalyse

Um die Laufzeit zu analysieren, wurde der Algorithmus mit mehreren symmetrischen trivariaten Funktionen getestet. Hierbei enthält ein Tensorelement x_{ijk} den Funktionswert $f(x_i, y_j, z_k)$. Der Einfachheit halber gilt auch hier immer $n := I = J = K$. Mit dieser Einschränkung und der Symmetrie folgt, dass $\boldsymbol{A} = \boldsymbol{B} = \boldsymbol{C}$ gilt. Deswegen muss nur eine Faktormatrix berechnet werden und nicht wie im allgemeinen Fall drei. Die Anzahl der Eigenwerte $M = P = Q$ wird mit k bezeichnet. Die auf einem Kern eines Intel Core2 Prozessors gemessenen CPU-Zeiten enthalten die Zeit zum Aufstellen der Matrix $\boldsymbol{X}_{(1)}$ und die Zeit zur Berechnung der Faktormatrix $\boldsymbol{A}$, sowie des Kerntensors $\mathcal{G}$. Zur Bestimmung des Fehlers ε wird einmal die Norm des Tensors $\mathcal{X}$ und die Norm des Kerntensors $\mathcal{G}$ berechnet. Damit kann der Fehler mittels

$$\varepsilon = \frac{\| \mathcal{X} - \widetilde{\mathcal{X}} \|}{\| \mathcal{X} \|} = \frac{\sqrt{\| \mathcal{X} \|^2 - \| \mathcal{G} \|^2}}{\| \mathcal{X} \|}$$

berechnet werden. Die Dauer hierfür und für die Bestimmung der Kompressionsrate $\kappa = \frac{MI + M^3}{I^3}$ sind auch in den Laufzeiten enthalten.

Zuerst wurde der Algorithmus mit der Funktion

$$f(x, y, z) = \frac{1}{\sqrt{1 + x^2 + y^2 + z^2}} \tag{15}$$

und den Stützstellen $x_i = y_i = z_i = \frac{1}{n}(i - \frac{1}{2})$, $i = 1, \dots, n$ getestet. Tabelle 1 enthält die gemessenen Ergebnisse. Diese Funktion lässt sich sehr gut mit der Tucker-Zerlegung approximieren. Auch für große Dimensionen n ändert sich die benötigte Anzahl M an Eigenvektoren für eine vorgegebene Fehlerschranke kaum. Anders sieht dies für singuläre Funktionen, z.B.

$$g(x, y, z) = \frac{1}{\sqrt{x^2 + y^2 + z^2}}, \tag{16}$$

	$\varepsilon = 10^{-3}$			$\varepsilon = 10^{-5}$		
n	k	Zeit$[s]$	κ	k	Zeit$[s]$	κ
100	2	0.05	$2.08 \cdot 10^{-4}$	3	0.06	$3.27 \cdot 10^{-4}$
200	2	0.62	$5.10 \cdot 10^{-5}$	3	0.64	$7.84 \cdot 10^{-5}$
300	2	2.49	$2.25 \cdot 10^{-5}$	3	2.51	$3.43 \cdot 10^{-5}$
400	2	6.89	$1.26 \cdot 10^{-5}$	3	6.92	$1.92 \cdot 10^{-5}$
500	2	15.40	$8.06 \cdot 10^{-6}$	4	16.88	$1.65 \cdot 10^{-5}$
600	2	30.03	$5.59 \cdot 10^{-6}$	3	30.77	$8.46 \cdot 10^{-6}$
700	2	53.22	$4.10 \cdot 10^{-6}$	4	57.42	$8.35 \cdot 10^{-6}$
800	2	89.40	$3.14 \cdot 10^{-6}$	3	88.02	$4.74 \cdot 10^{-6}$
900	2	140.80	$2.48 \cdot 10^{-6}$	3	137.01	$3.74 \cdot 10^{-6}$
1 000	2	205.78	$2.01 \cdot 10^{-6}$	3	203.94	$3.03 \cdot 10^{-6}$

Tabelle 1: abgeschnittene Tucker-Zerlegung für $f(x, y, z)$.

aus. Für diese Funktion werden für die gleiche Genauigkeit deutlich mehr Eigenvektoren benötigt. Zusätzlich fallen die Eigenwerte auch nicht so stark ab, was man daran erkennen kann, dass für eine Erhöhung der Genauigkeit mehr zusätzliche Eigenvektoren benötigt werden als in (15). Außerdem ist die Anzahl der Eigenvektoren nicht unabhängig von der Größe der Matrix. Die Ergebnisse zu (16) finden sich in Tabelle 2.

	$\varepsilon = 10^{-3}$			$\varepsilon = 10^{-5}$		
n	k	Zeit$[s]$	κ	k	Zeit$[s]$	κ
100	8	0.07	$1.31 \cdot 10^{-3}$	12	0.08	$2.93 \cdot 10^{-3}$
200	8	0.70	$2.64 \cdot 10^{-4}$	13	0.76	$6.00 \cdot 10^{-4}$
300	9	2.97	$1.27 \cdot 10^{-4}$	14	3.03	$2.57 \cdot 10^{-4}$
400	9	8.01	$6.76 \cdot 10^{-5}$	15	8.18	$1.46 \cdot 10^{-4}$
500	9	17.57	$4.18 \cdot 10^{-5}$	15	17.94	$8.70 \cdot 10^{-5}$
600	10	34.22	$3.24 \cdot 10^{-5}$	16	33.46	$6.34 \cdot 10^{-5}$
700	10	61.53	$2.33 \cdot 10^{-5}$	16	59.09	$4.46 \cdot 10^{-5}$
800	10	97.77	$1.76 \cdot 10^{-5}$	16	96.16	$3.30 \cdot 10^{-5}$
900	10	149.99	$1.37 \cdot 10^{-5}$	17	151.73	$2.77 \cdot 10^{-5}$
1 000	10	221.35	$1.10 \cdot 10^{-5}$	17	225.78	$2.19 \cdot 10^{-5}$

Tabelle 2: abgeschnittene Tucker-Zerlegung für $g(x, y, z)$.

Als drittes wurde die Funktion

$$h(x, y, z) = x + y + z \tag{17}$$

getestet. Die Laufzeiten und Kompressionsraten finden sich in Tabelle 3. Diese Funktion lässt sich ähnlich gut wie (15) approximieren, da sie nicht singulär ist.

Vergleicht man die Laufzeiten der einzelnen Funktionen, so stellt man fest, dass die Zeiten für eine feste Größe in der gleichen Größenordnung liegen. Die Dauer der Berechnung hängt also kaum davon ab, wie gut sich die Funktion approximieren lässt.

	$\varepsilon = 10^{-3}$			$\varepsilon = 10^{-5}$		
n	k	Zeit[s]	κ	k	Zeit[s]	κ
100	2	0.04	$2.08 \cdot 10^{-3}$	2	0.04	$2.08 \cdot 10^{-3}$
200	2	0.46	$5.10 \cdot 10^{-5}$	2	0.46	$5.10 \cdot 10^{-5}$
300	2	1.96	$2.25 \cdot 10^{-5}$	2	1.96	$2.25 \cdot 10^{-5}$
400	2	5.61	$1.26 \cdot 10^{-5}$	2	5.61	$1.26 \cdot 10^{-5}$
500	2	12.92	$8.06 \cdot 10^{-6}$	2	12.92	$8.06 \cdot 10^{-6}$
600	2	25.72	$5.59 \cdot 10^{-6}$	2	25.72	$5.59 \cdot 10^{-6}$
700	2	46.36	$4.10 \cdot 10^{-6}$	2	46.36	$4.10 \cdot 10^{-6}$
800	2	77.40	$3.14 \cdot 10^{-6}$	2	77.40	$3.14 \cdot 10^{-6}$
900	2	121.91	$2.48 \cdot 10^{-6}$	2	121.91	$2.48 \cdot 10^{-6}$
1 000	2	183.17	$2.01 \cdot 10^{-6}$	2	183.17	$2.01 \cdot 10^{-6}$

Tabelle 3: abgeschnittene Tucker-Zerlegung für $h(x, y, z)$.

3.9. Anwendung

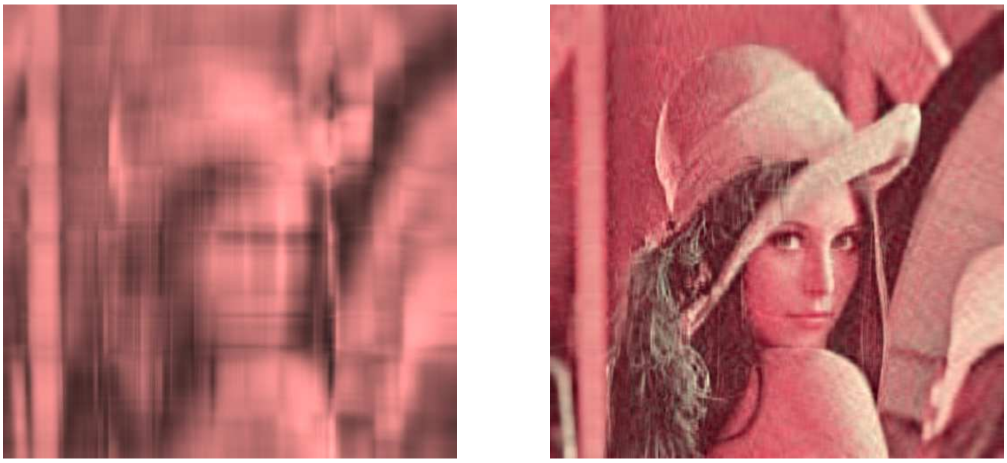
Mit der abgeschnittenen Tucker-Zerlegung wurde versucht, Bilder zu komprimieren. Dabei ist die erste Dimension die Höhe des Bildes, die zweite die Breite des Bildes und die dritte Dimension besteht aus den einzelnen Farbkanälen (im RGB-Modus). Als Testbild wird das bekannte Lena-Bild (http://sipi.usc.edu/database/misc/4.2.04.tiff) mit einer Größe von 512×512 Pixel verwendet. Es wird die abgeschnittene Tucker-Zerlegung zu verschiedenen M, P und Q berechnet, um Bilder mit einem Fehler ε von 1%, 3%, 5%, 10% und 20% zu erhalten. Dabei bezeichnet $\kappa = \frac{IM+JP+KQ+MPQ}{IJK}$ die Kompressionsrate, also das Verhältnis zwischen dem Speicherplatz des komprimierten Bildes und des Originals.

M	P	Q	ε	κ
1	1	1	0.2895	0.0013
7	7	1	0.1989	0.0092
42	37	2	0.1039	0.0554
84	76	3	0.0500	0.1285
160	150	3	0.0306	0.2934
412	402	3	0.0103	1.1618

Tabelle 4: Approximation eines Bildes durch die abgeschnittene Tucker-Zerlegung.

Schaut man sich die einzelnen Bilder (Abbildung 5) zu den jeweiligen Fehlern an, so kann man an Hand der auftauchenden Farben gut erkennen, ob $Q = 1$, $Q = 2$ oder $Q = 3$ gilt.

Bei genauerer Betrachtung der Werte für M, P, Q und dem zugehörigen Fehler ε stellt man fest, dass immer mehr Eigenwerte benötigt werden, um den Fehler zu halbieren. Dies steht im direkten Zusammenhang mit dem Abfall der Eigenwerte der zugehörigen Matrizen. Die ersten Eigenwerte fallen stark ab, wohingegen die letzten Eigenwerte nur noch schwach abfallen. Das heißt, zu Beginn kann man mit wenigen Eigenwerten mehr eine große Verbesserung erzielen, aber am Ende werden viele zusätzliche Eigenwerte für eine



(a) 20% Fehler

(b) 10% Fehler

(c) 5% Fehler

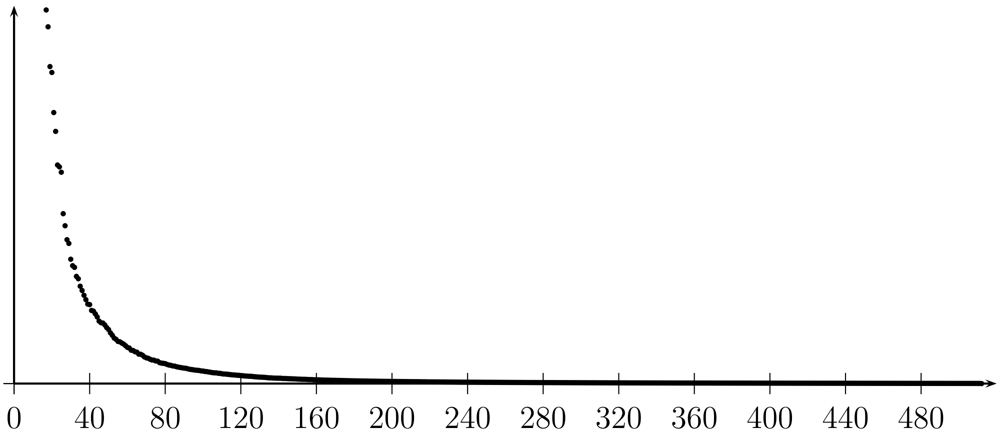
(d) 3% Fehler

(e) 1% Fehler

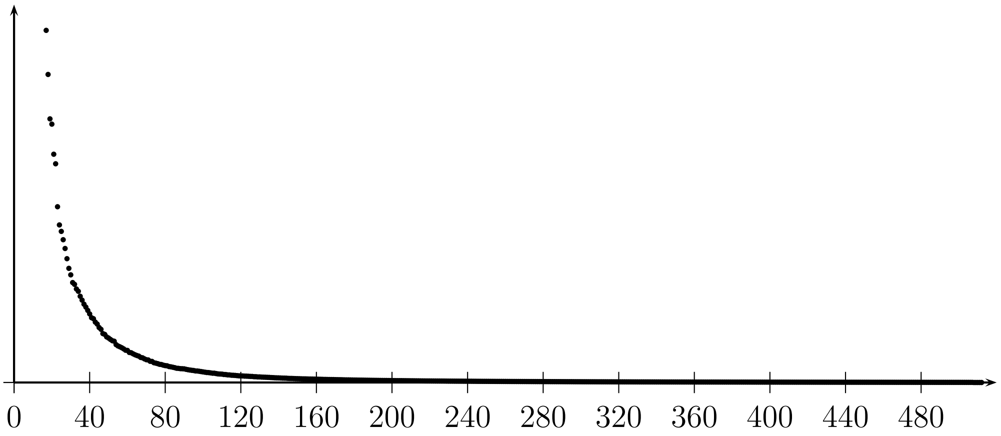
(f) original

Abbildung 5: Approximation eines Bildes durch die abgeschnittene Tucker-Zerlegung.

kleine Verbesserung benötigt. Deswegen wird bei guten Approximationen der benötigte Speicherplatz schnell groß und kann sogar den Speicherplatz des ursprünglichen Bildes überschreiten. Daher eignet sich die abgeschnittene Tucker-Zerlegung nicht zur Bildkompression.



(a) Eigenwerte der Matrix $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$



(b) Eigenwerte der Matrix $\boldsymbol{X}_{(2)}\boldsymbol{X}_{(2)}^T$

Abbildung 6: Eigenwerte zur Tucker-Zerlegung des Lena-Testbildes.

Zusätzlich wurde die Tucker-Zerlegung auf Videos angewandt. Dabei ist $I = 430$ Pixel die Höhe der einzelnen Bilder, $J = 740$ Pixel ist die Breite der Bilder und $K = 185$ ist die Anzahl der Bilder. Für jede der drei Farben wurde eine unabhängige Tucker-Zerlegung berechnet. Man kann die Farbe eines Pixels im RGB-Modus auch als eine Zahl betrachten, allerdings würde sich eine Approximation nur auf die hinteren Stellen auswirken. Dies hätte zur Folge, dass es eine große Abweichung bei den Blautönen gäbe, eventuell eine geringe Abweichung bei den Grüntönen und keine Abweichung bei den Rottönen. Durch eine zu große Änderung an der Grenze zwischen Grün und Blau könnte es zusätzlich zu Farbsprüngen kommen.

Zum Testen wurde ein Morphing-Video (RTL Extra, Sendung vom 12.04.2010) verwendet, in dem ein 6-jähriges Mädchen zur 50-jährigen Frau altert. Tabelle 5 zeigt die in 81.65 Sekunden und 92.13 Sekunden berechneten Ergebnisse.

Farbe	M	P	Q	ε	κ
r	48	31	7	0.0290	0.0010
g	57	36	7	0.0300	0.0012
b	73	46	10	0.0310	0.0019
r	132	92	36	0.0097	0.0099
g	134	94	34	0.0098	0.0098
b	174	126	102	0.0103	0.0414

Tabelle 5: Approximation eines Videos durch die abgeschnittene Tucker-Zerlegung.

Man sieht, dass sich nicht unbedingt alle Farben gleich gut approximieren lassen. In diesem Fall lässt sich Blau am schlechtesten approximieren. An den einzelnen Bildern des Videos mit 3% Fehler kann man auch erkennen, dass sich unterschiedliche Bereiche verschieden gut approximieren lassen. Der gleich bleibende Hintergrund mit dem Farbverlauf sieht auch bei 3% Fehler dem Original sehr ähnlich. Das Gesicht ist noch erkennbar, aber deutlich unschärfer als beim Original. Die hochlaufenden Zahlen am rechten Rand lassen sich nicht mehr fehlerfrei auseinander halten und das Logo auf der linken Seite ist komplett verschwunden. Die Bilder mit 1% Fehler kann man kaum vom Original unterscheiden, sie sind nur minimal unschärfer, wenn man die Bereiche um die Augen und die Konturen betrachtet.

Im Vergleich zum Lena-Testbild stellt man fest, dass die Kompressionsraten bei gleicher Fehlerschranke deutlich kleiner sind. Allerdings erscheint ein einzelnes Bild aus dem Video mit 3% Fehler schlechter als das Lena-Bild mit 3% Fehler. Insgesamt betrachtet ist die abgeschnittene Tucker-Zerlegung eher für Videos geeignet, da dort die dritte Dimension nicht deutlich kleiner als die ersten beiden ist und so die Kompressionsraten nicht zu groß werden.

(a) original (b) original

(c) 1% Fehler (d) 1% Fehler

(e) 3% Fehler (f) 3% Fehler

Abbildung 7: Approximation eines Videos durch die abgeschnittene Tucker-Zerlegung.

(a) original (b) original

(c) 1% Fehler (d) 1% Fehler

(e) 3% Fehler (f) 3% Fehler

Abbildung 8: Approximation eines Videos durch die abgeschnittene Tucker-Zerlegung.

4. Alternating Least Squares Algorithmus

4.1. Problemstellung

Mit dem ALS Algorithmus wird eine Rang-R Approximation an einen Tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ gesucht. Dabei wird versucht, den Ausdruck

$$\min_{\widetilde{\mathcal{X}}_R} \left\| \mathcal{X} - \widetilde{\mathcal{X}}_R \right\| \qquad \text{mit } \widetilde{\mathcal{X}}_R = \sum_{r=1}^{R} \boldsymbol{a}_r \circ \boldsymbol{b}_r \circ \boldsymbol{c}_r \tag{18}$$

zu minimieren, mit R als Parameter für den Rang der gesuchten Approximation und den Vektoren $\boldsymbol{a}_r \in \mathbb{R}^I$, $\boldsymbol{b}_r \in \mathbb{R}^J$, $\boldsymbol{c}_r \in \mathbb{R}^K$ mit $r = 1, \dots, R$.

Für $R =$Rang$(\mathcal{X})$ ist die Lösung eine exakte Zerlegung des gegebenen Tensors in R Rang-1 Tensoren (siehe Kapitel 2.6).

Seien im Folgenden $\boldsymbol{A} = (a_{ir}) \in \mathbb{R}^{I \times R}$, $\boldsymbol{B} = (b_{jr}) \in \mathbb{R}^{J \times R}$, $\boldsymbol{C} = (c_{kr}) \in \mathbb{R}^{K \times R}$, dann erhält man eine elementweise Darstellung von $\widetilde{\mathcal{X}}_R$ durch

$$\widetilde{x}_{ijk} = \sum_{r=1}^{R} a_{ir} \cdot b_{jr} \cdot c_{kr}. \tag{19}$$

Die tatsächliche Berechnung der Approximation erfolgt durch Reduktion des Ausdrucks auf ein eindimensionales Least Squares Problem. Hierbei werden zwei Dimensionen fixiert und für die dritte Dimension wird eine Least Squares Lösung berechnet.

Satz 10. *Seien $\boldsymbol{B}$, $\boldsymbol{C}$ fest. Dann führt (18) zu folgendem Minimierungsproblem:*

$$\min_{\widetilde{\boldsymbol{A}}} \left\| \boldsymbol{X}_{(1)} - \widetilde{\boldsymbol{A}} (\boldsymbol{C} \odot \boldsymbol{B})^T \right\|. \tag{20}$$

Beweis. Seien $\boldsymbol{B}$, $\boldsymbol{C}$ und R gegeben.
Betrachte

$$\widetilde{x}_{ijk} = \sum_{r=1}^{R} a_{ir} \cdot b_{jr} \cdot c_{kr} \tag{21}$$

und setze

$$g_{sr} := b_{jr} \cdot c_{kr} \text{ mit } s = J \cdot (k-1) + j,$$

so dass sich in Matrixschreibweise $(g_{sr}) = \boldsymbol{G} = (\boldsymbol{C} \odot \boldsymbol{B}) \in \mathbb{R}^{S \times R}$ mit $S = J \cdot K$ ergibt.

$$\Rightarrow \widetilde{x}_{is} = \sum_{r=1}^{R} a_{ir} g_{sr}, \tag{22}$$

mit der Matrixtransformation $(\widetilde{x}_{is}) = \widetilde{\boldsymbol{X}}_{(1)} \in \mathbb{R}^{I \times J \cdot K}$ des gesuchten Tensors $\widetilde{\boldsymbol{\mathfrak{X}}}_R$. Nun entspricht aber (22) genau der Matrixmultiplikation von $\boldsymbol{A}$ mit der Matrix $\boldsymbol{G}^T$.
In Matrixschreibweise lautet dies:

$$\widetilde{\boldsymbol{X}}_{(1)} = \boldsymbol{A} \cdot \boldsymbol{G}^T = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1R} \\ a_{21} & a_{22} & \dots & a_{2R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1} & a_{I2} & \dots & a_{IR} \end{bmatrix} \begin{bmatrix} b_{11} \cdot c_{11} & b_{21} \cdot c_{11} & \dots & b_{J1} \cdot c_{K1} \\ b_{12} \cdot c_{12} & b_{22} \cdot c_{12} & \dots & b_{J2} \cdot c_{K2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1R} \cdot c_{1R} & b_{2R} \cdot c_{1R} & \dots & b_{JR} \cdot c_{KR} \end{bmatrix}.$$

Für die Lösung von (18) ergibt sich mit dem fixierten $\boldsymbol{G}^T$ und der Matrixdarstellung 1-ter Art des Tensors $\boldsymbol{\mathfrak{X}}$ die eindimensionale Minimierungsaufgabe:

$$\min_{\widetilde{\boldsymbol{A}}} \left\| \boldsymbol{X}_{(1)} - \widetilde{\boldsymbol{A}} \cdot \boldsymbol{G}^T \right\| = \min_{\widetilde{\boldsymbol{A}}} \left\| \boldsymbol{X}_{(1)} - \widetilde{\boldsymbol{A}}(\boldsymbol{C} \odot \boldsymbol{B})^T \right\|.$$

□

Bemerkung: Für die Richtungen $\boldsymbol{B}$, $\boldsymbol{C}$ erhält man analoge Minimierungsprobleme:

$$\min_{\widetilde{\boldsymbol{B}}} \left\| \boldsymbol{X}_{(2)} - \widetilde{\boldsymbol{B}}(\boldsymbol{C} \odot \boldsymbol{A})^T \right\| \tag{23}$$

$$\min_{\widetilde{\boldsymbol{C}}} \left\| \boldsymbol{X}_{(3)} - \widetilde{\boldsymbol{C}}(\boldsymbol{B} \odot \boldsymbol{A})^T \right\| \quad . \tag{24}$$

Hieraus ergeben sich sofort die nachfolgenden Matrixdarstellungen der gesuchten Approximation:

$$\widetilde{\boldsymbol{X}}_{(1)} = \boldsymbol{A}(\boldsymbol{C} \odot \boldsymbol{B})^T \tag{25}$$

$$\widetilde{\boldsymbol{X}}_{(2)} = \boldsymbol{B}(\boldsymbol{C} \odot \boldsymbol{A})^T \tag{26}$$

$$\widetilde{\boldsymbol{X}}_{(3)} = \boldsymbol{C}(\boldsymbol{B} \odot \boldsymbol{A})^T. \tag{27}$$

Korollar 2. *Das in (20) gestellte Minimierungsproblem besitzt folgende Lösung:*

$$\boldsymbol{A} = \boldsymbol{X}_{(1)}[(\boldsymbol{C} \odot \boldsymbol{B})^T]^\dagger. \tag{28}$$

Beweis. Aus der Linearen Algebra ist bekannt, dass die eindeutige Lösung des linearen Ausgleichsproblems der Form

$$\| \boldsymbol{F} - \boldsymbol{E}\boldsymbol{X} \| \to \min$$

mit $\boldsymbol{F} \in \mathbb{R}^{m \times n}, \boldsymbol{X} \in \mathbb{R}^{l \times n}$ und $\boldsymbol{E} \in \mathbb{R}^{m \times l}$ gegeben ist durch $\boldsymbol{X} = \boldsymbol{E}^\dagger \boldsymbol{F}$.
Weiterhin gilt:

$$\begin{aligned} \boldsymbol{X} &= \boldsymbol{E}^\dagger \boldsymbol{F} \\ \Leftrightarrow \boldsymbol{X}^T &= (\boldsymbol{E}^\dagger \boldsymbol{F})^T \\ \Leftrightarrow \boldsymbol{X}^T &= \boldsymbol{F}^T (\boldsymbol{E}^\dagger)^T. \end{aligned}$$

Für (28) folgt mit $\boldsymbol{X} = \boldsymbol{A}^T$, $\boldsymbol{F} = \boldsymbol{X}_{(1)}^T$, $\boldsymbol{E} = (\boldsymbol{C} \odot \boldsymbol{B})$, sowie $m = I, n = J \cdot K$ und $l = R$ genau die Behauptung:

$$\begin{aligned} \boldsymbol{A} &= \boldsymbol{X}_{(1)} \left[(\boldsymbol{C} \odot \boldsymbol{B})^\dagger \right]^T \\ \Leftrightarrow \boldsymbol{A} &= \boldsymbol{X}_{(1)} \left[(\boldsymbol{C} \odot \boldsymbol{B})^T \right]^\dagger . \end{aligned}$$

Hierbei wird die Eigenschaft $\left[\boldsymbol{X}^T\right]^\dagger = \left[\boldsymbol{X}^\dagger\right]^T$ der Pseudoinversen ausgenutzt. □

Wiederum analog ergeben sich die Lösungen für $\boldsymbol{B}$, $\boldsymbol{C}$:

$$\boldsymbol{B} = \boldsymbol{X}_{(2)}[(\boldsymbol{C} \odot \boldsymbol{A})^T]^\dagger \tag{29}$$

$$\boldsymbol{C} = \boldsymbol{X}_{(3)}[(\boldsymbol{B} \odot \boldsymbol{A})^T]^\dagger. \tag{30}$$

Bei dem iterativen ALS-Algorithmus in 3D wird bei gegebenen Startwerten für $\boldsymbol{B}$, $\boldsymbol{C}$ zunächst eine Lösung für $\boldsymbol{A}$ berechnet. Anschließend werden $\boldsymbol{A}$, $\boldsymbol{C}$ fixiert und es wird eine bessere Lösung für $\boldsymbol{B}$ gesucht und zuletzt wird das Minimierungsproblem für $\boldsymbol{C}$ bei festgehaltenen $\boldsymbol{A}$, $\boldsymbol{B}$ gelöst.

Diese Prozedur wird nun solange wiederholt, bis ein vorgegebenes Abbruchkriterium erfüllt wird, das heißt, entweder eine gewisse Anzahl an Iterationen überschritten wird oder in den erhaltenen Näherungen keine Verbesserungen mehr eintreten.

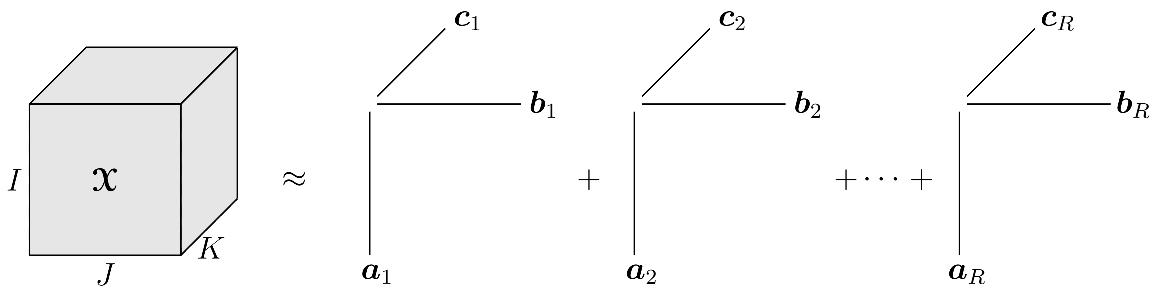


Abbildung 9: Approximation eines Tensors $\mathfrak{X} \in \mathbb{R}^{I \times J \times K}$ mit R Summanden (vgl. [10]).

4.1.1. Wahl des Parameters R

Um eine wesentliche Kompression des gegebenen Datenarrays zu erhalten, ist es erstrebenswert, R möglichst klein zu wählen. Deshalb wird in den meisten Fällen $R < \text{Rang}(\mathfrak{X})$ eintreten. Da jedoch erst ab $\text{Rang}(\mathfrak{X})$ Summanden eine exakte Darstellung des Tensors erfolgt und im Allgemeinen der Rang des gegebenen Tensors unbekannt ist, gilt es einen Kompromiss zwischen Kompression und Exaktheit zu finden, denn je kleiner das gewählte R, desto wahrscheinlicher ist es, dass der resultierende Fehler groß wird.

Eine Orientierung für die letztendliche Wahl von R können die bereits zuvor in Kapitel 2.6 erwähnten optimalen und maximalen Ränge für Tensoren bestimmter Größe geben. Auch hierbei ist zu beachten, dass es durchaus Tensoren geben kann, die schon sehr gut mit Tensoren niedrigerer Ränge approximiert werden können, dies jedoch meistens nicht gegeben ist [11].

4.1.2. Eigenschaften

Im Folgenden werden einige Eigenschaften der Darstellung eines Rang R-Tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ in der zerlegten Form betrachtet (vgl. [10]).
Zunächst ein hilfreiches

Lemma 3. *Seien* $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I \times J \times K}$ *mit* $\mathcal{X} = \boldsymbol{a} \circ \boldsymbol{b} \circ \boldsymbol{c}$ *und* $\mathcal{Y} = \boldsymbol{d} \circ \boldsymbol{e} \circ \boldsymbol{f}$.
Dann gilt

$$\big\langle \mathcal{X}, \mathcal{Y} \big\rangle = \big\langle \boldsymbol{a}, \boldsymbol{d} \big\rangle \cdot \big\langle \boldsymbol{b}, \boldsymbol{e} \big\rangle \cdot \big\langle \boldsymbol{c}, \boldsymbol{f} \big\rangle. \tag{31}$$

Beweis.

$$\begin{aligned}
\big\langle \mathcal{X}, \mathcal{Y} \big\rangle &= \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} x_{ijk} \cdot y_{ijk} \\
&= \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} a_i \cdot b_j \cdot c_k \cdot d_i \cdot e_j \cdot f_k \\
&= \left(\sum_{i=1}^{I} a_i \cdot d_i \right) \left(\sum_{j=1}^{J} b_j \cdot e_j \right) \left(\sum_{k=1}^{K} c_k \cdot f_k \right) \\
&= \big\langle \boldsymbol{a}, \boldsymbol{d} \big\rangle \cdot \big\langle \boldsymbol{b}, \boldsymbol{e} \big\rangle \cdot \big\langle \boldsymbol{c}, \boldsymbol{f} \big\rangle.
\end{aligned}$$

□

Mithilfe dieser Feststellung erhält man den

Satz 11. *Die Norm eines dreidimensionalen Rang-R Tensors lässt sich darstellen durch*

$$\| \mathcal{X} \|^2 = \sum_{r_1=1}^{R} \sum_{r_2=1}^{R} \big((\boldsymbol{A}^T \boldsymbol{A}) * (\boldsymbol{B}^T \boldsymbol{B}) * (\boldsymbol{C}^T \boldsymbol{C}) \big)_{r_1 r_2}. \tag{32}$$

Beweis.

$$\begin{aligned}
\| \boldsymbol{\mathcal{X}} \|^2 &= \Big\langle \sum_{r_1=1}^{R} \boldsymbol{a}_{r_1} \circ \boldsymbol{b}_{r_1} \circ \boldsymbol{c}_{r_1}, \sum_{r_2=1}^{R} \boldsymbol{a}_{r_2} \circ \boldsymbol{b}_{r_2} \circ \boldsymbol{c}_{r_2} \Big\rangle \\
&= \sum_{r_1=1}^{R} \sum_{r_2=1}^{R} \big\langle \boldsymbol{a}_{r_1} \circ \boldsymbol{b}_{r_1} \circ \boldsymbol{c}_{r_1}, \boldsymbol{a}_{r_2} \circ \boldsymbol{b}_{r_2} \circ \boldsymbol{c}_{r_2} \big\rangle \\
&\overset{\text{mit (3)}}{=} \sum_{r_1=1}^{R} \sum_{r_2=1}^{R} (\boldsymbol{a}_{r_1}^T \boldsymbol{a}_{r_2}) \cdot (\boldsymbol{b}_{r_1}^T \boldsymbol{b}_{r_2}) \cdot (\boldsymbol{c}_{r_1}^T \boldsymbol{c}_{r_2}) \\
&= \sum_{r_1=1}^{R} \sum_{r_2=1}^{R} \big((\boldsymbol{A}^T \boldsymbol{A}) * (\boldsymbol{B}^T \boldsymbol{B}) * (\boldsymbol{C}^T \boldsymbol{C}) \big)_{r_1 r_2} .
\end{aligned}$$

□

Nun betrachte man das Skalarprodukt $\big\langle \boldsymbol{\mathcal{X}}, \widetilde{\boldsymbol{\mathcal{X}}}_R \big\rangle$, wobei im Folgenden R der gewählte Parameter des ALS Algorithmus ist und nicht mehr dem Rang des Tensors $\boldsymbol{\mathcal{X}}$ entsprechen muss:

Lemma 4.

$$\big\langle \boldsymbol{\mathcal{X}}, \widetilde{\boldsymbol{\mathcal{X}}}_R \big\rangle = \big\langle \operatorname{vec}(\boldsymbol{\mathcal{X}}), (\boldsymbol{C} \odot \boldsymbol{B} \odot \boldsymbol{A}) \cdot \mathbf{1} \big\rangle, \tag{33}$$

wobei $\mathbf{1} \in \mathbb{R}^R$ *einen Vektor bezeichnet, dessen Einträge alle gleich* 1 *sind.*

Beweis. Es ist

$$(\boldsymbol{A} \odot \boldsymbol{B} \odot \boldsymbol{C}) \cdot \mathbf{1} = \left(\begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1R}b_{1R} \\ a_{11}b_{21} & a_{12}b_{22} & \dots & a_{1R}b_{2R} \\ \vdots & \vdots & \vdots & \vdots \\ a_{11}b_{J1} & a_{12}b_{J2} & \dots & a_{1R}b_{JR} \\ a_{21}b_{11} & a_{22}b_{12} & \dots & a_{2R}b_{JR} \\ \vdots & \vdots & \vdots & \vdots \\ a_{I1}b_{J1} & a_{I2}b_{J2} & \dots & a_{IR}b_{JR} \end{bmatrix} \odot \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1R} \\ \vdots & \vdots & \vdots & \vdots \\ c_{K1} & c_{K2} & \dots & c_{KR} \end{bmatrix} \right) \cdot \mathbf{1}$$

$$= \begin{bmatrix} a_{11}b_{11}c_{11} & + & a_{12}b_{12}c_{12} & + & \dots & + & a_{1R}b_{1R}c_{1R} \\ a_{11}b_{11}c_{21} & + & a_{12}b_{12}c_{22} & + & \dots & + & a_{1R}b_{1R}c_{2R} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_{11}b_{11}c_{K1} & + & a_{12}b_{12}c_{K2} & + & \dots & + & a_{1R}b_{1R}c_{KR} \\ a_{21}b_{21}c_{11} & + & a_{12}b_{22}c_{12} & + & \dots & + & a_{2R}b_{2R}c_{1R} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_{11}b_{J1}c_{11} & + & a_{11}b_{J2}c_{K2} & + & \dots & + & a_{1R}b_{JR}c_{KR} \\ a_{21}b_{21}c_{11} & + & a_{22}b_{12}c_{12} & + & \dots & + & a_{2R}b_{1R}c_{1R} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_{I1}b_{J1}c_{K1} & + & a_{I2}b_{J2}c_{K2} & + & \dots & + & a_{IR}b_{JR}c_{KR} \end{bmatrix} =: \boldsymbol{y} \in \mathbb{R}^{I \cdot J \cdot K}. \tag{34}$$

Betrachte außerdem

$$\begin{aligned}\langle \mathcal{X}, \widetilde{\mathcal{X}} \rangle &= \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} x_{ijk} \cdot \tilde{x}_{ijk} \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} x_{ijk} \cdot \left(\sum_{r=1}^{R} a_{ir} \cdot b_{jr} \cdot c_{kr} \right) \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} x_{ijk} \cdot (a_{i1} \cdot b_{j1} \cdot c_{k1} + \ldots + a_{iR} \cdot b_{jR} \cdot c_{kR}) \\ &\overset{\text{mit (34)}}{=} \big\langle \operatorname{vec}(\mathcal{X}), \boldsymbol{y} \big\rangle .\end{aligned}$$

Damit ergibt sich die Behauptung. □

Für die Norm des Fehlers gilt folgende Gleichheit:

$$\| \mathcal{X} - \widetilde{\mathcal{X}}_R \|^2 = \| \mathcal{X} \|^2 - 2\langle \mathcal{X}, \widetilde{\mathcal{X}}_R \rangle + \| \widetilde{\mathcal{X}}_R \|^2 . \tag{35}$$

Mithilfe von (32) und (33) kann man also die Norm des Fehlers einer Rang-R Approximation an einen dreidimensionalen Tensor anhand der erhaltenen Zerlegung berechnen, ohne den approximativen Tensor explizit aufstellen zu müssen:

$$\| \mathcal{X} - \widetilde{\mathcal{X}}_R \|^2 = \| \mathcal{X} \|^2 - 2\big\langle \operatorname{vec}(\mathcal{X}), (\boldsymbol{C} \odot \boldsymbol{B} \odot \boldsymbol{A}) \big\rangle + \sum_{r_1=1}^{R}\sum_{r_2=1}^{R} \big((\boldsymbol{A}^T\boldsymbol{A}) * (\boldsymbol{B}^T\boldsymbol{B}) * (\boldsymbol{C}^T\boldsymbol{C}) \big)_{r_1 r_2} .$$

4.2. Implementierung

Bei der Implementierung des ALS Algorithmus werden die in Lemma 1 gezeigten Eigenschaften des Khatri-Rao Produkts und des Hadamard-Produkts ausgenutzt.
Die Berechnung der Pseudoinversen von $(\boldsymbol{C} \odot \boldsymbol{B})^T$ kann nun vereinfacht werden, in dem man die Rechnung in zwei Schritte einteilt:

$$\begin{aligned}\boldsymbol{V} &= (\boldsymbol{B}^T\boldsymbol{B}) * (\boldsymbol{C}^T\boldsymbol{C}) \\ \big[(\boldsymbol{C} \odot \boldsymbol{B})^T\big]^\dagger &= (\boldsymbol{C} \odot \boldsymbol{B}) \cdot \boldsymbol{V}^\dagger .\end{aligned}$$

Somit muss lediglich von der symmetrischen $R \times R$-Matrix $\boldsymbol{V}$ die Pseudoinverse berechnet werden, anstelle von der nicht-quadratischen und unter Umständen großen und singulären Matrix $(\boldsymbol{C} \odot \boldsymbol{B})^T \in \mathbb{R}^{R \times J \cdot K}$.

4.2.1. Normalisierung

In jedem Iterationsschritt werden nach der Berechnung der Lösung für eine Dimension die Spalten der erhaltenen Matrix normalisiert, damit deren Einträge nicht zu sehr anwachsen. Für die jeweils letzte Dimension, also $\boldsymbol{C}$, werden die Längen der Spalten in dem Vektor

$\boldsymbol{\lambda} \in \mathbb{R}^R$ gespeichert und zurückgegeben.
So ergibt sich schließlich die Darstellung der Approximation durch:

$$\tilde{\mathcal{X}}_R = \sum_{r=1}^{R} \lambda_r \boldsymbol{a_r} \circ \boldsymbol{b_r} \circ \boldsymbol{c_r}. \tag{36}$$

4.2.2. Der ALS Algorithmus in 3D

Im Folgenden ist der ALS Algorithmus in Pseudocode gegeben. Dabei wurde die Version für dreidimensionale Tensoren gewählt, eine Verallgemeinerung für N-dimensionale Tensoren ist aber problemlos möglich.

Algorithmus 4: Alternating Least Squares $(\mathcal{X}, R)$

```
initialisiere $\boldsymbol{B} \in \mathbb{R}^{J \times R}$ und $\boldsymbol{C} \in \mathbb{R}^{K \times R}$
repeat
    $\boldsymbol{V} \leftarrow (\boldsymbol{C}^T\boldsymbol{C}) * (\boldsymbol{B}^T\boldsymbol{B})$
    $\boldsymbol{A} \leftarrow \boldsymbol{X}_{(1)}(\boldsymbol{C} \odot \boldsymbol{B})\boldsymbol{V}^\dagger$
    normalisiere Spalten von $\boldsymbol{A}$
    $\boldsymbol{V} \leftarrow (\boldsymbol{C}^T\boldsymbol{C}) * (\boldsymbol{A}^T\boldsymbol{A})$
    $\boldsymbol{B} \leftarrow \boldsymbol{X}_{(2)}(\boldsymbol{C} \odot \boldsymbol{A})\boldsymbol{V}^\dagger$
    normalisiere Spalten von $\boldsymbol{B}$
    $\boldsymbol{V} \leftarrow (\boldsymbol{B}^T\boldsymbol{B}) * (\boldsymbol{A}^T\boldsymbol{A})$
    $\boldsymbol{C} \leftarrow \boldsymbol{X}_{(3)}(\boldsymbol{B} \odot \boldsymbol{A})\boldsymbol{V}^\dagger$
    normalisiere Spalten von $\boldsymbol{C}$ und speichere die Normen in $\boldsymbol{\lambda} \in \mathbb{R}^R$
until Abbruchkriterium tritt ein
return $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{\lambda}$
```

4.2.3. Zusammensetzung der Approximation

Möchte man nur auf bestimmte Einträge der berechneten Approximation zugreifen, so bietet sich die elementweise Darstellung der Lösung an:

$$\widetilde{x}_{ijk} = \sum_{r=1}^{R} \lambda_r \cdot a_{ir} \cdot b_{jr} \cdot c_{kr}. \tag{37}$$

Bei der Approximation von Bildern benötigt man jedoch alle Einträge des Tensors. Hier liefert der obige Zugriff auf die Matrix einen sehr großen Aufwand. Daher empfiehlt es sich, den Tensor anhand der folgenden Identität wieder zusammenzusetzen:

$$\widetilde{\boldsymbol{X}}_{(3)} = \boldsymbol{C}\boldsymbol{\Lambda}(\boldsymbol{B} \odot \boldsymbol{A})^T, \tag{38}$$

wobei $\boldsymbol{\Lambda} = \operatorname{diag}(\boldsymbol{\lambda})$ eine Diagonalmatrix mit den Einträgen des Vektors $\boldsymbol{\lambda}$ auf der Diagonale ist.

(38) folgt sofort aus (27), da $\boldsymbol{\lambda}$ genau die berechneten Normen der Spalten von $\boldsymbol{C}$ sind. Analog zu (38) oder mithilfe von Umsortieren erhält man die Darstellungen $\widetilde{\boldsymbol{X}}_{(1)}$ und $\widetilde{\boldsymbol{X}}_{(2)}$.

4.2.4. Aufwand

Im Folgenden wird exemplarisch der Aufwand eines Iterationsschritts für die Richtung $\boldsymbol{A}$ betrachtet:

- Matrixmultiplikation von $\boldsymbol{B}^T\boldsymbol{B}$ und $\boldsymbol{C}^T\boldsymbol{C}$:
  Dies sind $JR^2 + KR^2$ Rechenoperationen.
- Die Berechnung des Hadamard-Produkts liefert $JR + KR$ Multiplikationen.
- Das Aufstellen einer $R \times R$-Matrix beträgt $\mathcal{O}(R^2)$.
- Berechnung der Pseudoinversen von $\boldsymbol{V}$:
  Die Berechnung der Pseudoinversen erfolgt, indem die Least Squares Lösung für $\boldsymbol{V}\boldsymbol{X} = \boldsymbol{I}$ mit einer Singulärwertzerlegung berechnet wird. Hierbei bezeichnet $\boldsymbol{I}$ die $r \times r$ -Einheitsmatrix. Der Aufwand dafür beträgt $\mathcal{O}(R^3)$.
- Das Khatri-Rao Produkt benötigt JKR Multiplikationen.
- Matrixmultiplikation von $(\boldsymbol{C} \odot \boldsymbol{B})^T\boldsymbol{V}^\dagger$:
  Es ergeben sich JKR^2 Rechenoperationen.
- Matrixmultiplikation von $\boldsymbol{X}_{(1)}((\boldsymbol{C} \odot \boldsymbol{B})^T\boldsymbol{V}^\dagger)$:
  Dies sind $IJKR$ Rechenoperationen.
- Normalisierung: R-mal wird die Norm eines Vektors der Länge I berechnet, also $\mathcal{O}(RI)$.

Hierbei wurden sowohl die Matrixmultiplikationen als auch die Berechnungen der Pseudoinversen und der Norm jeweils mit LAPACK [3] und BLAS [1] ausgeführt, um optimale Laufzeiten zu erhalten.
Insgesamt ergeben sich somit für einen Iterationsschritt $(2 \cdot R^2 + 3 \cdot R)(I + J + K) + (R + R^2)(IJ + IK + JK) + 3IJKR + \mathcal{O}(R^3) + \mathcal{O}(R^2)$ Multiplikationen und man erhält in jedem Iterationsschritt einen Aufwand von $\mathcal{O}(IJKR)$.

4.2.5. Konvergenz

Es ist nicht bewiesen, dass der ALS Algorithmus für das gestellte Minimierungsproblem (18) tatsächlich eine globale Lösung findet. Es hat sich jedoch in der Praxis gezeigt, dass der Algorithmus aufgrund seiner Least Squares Struktur konvergiert und dabei lokale Minima erreicht werden.
Weiterhin kann es eine sehr große Anzahl an Iterationen benötigen, bis sich Konvergenz einstellt und nur nach mehrmaligem Ausführen des Algorithmus mit unterschiedlichen

Startwerten kann auch beurteilt werden, ob die berechneten Lösungen lokal oder global sind [5, 11].
Inwiefern der ALS Algorithmus wirklich gute Ergebnisse liefert, wird also stark von verschiedenen Faktoren, die im Folgenden betrachtet werden, beeinflusst.

4.2.6. Abbruchkriterien

Ein Kriterium zum Abbruch der Iteration ist das Erreichen einer gewissen vorgegebenen Anzahl an Iterationsschritten. Hierbei ist die Wahl sowohl von dem gewählten R als auch von der Größe des Tensors abhängig.
Liegt R zum Beispiel nahe am tatsächlichen Rang des Tensors, so kann die Iteration schon schnell exakt sein. Ist jedoch das R sehr klein gewählt im Verhältnis zu I, J, K und der unbekannte Rang sehr hoch, kann es sehr viele Iterationen dauern, bis der Algorithmus zu einigermaßen genauen Ergebnissen kommt.
Deshalb reicht das Überschreiten eines bestimmten Iterationsschrittes nicht unbedingt aus, um gute Aussagen über die Approximation treffen zu können, und es bietet sich an, auch andere Kriterien zu betrachten.
Berechnet man in jedem Iterationsschritt die Norm des resultierenden Tensors (dies ist mithilfe der in Kapitel 4.1 erhaltenen Eigenschaft (32) möglich, ohne den Tensor aufzustellen) und vergleicht sie mit der des vorherigen Iterationsschrittes, so kann man leicht überprüfen, ob die Iteration konvergiert.
Weiterhin kann man auch die Konvergenz innerhalb der drei Richtungen beobachten. Scheinen diese sich nicht weiter zu verändern, ist es wenig sinnvoll, die Iteration weiter auszuführen.
Alternativ kann in Intervallen die Fehlernorm bestimmt und die Iteration bei Erreichen einer gewählten kleinen Fehlerschranke ϵ abgebrochen werden. Dabei ist jedoch zu beachten, dass, abhängig von R, unter Umständen die Iteration nicht zu einem Ende kommt und so eine maximal vorgegebene Iterationsanzahl den Abbruch erzwingen muss.

4.2.7. Wahl der Iterationsstartwerte

Die Anfangsinitialisierungen der Richtungen $\boldsymbol{B}$, $\boldsymbol{C}$ sind sowohl für die Güte der erhaltenen Approximation als auch für die Anzahl der benötigten Iterationen relevant.

- **Methode 1:** Die einfachste Methode ist es, die Matrizen mit zufällig generierten Zahlen aus einem vorher gewählten Intervall zu besetzen.

- **Methode 2:** Eine weitere Möglichkeit besteht darin, die mit Tucker berechneten R ersten linken Singulärvektoren einer Singulärwertzerlegung von $\boldsymbol{X}_{(2)}$, $\boldsymbol{X}_{(3)}$ für $\boldsymbol{B}$ und $\boldsymbol{C}$ zu wählen oder

- **Methode 3:** für $\boldsymbol{B}$ wie zuvor die R ersten linken Singulärvektoren von $\boldsymbol{X}_{(2)}$ zu nehmen und für $\boldsymbol{C}$ die R ersten Singulärvektoren einer Singulärwertzerlegung von

$(\boldsymbol{B}^T \boldsymbol{X}_{(2)})_{(3)}$. Im Unterschied zu Methode 2 sind nun die Werte von $\boldsymbol{B}$, $\boldsymbol{C}$ gekoppelt, was durchaus einen Einfluss auf die Iteration haben kann [6].

Bei dem mehrfachen Durchführen des Algorithmus zeigt sich hierbei, dass es bei gleich gewählten Parametern durchaus zu sichtbaren Unterschieden bei den berechneten Lösungen kommen kann.
Gerade mithilfe von Methode 2 und 3 ist es möglich, mit dem ALS Algorithmus gute Approximationen zu erzielen. Jedoch ist es nicht garantiert, dass dies wirklich immer zu den besten Ergebnissen führt. Auch hierbei kann es, abhängig von dem zu approximierenden Tensor, durchaus passieren, dass mit Zufallszahlen bessere Näherungen eintreten [5].

4.3. Laufzeitanalyse

Um die Laufzeiten des Algorithmus zu testen, wurde er auf dieselben drei trivariaten Funktionen wie aus Kapitel 3.8 angewendet. Hierbei wurden die gleichen Stützstellen gewählt und es gilt weiterhin $n = I = J = K$.
Zur Erinnerung noch einmal die Funktionen (15), (16) und (17):

$$\begin{aligned} f(x,y,z) &= \frac{1}{\sqrt{x^2+y^2+z^2+1}} \\ g(x,y,z) &= \frac{1}{\sqrt{x^2+y^2+z^2}} \\ h(x,y,z) &= x+y+z. \end{aligned}$$

R steht wie zuvor für den Rang der berechneten Approximation. Die CPU-Zeit wird in Sekunden gemessen, und die Spalte # It der folgenden Tabellen enthält die Anzahl der benötigten Iterationen.
Es bezeichnet

$$\kappa = \frac{(I+J+K)\cdot R}{I\cdot J\cdot K} = \frac{3\cdot n\cdot R}{n^3} = \frac{3\cdot R}{n^2}$$

den Grad der Kompression, das heißt, die Größe des benötigten Speichers der Approximation wird geteilt durch die ursprüngliche Größe des Tensors. Die betrachtete Genauigkeit ist wie in Kapitel 3.8 der relative Fehler von $\widetilde{\mathcal{X}}_R$ an $\mathcal{X}$:

$$\| \mathcal{X} - \widetilde{\mathcal{X}}_R \| \leq \epsilon \cdot \| \mathcal{X} \|.$$

Die Laufzeiten gelten nur für die Iterationen. Die Anfangsinitialisierung, sowie die Fehlernormbestimmung am Ende sind nicht berücksichtigt.

Zunächst wurden die vorgestellten Anfangsinitialisierungen getestet.
Für Methode 1 sind Zufallszahlen aus dem Intervall $[0,1]$ gewählt worden.
Für Methode 2 ergibt sich ein Spezialfall:

# It	**Methode 1**	**Methode 2**	**Methode 3**
10	0.010744200	0.0045938800	0.003836340
100	0.010732900	0.0014200000	0.001379210
1 000	0.001444770	0.0004345517	0.000434346
5 000	0.000426975	0.0004266760	0.000426976

Tabelle 6: Vergleich der ϵ beim ALS für (15) und $n = 200$.

# It	**Methode 1**	**Methode 2**	**Methode 3**
10	0.034684300	0.013174000	0.012815700
100	0.000173773	0.005065600	0.005058500
1 000	0.000173202	0.001832250	0.001831670
10 000	0.000167784	0.000619484	0.000619464
20 000	0.000162325	0.000442899	0.000442891

Tabelle 7: Vergleich der ϵ beim ALS für (17) und $n = 100$.

Da es sich bei allen drei Funktionen um symmetrische Funktionen handelt, entfällt das Umsortieren des Tensors von $\boldsymbol{X}_{(1)}$ nach $\boldsymbol{X}_{(2)}$ und $\boldsymbol{X}_{(3)}$. Folglich sind die R ersten linken Singulärvektoren von $\boldsymbol{B}$, $\boldsymbol{C}$ identisch.

Tabelle 6 und 7 zeigen das unterschiedliche Verhalten des Algorithmus bei verschiedenen Startwerten für die Funktion (15) mit $n = 200$ und für (17) mit $n = 100$.
Mit der wachsenden Anzahl an Iterationen fällt der relative Fehler monoton. Dabei ist der Abfall für Methode 2 und 3 bei der Funktion (15) zunächst wesentlich größer als mit Zufallszahlen, kann aber letztendlich keine höhere Genauigkeit als die Zufallszahlen erzeugen.
Bei (17) zeigt sich jedoch ein gegenteiliges Verhalten. Hier scheint mit zufällig erzeugten Ergebnissen ein besseres Verhalten des Fehlerabfalls einzutreten als mit den Singulärvektoren. Außerdem konvergiert der Fehler erst bei $I = 20\,000$, während dies bei (15) schon vor 5 000 Iterationen der Fall ist.
Insgesamt kann man sagen, dass bei Methode 3 die erhaltenen Näherungen minimal besser sind als bei Methode 2, die Iteration aber bei ähnlichen Anzahlen von Iterationsschritten konvergiert und auch die erhaltene Genauigkeit von der gleichen Größenordnung ist.
Um die Iteration genauer zu betrachten, wurde in Tabelle 8 das Konvergenzverhalten für (15) anhand kleiner n dargestellt. Hierbei ist für die Startwerte Methode 3 zur Initialisierung gewählt worden.
Man kann erkennen, dass bei wachsendem n die Anzahl der Iterationen im selben Bereich liegt. Außerdem behält die Genauigkeit die gleiche Größenordnung, auch wenn bei steigendem n eine minimale Vergrößerung zu beobachten ist.
Die Iterationsanzahl bis zur Konvergenz scheint also Tensor-abhängig zu sein.
Interessant ist es, weiterhin zu beobachten, dass die ALS Approximation in dem in Tabelle 8 betrachteten Fall bei einem nur unwesentlich kleineren ϵ konvergiert, als der Tucker Algorithmus es liefert (zum Vergleich: der Tucker Algorithmus liefert für alle n bei $k = 2$

n	R	# It	ϵ
10	2	3 044	0.000418999
20	2	3 164	0.000425003
40	2	3 010	0.000426497
80	2	2 993	0.000426870

Tabelle 8: Konvergenz des ALS für (15) mit Methode 3.

die Genauigkeit $\epsilon = 0.0004269$). Dieses Verhalten lässt sich auch bei den anderen betrachteten symmetrischen Funktionen beobachten.

In den weiteren Tabellen 9, 10, 11 und 12 wurde zur besseren Vergleichbarkeit jeweils Methode 2 zur Initialisierung gewählt.
Tabelle 9 zeigt die schnellsten Laufzeiten des ALS Algorithmus für (15) um ϵ zu erfüllen. In Tabelle 10 ist für (15) jeweils ein um 1 kleinerer Rang gewählt worden als in Tabelle 9. Man sieht, dass die geforderten ϵ auch erfüllt werden können, sich die Laufzeit jedoch wesentlich erhöht, da eine erhebliche Steigerung der Iterationen notwendig ist, um die gewünschte Genauigkeit zu erreichen (siehe für $\epsilon = 10^{-3}$: Rang 2 benötigt 173 Iterationen, Rang 3 lediglich 3, genauso für $\epsilon = 10^{-5}$).

	$\varepsilon = 10^{-3}$				$\varepsilon = 10^{-5}$			
n	R	#It	Zeit$[s]$	κ	R	# It	Zeit$[s]$	κ
100	3	3	0.05	$9.00 \cdot 10^{-4}$	8	4	0.05	$2.40 \cdot 10^{-3}$
200	3	3	0.41	$2.25 \cdot 10^{-4}$	8	4	0.40	$6.00 \cdot 10^{-4}$
300	3	3	1.30	$3.97 \cdot 10^{-4}$	8	4	1.65	$2.67 \cdot 10^{-4}$
400	3	3	2.95	$5.63 \cdot 10^{-5}$	8	4	2.98	$1.50 \cdot 10^{-4}$
500	3	3	5.49	$3.60 \cdot 10^{-5}$	8	4	5.81	$9.60 \cdot 10^{-5}$
600	3	3	9.15	$2.50 \cdot 10^{-5}$	8	4	9.72	$6.67 \cdot 10^{-5}$
700	3	3	14.61	$1.84 \cdot 10^{-5}$	8	4	15.34	$4.89 \cdot 10^{-5}$
800	3	3	21.79	$1.41 \cdot 10^{-5}$	8	4	22.75	$3.75 \cdot 10^{-5}$
900	3	3	30.65	$1.11 \cdot 10^{-5}$	8	5	40.48	$2.96 \cdot 10^{-5}$
1 000	3	3	42.34	$9.00 \cdot 10^{-6}$	8	4	44.00	$2.40 \cdot 10^{-5}$

Tabelle 9: Laufzeiten des ALS für (15).

Auch bei Tabelle 11 und 12 wurden jeweils die schnellsten Zeiten des Algorithmus aufgelistet und so ein höherer Rang in Kauf genommen.
Für die singuläre Funktion (16) sind lediglich die Ergebnisse für $\epsilon = 10^{-1}, 10^{-2}$ aufgelistet, da schon allein für $n = 100$ weit über 20 000 Iterationen notwendig waren, um einen Fehler der Größenordnung 10^{-3} oder 10^{-5} zu erhalten. Tendenziell ließ sich aber auch hier, genau wie bei dem Tucker Algorithmus, ein Anwachsen des Ranges bei steigendem n beobachten, auch wenn dies für die relativ großen ϵ noch nicht deutlich erkennbar ist.
Für die Funktionen (15) und (17) lässt sich jedoch sofort ablesen, dass der benötigte Rang R, aber auch die Anzahl an Iterationen, um das vorgegebene ϵ zu erfüllen, unabhängig

	$\varepsilon = 10^{-3}$				$\varepsilon = 10^{-5}$			
n	R	# It	Zeit$[s]$	κ	R	# It	Zeit$[s]$	κ
100	2	173	1.60	$6.00 \cdot 10^{-4}$	7	14	0.16	$2.10 \cdot 10^{-3}$
200	2	173	12.86	$1.50 \cdot 10^{-4}$	7	4	0.40	$5.25 \cdot 10^{-4}$
300	2	173	43.44	$6.67 \cdot 10^{-5}$	7	13	4.24	$2.33 \cdot 10^{-4}$
400	2	173	90.83	$3.75 \cdot 10^{-5}$	7	14	10.34	$1.31 \cdot 10^{-4}$
500	2	173	163.00	$2.40 \cdot 10^{-5}$	7	16	23.07	$8.40 \cdot 10^{-5}$

Tabelle 10: Laufzeiten des ALS für (15).

	$\varepsilon = 10^{-1}$				$\varepsilon = 10^{-2}$			
n	R	# It	Zeit$[s]$	κ	R	# It	Zeit$[s]$	κ
100	5	3	0.03	$1.50 \cdot 10^{-3}$	10	20	0.29	$3.00 \cdot 10^{-3}$
200	5	3	0.27	$3.75 \cdot 10^{-4}$	12	21	2.63	$9.00 \cdot 10^{-4}$
400	5	3	2.04	$9.38 \cdot 10^{-5}$	12	20	18.94	$2.25 \cdot 10^{-4}$
600	5	4	8.94	$4.17 \cdot 10^{-5}$	12	22	61.23	$1.00 \cdot 10^{-4}$

Tabelle 11: Laufzeiten des ALS für (16).

von der Größe von n ist.

Anhand der unterschiedlichen Ergebnisse für die drei Funktionen kann man erkennen, wie abhängig der ALS Algorithmus tatsächlich von dem zu approximierendem Datentensor ist. Für die singuläre Funktion ist es sehr schwierig (auch mit unterschiedlichen Startwerten) einigermaßen gute Genauigkeit zu erlangen, während sich (17) relativ gut und schnell approximieren lässt.
Ein direkter Vergleich der Ränge der Approximation und der Genauigkeit mit dem Tucker Algorithmus folgt in Kapitel 5.

4.4. Anwendung

Der ALS Algorithmus wurde genau wie in Kapitel 3.9 auf das Testbild 'Lena' (siehe Internet: http://sipi.usc.edu/database/misc/4.2.04.tiff) angewandt.
Der Datentensor hat die Größe $512 \times 512 \times 3$, wobei wie zuvor 512×512 die Pixelanzahl des Bildes ist und die dritte Richtung für die drei Farbkanäle Rot, Grün, Blau steht; es ist also $I = J = 512$ und $K = 3$.
Die dritte Dimension ist im Vergleich zu den ersten beiden sehr klein, was für die Wahl des Parameters R eine entscheidende Rolle spielt, denn höchstwahrscheinlich wird $\text{Rang}(\mathcal{X}) > 3$ sein. Dies hat weiter zur Folge, dass eine Wahl der Startwerte mit Methode 2 oder 3 nicht sinnvoll erscheint, da in diesem Fall für $\boldsymbol{C}$ $R \leq 3$ sein muss und somit das höchste zu wählende R lediglich $R = 3$ sein kann.
Aus diesem Grund wurden für das Bild Zufallszahlen im Intervall $[0, 255]$ als Initialisierungen für $\boldsymbol{B}$, $\boldsymbol{C}$ gewählt. Da die Werte eines Bildes Zufallszahlen sind, scheint diese Methode auch geeignet zu sein, um gute Ergebnisse zu erzielen.

	$\varepsilon = 10^{-3}$				$\varepsilon = 10^{-5}$			
n	R	# It	Zeit[s]	κ	R	# It	Zeit[s]	κ
100	3	3	0.05	$9.00 \cdot 10^{-4}$	3	5	0.08	$9.00 \cdot 10^{-4}$
200	3	3	0.41	$2.25 \cdot 10^{-4}$	3	5	0.68	$2.25 \cdot 10^{-4}$
400	3	5	4.91	$5.63 \cdot 10^{-5}$	3	7	6.87	$5.63 \cdot 10^{-5}$
600	3	5	15.25	$2.50 \cdot 10^{-5}$	3	6	18.29	$2.50 \cdot 10^{-5}$
800	3	3	21.78	$1.41 \cdot 10^{-5}$	3	5	36.45	$1.41 \cdot 10^{-5}$
1 000	3	5	70.57	$9.00 \cdot 10^{-6}$	3	6	84.68	$9.00 \cdot 10^{-6}$

Tabelle 12: Laufzeiten des ALS für (17).

In Tabelle 13 sind die benötigten Ränge, Iterationen und Kompressionsraten für die betrachteten ϵ aufgelistet. Hierbei zeigt sich, dass die Kompressionsraten im Vergleich zu dem Tucker Algorithmus geringer sind, da dieser jeweils noch zusätzlich zu den drei Matrizen den Kerntensor abspeichern muss.
Bei 20%, 10% und 5% sind weniger Konturen als bei dem Tucker Algorithmus zu erkennen, dafür ist das Farbsprektum höher. Für 3% und 1% sind jedoch kaum noch Unterschiede zum Originalbild zu erkennen (siehe Abbildung 10).
Insgesamt gestaltet sich doch ein Vergleich der Verfahren für Bilder schwieriger, da bei dem Tucker Algorithmus für jede Richtung unterschiedliche Parameter gesetzt sind, während beim ALS Algorithmus immer nur ein R gewählt werden kann.

ϵ	R	κ	# It
0.2889	1	$1.31 \cdot 10^{-3}$	1
0.1978	5	$6.53 \cdot 10^{-3}$	11
0.1059	25	$3.27 \cdot 10^{-3}$	80
0.0495	100	$1.31 \cdot 10^{-1}$	20
0.0309	200	$2.62 \cdot 10^{-1}$	20
0.0103	700	$9.14 \cdot 10^{-1}$	3

Tabelle 13: Werte des ALS für das Lena-Bild.

Weiterhin wurde versucht, denselben Videoausschnitt wie in Kapitel 3.9 (RTL Extra, Sendung vom 12.04.2010) zu komprimieren.
Bei dem entstandenen Tensor der Größe $740 \times 340 \times 185$ stehen die ersten beiden Dimensionen wieder für die Pixelzahl der einzelnen Bildsequenzen, also $I = 740$ und $J = 430$, während $K = 185$ die Anzahl der betrachteten Bilder angibt.
Um gute Ergebnisse zu erzielen, wurde der Algorithmus für jeden Farbkanal Rot, Grün, Blau getrennt angewandt, da bei einer Approximation in normaler RGB-Speicherweise nur die hinteren Werte verändert und somit ein bis zwei Farbkanäle sehr schlecht angenähert werden, wohingegen Rot nahezu exakt bleibt.
Als Startwerte für $\boldsymbol{B}$, $\boldsymbol{C}$ wurden wieder Zufallszahlen aus dem Intervall $[0, 255]$ gewählt.
Das Video hat geringere Kompressionsraten als das zuvor betrachtete Lena-Bild, da die einzelnen Dimensionen ausgeglichener sind, jedoch ist, wie man erkennen kann, eine höhere

Genauigkeit nötig, um Ergebnisse zu erhalten, die kaum noch Abweichungen zum Original aufzeigen.
Dabei hat es sich mit dem ALS Algorithmus als schwierig erwiesen, auf kleinere relative Fehler als 3% zu kommen und besonders der Farbkanal Blau ließ sich nur schwer annähern. Dafür war sowohl ein sehr hoher Rang als auch eine höhere Anzahl an Iterationen nötig als bei der Berechnung für die 3%, da der Fehler unter dieser Grenze nur sehr langsam abfällt.
Die Kompressionsraten sind leicht höher als beim Tucker Algorithmus, sind aber insgesamt betrachtet noch als gering anzusehen.

Farbe	ϵ	R	κ	# It
r	0.0301843	210	$4.83 \cdot 10^{-3}$	8
g	0.0305967	250	$5.75 \cdot 10^{-3}$	8
b	0.0336162	350	$8.06 \cdot 10^{-3}$	8
r	0.0174806	500	$1.15 \cdot 10^{-2}$	50
g	0.0188867	500	$1.15 \cdot 10^{-2}$	50
b	0.0253550	500	$1.15 \cdot 10^{-2}$	50

Tabelle 14: Werte des ALS für den Videoausschnitt.

(a) 20% Fehler

(b) 10% Fehler

(c) 5% Fehler

(d) 3% Fehler

(e) 1% Fehler

(f) original

Abbildung 10: ALS-Approximation des Lena-Bildes.

(a) original (b) original

(c) 2% Fehler (d) 2% Fehler

(e) 3% Fehler (f) 3% Fehler

Abbildung 11: Approximation eines Videos mit dem ALS.

(a) original (b) original

(c) 2% Fehler (d) 2% Fehler

(e) 3% Fehler (f) 3% Fehler

Abbildung 12: Approximation eines Videos mit dem ALS.

5. Vergleich der Methoden

In diesem Abschnitt werden die in Kapitel 3 und 4 behandelten Algorithmen mit der Adaptive Cross Approximation verglichen.
Die Funktionsweise ist grundlegend anders als zuvor, da hierbei nicht auf alle Einträge des zu approximierenden Tensors zugegriffen werden muss und eine erhebliche Reduzierung des Aufwandes gegeben ist.

5.1. Adaptive Cross Approximation

Das genaue Modell wird in [4] für Funktionen von zwei, drei und vier Variablen beschrieben. An dieser Stelle folgt eine kurze Vorstellung der Idee für trivariate Funktionen. Die Notation ist hierbei [4] entnommen.

Für $\kappa : X \times Y \times Z \to \mathbb{C}$ sei $\widetilde{R}_0(x,y,z) = \kappa(x,y,z)$ und

$$\widetilde{R}_k(x,y,z) = \widetilde{R}_{k-1}(x,y,z) - \frac{\widetilde{R}_{k-1}(x,y_k,z_k)}{\widetilde{R}_{k-1}(x_k,y_k,z_k)} \boldsymbol{\mathcal{A}}_{yz}\left[\widetilde{R}_{k-1}|x_k\right](y,z) \qquad \text{für } k = 0,1,2,\dots \tag{39}$$

Dabei bezeichnet $\boldsymbol{\mathcal{A}}_{yz}[f]$ die ACA für eine Funktion $f(y,z)$.
Diese ist von der Form

$$R_k(y,z) = R_{k-1}(y,z) - \frac{R_{k-1}(y,z_k)R_{k-1}(y_k,z)}{R_{k-1}(y_k,z_k)} \qquad \text{für } k = 0,1,2,\dots \tag{40}$$

mit $R_0(y,z) = f(y,z)$.

Dies bedeutet, dass für trivariate Funktionen zunächst eine Restriktion auf eine Funktion mit zwei Variablen erfolgt, auf die dann der ACA Algorithmus für zwei Dimensionen angewendet werden kann und man so schließlich eine Restriktion auf eine Funktion mit nur einer Variablen erhält.
Hierbei ist die gewählte Rekursionstiefe k der Rang der Approximation.
Bei der Approximation an einen Tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n\times n\times n}$, dessen Einträge die Auswertungen einer Funktion $\kappa(x_{i_1}, y_{i_2}, z_{i_3})$ mit fest gewählten Stützstellen $(x_{i_1}, y_{i_2}, z_{i_3})$ mit $i_1, i_2, i_3 = 1,\dots,n$ sind, erhält man die folgende Darstellung des dreidimensionalen Tensors:

$$\kappa(x_{i_1}, y_{i_2}, z_{i_3}) \approx \sum_{l=1}^{k} (w_l)_{i_1} \sum_{\nu=1}^{k_l} (u_{l\nu})_{i_2} (v_{l\nu})_{i_3}, \tag{41}$$

wobei k_l für den Rang der l-ten zweidimensionalen Approximation und k für den Rang der dreidimensionalen Approximation steht. Die drei Vektoren $w_l = (R_{k-1})_{1:n,i_2,i_3}$, $u_{l\nu} = (R_{k-1})_{i_1,1:n,i_3}$ und $v_{l\nu} = (R_{k-1})_{i_1,i_2,1:n}$ mit $l = 1,\dots,k$, $\nu = 1\dots,k_l$ sind dabei durch geeignet gewählte Indizes $(i_1^{(k)}, i_2^{(k)}, i_3^{(k)})$ bestimmt.
Durch dieses Prinzip wird lediglich ein Speicherplatz von $\mathcal{O}(n \cdot k_S)$ mit $k_S = \sum_{l=1}^{k} k_l$ benötigt und man erhält einen Aufwand von $\mathcal{O}(n \cdot \sum_{l=1}^{k} k_l^2)$ (vergleiche [4]).

5.2. Laufzeitvergleich

Es folgt eine Auflistung der Laufzeiten für die bereits in Kapitel 3 und 4 betrachteten Funktionen im Vergleich mit dem ACA Algorithmus.

In Tabelle 15 sind die Laufzeiten der drei Algorithmen für (15)

$$f(x, y, z) = \frac{1}{\sqrt{1 + x^2 + y^2 + z^2}}$$

dargestellt.

Dabei bezeichnet k den Rang bzw. die Rekursionstiefe der Algorithmen, während k_S wie in Kapitel 5.1 die Summe der Rekursionstiefen der zweidimensionalen ACA beschreibt, die bei der ACA für dreidimensionale Tensoren mehrfach angewandt wird.

ϵ ist wie zuvor die relative Genauigkeit und κ bezeichnet die jeweilige Kompressionsrate der Algorithmen. Für den ALS Algorithmus und Tucker Algorithmus stimmt diese mit denen bereits in Kapitel 3.9 und 4.3 erwähnten Raten ein, für den ACA Algorithmus lautet diese: $\kappa = (2k_S + 1)/n^2$.

Zunächst wird die Funktion (15) in Tabelle 15 betrachtet:

Für die gewählten n und ϵ benötigt der ACA Algorithmus durchweg 0.0 Sekunden.

Der Tucker Algorithmus dauert zwar für $n = 1\,000$ bereits 205.78 Sekunden, jedoch fällt auf, dass er lediglich den Rang $k = 2$ benötigt und somit bei den in Tabelle 17 gezeigten Kompressionsraten am besten abschneidet.

Der ALS Algorithmus zeigt schlechtere Resultate (obwohl er naiv gesehen schnellere Laufzeiten als der Tucker Algorithmus erzielt), da die gewählten Ränge k gerade bei $\epsilon = 10^{-5}$ über den des Tucker Algorithmus liegen, bei dem ACA Algorithmus sind die k_S jedoch noch größer.

Dass der ALS Algorithmus auch für $k = 2$ in den Bereich $\epsilon = 10^{-4}$ kommt, wurde in Kapitel 4.3 gezeigt, jedoch benötigt er dazu weitaus längere Zeit.

	$\varepsilon = 10^{-3}$							$\varepsilon = 10^{-5}$						
	ACA			Tucker		ALS		ACA			Tucker		ALS	
n	k	k_S	Zeit[s]	k	Zeit[s]	k	Zeit[s]	k	k_S	Zeit[s]	k	Zeit[s]	k	Zeit[s]
100	3	9	0.00	2	0.05	3	0.05	4	18	0.00	3	0.06	8	0.05
200	3	9	0.00	2	0.62	3	0.41	4	18	0.00	3	0.64	8	0.40
300	3	9	0.00	2	2.49	3	1.30	4	18	0.00	3	2.51	8	1.65
400	3	9	0.00	2	6.89	3	2.95	4	18	0.00	3	6.92	8	2.98
500	3	9	0.00	2	15.40	3	5.49	4	18	0.00	4	16.88	8	5.81
600	3	9	0.00	2	30.03	3	9.15	4	18	0.00	3	30.77	8	9.72
700	3	9	0.00	2	53.22	3	14.61	4	18	0.00	4	57.42	8	15.34
800	3	9	0.00	2	89.40	3	21.79	4	18	0.00	3	88.02	8	22.75
900	3	9	0.00	2	140.80	3	30.65	4	18	0.00	3	137.01	8	40.48
1 000	3	9	0.00	2	205.78	3	42.34	4	18	0.00	3	203.94	8	44.00

Tabelle 15: Vergleich von ACA, Tucker, ALS für (15).

In Tabelle 16 finden sich die Laufzeiten von dem Tucker Algorithmus und dem ACA Algorithmus für die singuläre Funktion (16):

$$g(x, y, z) = \frac{1}{\sqrt{x^2 + y^2 + z^2}}.$$

Wie bereits in Kapitel 4.3 erwähnt, liefert der ALS Algorithmus dafür keine vergleichbaren Ergebnisse.

Die Zeit für die Berechnung der Tucker Approximation liegt in der selben Größenordnung wie bei (15) und auch der ACA Algorithmus benötigt nur minimal länger als bei der regulären Funktion. Bei beiden Algorithmen wird das k beziehungsweise k_S größer, jedoch scheint es bei dem ACA Algorithmus im Gegensatz zu dem Tucker Algorithmus nicht monoton anzusteigen.

	$\varepsilon = 10^{-3}$					$\varepsilon = 10^{-5}$				
	ACA			Tucker		ACA			Tucker	
n	k	k_S	Zeit[s]	k	Zeit[s]	k	k_S	Zeit[s]	k	Zeit[s]
100	10	92	0.00	8	0.07	15	230	0.02	12	0.08
200	10	102	0.01	8	0.70	16	269	0.06	13	0.76
300	10	102	0.02	9	2.97	15	260	0.08	14	3.03
400	11	118	0.03	9	8.01	17	324	0.14	15	8.18
500	12	129	0.04	9	17.57	16	309	0.17	15	17.94
600	12	142	0.05	10	30.03	22	470	0.42	16	33.46
700	12	131	0.05	10	61.53	23	496	0.56	16	59.09
800	13	147	0.07	10	97.77	22	480	0.61	16	96.16
900	12	138	0.07	10	149.99	24	532	0.89	17	151.73
1 000	12	134	0.08	10	221.35	25	564	1.18	17	225.78

Tabelle 16: Vergleich von ACA, Tucker für (16).

Tabelle 17 zeigt die zu (15) gehörigen Kompressionsraten der Algorithmen. Bei allen drei Algorithmen können hierbei gute Ergebnisse erzielt werden, doch der Tucker Algorithmus weist die größte Kompression auf, da man, wie bereits erwähnt, aufgrund der Symmetrie der Funktionen wesentlichen Speicherplatz sparen kann.

Der ALS Algorithmus liefert geringfügig kleinere Kompressionsraten als der ACA Algorithmus.

Obwohl das k, beziehungsweise k_S bei größer werdendem n bei der singulären Funktion ansteigt, kann man in Tabelle 18 sehen, dass auch dort immer noch eine gute Kompression bei den beiden betrachteten Algorithmen erzielt werden kann.

Insgesamt scheint der ALS Algorithmus im Vergleich zu den anderen beiden Algorithmen nicht ratsam zur Anwendung auf Funktionen zu sein, da er die Eigenschaften, die sich der Tucker Algorithmus und der ACA Algorithmus gerade zu Nutze machen, nicht gewinnbringend einsetzen kann und man somit mit dem ALS Algorithmus schlechtere Ergebnisse erhält.

Der Tucker Algorithmus ist immer noch wesentlich langsamer als der ACA Algorithmus,

	$\varepsilon = 10^{-3}$			$\varepsilon = 10^{-5}$		
n	ACA	Tucker	ALS	ACA	Tucker	ALS
100	$1.90 \cdot 10^{-3}$	$2.08 \cdot 10^{-4}$	$9.00 \cdot 10^{-4}$	$3.70 \cdot 10^{-3}$	$3.27 \cdot 10^{-4}$	$2.40 \cdot 10^{-3}$
200	$4.75 \cdot 10^{-4}$	$5.10 \cdot 10^{-5}$	$2.25 \cdot 10^{-4}$	$9.25 \cdot 10^{-4}$	$7.84 \cdot 10^{-5}$	$6.00 \cdot 10^{-4}$
300	$2.11 \cdot 10^{-4}$	$2.25 \cdot 10^{-5}$	$3.97 \cdot 10^{-4}$	$4.11 \cdot 10^{-4}$	$3.43 \cdot 10^{-5}$	$2.67 \cdot 10^{-4}$
400	$1.19 \cdot 10^{-4}$	$1.26 \cdot 10^{-5}$	$5.63 \cdot 10^{-5}$	$2.31 \cdot 10^{-4}$	$1.92 \cdot 10^{-5}$	$1.50 \cdot 10^{-4}$
500	$7.60 \cdot 10^{-5}$	$8.06 \cdot 10^{-6}$	$3.60 \cdot 10^{-5}$	$1.48 \cdot 10^{-4}$	$1.65 \cdot 10^{-5}$	$9.60 \cdot 10^{-5}$
600	$5.28 \cdot 10^{-5}$	$5.59 \cdot 10^{-6}$	$2.50 \cdot 10^{-5}$	$1.03 \cdot 10^{-4}$	$8.46 \cdot 10^{-6}$	$6.67 \cdot 10^{-5}$
700	$3.88 \cdot 10^{-5}$	$4.10 \cdot 10^{-6}$	$1.84 \cdot 10^{-5}$	$7.55 \cdot 10^{-5}$	$8.35 \cdot 10^{-6}$	$4.89 \cdot 10^{-5}$
800	$2.97 \cdot 10^{-5}$	$3.14 \cdot 10^{-6}$	$1.41 \cdot 10^{-5}$	$5.78 \cdot 10^{-5}$	$4.74 \cdot 10^{-6}$	$3.75 \cdot 10^{-5}$
900	$2.35 \cdot 10^{-5}$	$2.48 \cdot 10^{-6}$	$1.11 \cdot 10^{-5}$	$4.57 \cdot 10^{-5}$	$3.74 \cdot 10^{-6}$	$2.96 \cdot 10^{-5}$
1 000	$1.90 \cdot 10^{-5}$	$2.01 \cdot 10^{-6}$	$9.00 \cdot 10^{-6}$	$3.70 \cdot 10^{-5}$	$3.03 \cdot 10^{-6}$	$2.40 \cdot 10^{-5}$

Tabelle 17: Vergleich der Kompressionsraten κ von ACA, Tucker, ALS für (15).

	$\varepsilon = 10^{-3}$		$\varepsilon = 10^{-5}$	
n	ACA	Tucker	ACA	Tucker
100	$1.85 \cdot 10^{-2}$	$1.31 \cdot 10^{-3}$	$4.61 \cdot 10^{-2}$	$2.93 \cdot 10^{-3}$
200	$5.13 \cdot 10^{-3}$	$2.64 \cdot 10^{-4}$	$1.35 \cdot 10^{-2}$	$6.00 \cdot 10^{-4}$
300	$2.28 \cdot 10^{-3}$	$1.27 \cdot 10^{-4}$	$5.79 \cdot 10^{-3}$	$2.57 \cdot 10^{-4}$
400	$1.48 \cdot 10^{-3}$	$6.76 \cdot 10^{-5}$	$4.06 \cdot 10^{-3}$	$1.46 \cdot 10^{-4}$
500	$1.04 \cdot 10^{-3}$	$4.18 \cdot 10^{-5}$	$2.48 \cdot 10^{-3}$	$8.70 \cdot 10^{-5}$
600	$7.92 \cdot 10^{-4}$	$3.24 \cdot 10^{-5}$	$2.61 \cdot 10^{-3}$	$6.34 \cdot 10^{-5}$
700	$5.37 \cdot 10^{-4}$	$2.33 \cdot 10^{-5}$	$2.03 \cdot 10^{-3}$	$4.46 \cdot 10^{-5}$
800	$4.61 \cdot 10^{-4}$	$1.76 \cdot 10^{-5}$	$1.50 \cdot 10^{-3}$	$3.30 \cdot 10^{-5}$
900	$3.42 \cdot 10^{-4}$	$1.37 \cdot 10^{-5}$	$1.31 \cdot 10^{-3}$	$2.77 \cdot 10^{-5}$
1 000	$2.69 \cdot 10^{-4}$	$1.10 \cdot 10^{-5}$	$1.13 \cdot 10^{-3}$	$2.19 \cdot 10^{-5}$

Tabelle 18: Vergleich der Kompressionsraten κ von ACA, Tucker für (16).

jedoch hat er eine bessere Kompressionsrate für die betrachteten Genauigkeiten. Der ACA Algorithmus ist sowohl in Bezug auf Laufzeit als auch Speicherplatz gut für die Approximation von trivariaten Funktionen geeignet.

6. Fazit

Die drei betrachteten Verfahren zeigen verschiedene Stärken und Schwächen.
Es ist leicht zu erkennen, dass der ACA Algorithmus den beiden älteren Algorithmen durch seinen linearen Aufwand weit überlegen ist. Nachteil ist jedoch, dass seine Funktionalität bei zufällig erzeugten Datenstrukturen nicht erwiesen ist.

Der Tucker Algorithmus dagegen liefert sowohl für Funktionen als auch für die Videos gute Näherungen. Sein Vorteil besteht darin, dass seine Laufzeit nur von der Größe des gegebenen Tensors abhängt und nahezu unabhängig von den Einträgen des Tensors ist. Lässt sich jedoch gerade bei den symmetrischen Funktionen erheblich die Symmetrie ausnutzen, so dass Speicherplatz gespart werden kann, ist bei Bildern für möglichst genaue Ergebnisse der Speicher gerade der Nachteil.

Der ALS Algorithmus zeigt bei der Anwendung auf die trivariaten Funktionen Schwächen. Es dauert oftmals sehr lange, bis die Iteration konvergiert und dabei ist es auch stark abhängig von dem gegebenen Tensor und den Startwerten, ob überhaupt ein globales Minimum des Problems gefunden werden kann.
Für Bilder und Videos scheint dieses Problem nicht ganz so ausgeprägt, da mit einer relativ guten Kompressionsrate approximiert werden kann. Dies erklärt wohl auch die Anwendung des ALS Algorithmus in den angewandten Naturwissenschaften und in der Statistik, wo man mit einer Zerlegung gegebener Strukturen Daten analysieren will.

Vergleicht man die Kompression eines Videos durch die Tucker-Zerlegung oder durch den ALS Algorithmus mit herkömmlichen Methoden zur Videokompression, so ergeben sich deutliche Unterschiede. Nach [13] liegen die Kompressionsraten bei herkömmlichen Verfahren zwischen 1 : 2 und 1 : 16, also zwischen 0.5 und 0.0625. Die mit Tucker Algorithmus und ALS Algorithmus erreichten Kompressionsraten von 0.01 bei einem Fehler von 1% sind deutlich kleiner. Ihr Nachteil besteht aber darin, dass die Berechnung des Videos aus den komprimierten Daten viel zu lange dauert.

7. Literatur

[1] *BLAS*, erhältlich online bei http://www.netlib.org/blas/.

[2] *Cimg*, erhältlich online bei http://cimg.sourceforge.net/.

[3] *LAPACK*, erhältlich online bei http://www.netlib.org/lapack/.

[4] M. Bebendorf, *Adaptive Cross Approximation of Multvariate Functions*, erscheint in Constructive Approximation (2009).

[5] R. Bro, *Multi-way Analysis in the Food Industry: Models, Algorithms, and Applications*, Ph.D thesis, University of Amsterdam, 1998, erhältlich online bei http://www.models.kvl.dk/research/theses.

[6] R. Bro and C. A. Andersson, *Improving the speed of multi-way algorithm: Part I: Tucker3*, Chemometrics Intell. Lab. **42** (1998), pp. 93–103.

[7] J. D. Carroll and J. J. Chang, *Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition*, Psychometrika **35** (1970), pp. 283–319.

[8] L. de Lathauwer, B. de Moor, and J. Vandewalle, *A Multilinear Singular Value Decomposition*, SIAM Journal on Matrix Analysis and Applications **21** (2000), no. 4, pp. 1253–1278.

[9] G. H. Golub and C. F. van Loan, *Matrix Computations*, 3 ed., The Johns Hopkins University Press, 1996.

[10] T. G. Kolda, *Multilinear Operators for Higher-Order Decompositions*, Tech. Report SAND2006-2081, Sandia National Laboratories, April 2006.

[11] T. G. Kolda and B. W. Bader, *Tensor Decompositions and Applications*, SIAM Review **51** (2009), no. 3, pp. 455–500.

[12] L. R. Tucker, *Some mathematical notes on three-mode factor analysis*, Psychometrika **31** (1966), pp. 279–311.

[13] Film und Videoclub Ahrweiler, *Video-Kompression*, http://www.videoclub-ahrweiler.de/2kompres.htm, 2009.

A. Anhang

A.1. Quellcode zur Tucker-Zerlegung

Der Algorithmus zur Tucker-Zerlegung wurde in C++ programmiert. Eine Matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ wird in einem Array der Länge $m \cdot n$ spaltenweise abgespeichert. Dabei entspricht die *leading dimension* `lda` der Zeilenanzahl m.
Im folgenden wird die Variante für symmetrische Funktionen beschrieben, da diese kürzer und einfacher verständlich ist. Dabei werden zunächst die externen Funktionen aus LAPACK [3] definiert.

Die Funktion `dseyvr` berechnet ausgewählte Eigenwerte und optional Eigenvektoren einer symmetrischen reellen Matrix $\boldsymbol{A}$. Das Argument `jobz` gibt an, ob nur Eigenwerte (`jobz='N'`) oder auch Eigenvektoren (`jobz='V'`) berechnet werden sollen. `range` gibt an, welche Eigenwerte bestimmt werden sollen. Bei `range='A'` werden alle Eigenwerte der Matrix berechnet, bei `range='V'` werden Eigenwerte aus einem halboffenen Intervall `(vl,vu]` ermittelt und bei `range='I'` werden die `il` bis `iu`-ten Eigenwerte berechnet. In dem Array `w` werden die gewünschten Eigenwerte in aufsteigender Reihenfolge abgespeichert und in das Array `z` werden die zugehörigen Eigenvektoren spaltenweise eingetragen.

Als nächstes wird die Matrixmultiplikation `dgemm` aus der BLAS-Bibliothek [1] definiert. Damit können zwei Matrizen $\boldsymbol{A}$ und $\boldsymbol{B}$ miteinander multipliziert werden. Das Ergebnis dieser Multiplikation wird im Array `c` gespeichert. Die Argumente `transa` und `transb` geben an, ob die Matrizen $\boldsymbol{A}$ und $\boldsymbol{B}$ in transponierter Form (`transa='T', transb='T'`) vorliegen oder nicht (`transa='N', transb='N'`).

Die BLAS-Bibliothek stellt auch eine Funktion `dnrm2` zur Berechnung der euklidischen Norm eines Vektors zur Verfügung. Da die einzelnen Matrizen spaltenweise als ein langer Vektor abgespeichert werden, entspricht die euklidische Norm dieses Vektors der Frobeniusnorm der zugehörigen Matrix.

Die Funktion `dcopy` ermöglicht es die Werte eines Vektors `x` in einen Vektor `y` zu kopieren.

Außerdem werden noch Funktionen benötigt, welche die Transformation eines Tensors in eine Matrix in eine andere Transformation umsortiert. Die Funktion `rearrange_12` sortiert $\boldsymbol{X}_{(1)}$ nach $\boldsymbol{X}_{(2)}$, `rearrange_23` sortiert $\boldsymbol{X}_{(2)}$ nach $\boldsymbol{X}_{(3)}$ und `rearrange_31` sortiert $\boldsymbol{X}_{(3)}$ nach $\boldsymbol{X}_{(1)}$. Dabei werden mit Hilfe von `dcopy` immer einzelne Vektoren kopiert.

```cpp
// Array von (1) nach (2) umsortieren
void rearrange_12(int m, int n, int l, double* x, double* y)
{
    for (int k=0; k<l; k++)
    {
        for (int j=0; j<n; j++)
        {
```

```
            double *p = &x[m*(j+k*n)];
            double *neu = &y[j+k*n*m];
            int incx = 1;
            int incy = n;
            dcopy_(&m, p, &incx, neu, &incy);
        }
    }
}

void rearrange_23(int m, int n, int l, double* x, double* y)
{
    for (int k=0; k<l; k++)
    {
        for (int i=0; i<m; i++)
        {
            double *p = &x[n*(i+k*m)];
            double *neu = &y[i*l+k];
            int incx = 1;
            int incy = m*l;
            dcopy_(&n, p, &incx, neu, &incy);
        }
    }
}

void rearrange_31(int m, int n, int l, double* x, double* y)
{
    for (int j=0; j<n; j++)
    {
        for (int i=0; i<m; i++)
        {
            double *p = &x[l*(i+j*m)];
            double *neu = &y[i+m*j];
            int incx = 1;
            int incy = m*n;
            dcopy_(&l, p, &incx, neu, &incy);
        }
    }
}
```

Nun wird eine selbstgeschriebene Funktion `nmbr_ev` definiert, die zu einer vorgegebenen Fehlertoleranz `eps` die Anzahl der benötigen Eigenwerte berechnet. Hierzu wird zuerst die Summe aller Eigenwerte berechnet und dann nach und nach ein Eigenwert hinzugenommen und überprüft, ob die Fehlertoleranz eingehalten wird. Da bei der Tucker-Zerlegung die ersten Eigenwerte groß sind und dann stark abfallen, wird mit einem Eigenwert gestartet und die anderen hinzugenommen. Würde man mit allen Eigenwerten starten und Schritt für Schritt einen weglassen, so würde dies länger dauern, da meistens nur wenige Eigenwerte benötigt werden.

```
int nmbr_ev(double* w, int n, double eps)
{
    int anzahl   = 1;
```

```
    double teil  = w[n-1];
    double summe = 0;

    for (int i=0; i<n; i++)
    summe = summe + w[i];

    while (teil < eps*summe)
    {
        anzahl++;
        teil = teil + w[n-anzahl];
    }
    return anzahl;
}
```

Für die eigentliche Berechnung der Tucker-Zerlegung wird die Funktion `matrix_M` benötigt. Sie berechnet zuerst die Matrixmultiplikation $\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^T$ mit `dgemm` und dann alle Eigenwerte dieser Matrix mit `dsyevr`. Danach wird mit `nmbr_ev` die Anzahl der benötigen Eigenwerte berechnet und hiernach werden mit `dsyevr` die zugehörigen Eigenvektoren bestimmt. Diese bilden dann die Matrix $\boldsymbol{A}$.

```
int matrix_M(int m, int n, int l, double* x, double* &z)
{
    // Lokale Variablen und Arrays definieren
    int zeilen  = m;
    int spalten = n*l;
    int il      = 0;
    int iu      = 0;
    double* w   = new double[zeilen];

    // Array X*XT berechnen
    double* a1 = new double[zeilen*zeilen];
    double* a2 = new double[zeilen*zeilen];
    dgemm('N', 'T', zeilen, zeilen, spalten, x, x, a1);
    dcopy(zeilen*zeilen, a1, a2);

    // alle Eigenwerte bestimmen
    dsyevr('N', 'A', zeilen, a1, il, iu, w, z);

    // Anzahl der benötigten Eigenvektoren berechnen
    int anzahl = nmbr_ev(w, zeilen, 0.9999);
    cout<<"Anzahl "<<anzahl<<endl;

    // Nur gewünschte Eigenvektoren berechnen
    iu  = zeilen;
    il  = zeilen-anzahl+1;
    z   = new double[anzahl*zeilen];
    dsyevr('V', 'I', zeilen, a2, il, iu, w, z);

    // Speicher freigeben
    delete[] a1; a1 = NULL;
    delete[] a2; a2 = NULL;
    delete[] w;  w  = NULL;
```

```

    return anzahl;
}
```

Die folgende Funktion `calculate_G` berechnet aus den drei Faktormatrizen $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$ den Kerntensor $\mathcal{G}$ mit (12).

```
void calculate_G(int I, int J, int K, int M, int P, int Q,
     double* a, double* b, double* c, double* x, double* g)
{
    double *zwischen1 = new double[I*J*K];
    double *zwischen2 = new double[I*J*K];

    // A^T * X
    dgemm('T', 'N', M, J*K, I, a, x, zwischen1);
    rearrange_12(M, J, K, zwischen1, zwischen2);

    // B^T * X
    dgemm('T', 'N', P, I*K, J, b, zwischen2, zwischen1);
    rearrange_23(M, P, K, zwischen1, zwischen2);

    // C^T * X = G
    dgemm('T', 'N', Q, P*M, K, c, zwischen2, zwischen1);
    rearrange_31(M, P, Q, zwischen1, g);

    // Speicher freigeben
    delete[] zwischen1; zwischen1 = NULL;
    delete[] zwischen2; zwischen2 = NULL;
}
```

Als letztes folgt die `main`-Funktion. Mit der ersten `for`-Schleife wird die gewünschte Funktion an den vorgegeben Stützstellen ausgewertet und die Funktionswerte werden in die Matrixversion $\boldsymbol{X}_{(1)}$ geschrieben. Danach müssten alle drei Faktormatrizen $\boldsymbol{A}$, $\boldsymbol{B}$ und $\boldsymbol{C}$ berechnet werden, da aber $I = J = K$ gilt und die getestete Funktion (15) symmetrisch ist, folgt $\boldsymbol{A} = \boldsymbol{B} = \boldsymbol{C}$. Deswegen muss nur die Matrix $\boldsymbol{A}$ berechnet werden. Im Schritt danach wird der Kerntensor $\mathcal{G}$ bestimmt. Zum Schluss wird noch der Fehler `eps` und die Kompressionsrate `kappa` wie in Kapitel 3.8 ermittelt.

```
int main() {
    clock_t start, end;
    start = clock();

    // Größe n
    int I = 350;
    int J = 350;
    int K = 350;

    // Matrix initialisieren
    double *x = new double[I*J*K];
    for (int j=0; j<J; j++)
    {
```

```
        for (int k=0; k<K; k++)
        {
            for (int i=0; i<I; i++)
            {
                double X = (1.0/I)*(i+0.5);
                double Y = (1.0/J)*(j+0.5);
                double Z = (1.0/K)*(k+0.5);
                x[i+I*(k+K*j)]=1/sqrt(1+X*X+Y*Y+Z*Z);
            }
        }
    }

    // Faktormatrizen berechnen
    double *a;
    int M = matrix_M(I, J, K, x, a);

    // G berechnen
    double* g = new double[M*M*M];
    calculate_G(I, J, K, M, M, M, a, a, a, x, g);

    // Fehler bestimmen
    double normX = dnrm2(I*J*K, x);
    double normG = dnrm2(M*M*M, g);
    double eps   = sqrt(abs(normX*normX-normG*normG))/normX;
    double kappa = (double) (I*M+M*M*M)/(I*J*K);
    cout<<"Fehler: "<<eps<<endl;
    cout<<"Größe: "<<kappa<<endl;

    // Speicher freigeben
    delete[] x; x = NULL;
    delete[] a; a = NULL;
    delete[] g; g = NULL;

    end = clock();
    cout << (double)(end-start)/ CLOCKS_PER_SEC<< ’s’<< endl;

    return 0;
}
```

A.2. Quellcode zum ALS-Algorithmus

Der Alternating Least Squares Algorithmus wurde für die Approximation eines dreidimensionalen Tensors $\mathcal{X} \in \mathbb{R}^{n_0\times n_1\times n_2}$ in C++ programmiert.
Wie beim Tucker Algorithmus aus A.1. wird der Tensor in einem Array der Länge $n_0 \cdot n_1 \cdot n_2$ dargestellt, wobei die Reihenfolge der Einträge der spaltenweisen abgespeicherten Matrixtransformation $\boldsymbol{X}_{(1)}$ des Tensors entspricht.
Im Folgenden wird die Funktion (15) approximiert. In diesem Fall sind die Darstellungen $\boldsymbol{X}_{(1)}, \boldsymbol{X}_{(2)}, \boldsymbol{X}_{(3)}$ identisch und so werden die Funktionen zur Umsortierung des Tensors nicht benötigt.
Die Startwerte für $\boldsymbol{B}$, $\boldsymbol{C}$ werden mithilfe der Tucker Zerlegung bestimmt und als Abbruchkriterium wird eine maximal vorgegebene Anzahl an Iterationen gewählt.
Es werden die BLAS Funktionen `dnrm2_` zum Berechnen der 2-Norm, `dgemm_` für die Matrixmultiplikation zweier Matrizen und `dcopy_` und `dscal_` zum Kopieren und Skalieren von Vektoren benutzt.

Mithilfe der LAPACK Routine `dgelss_` wird die Pseudoinverse bestimmt. Hierbei berechnet `dgelss_` eine Least Squares Lösung für das Problem $\boldsymbol{V}\boldsymbol{X} = \boldsymbol{I}$ mithilfe der Singulärwertzerlegung und ist somit auch für gegebenfalls singuläre Matrizen (wie sie bei der Initialisierung mit Zufallszahlen auftreten können) anwendbar.
Die letztendliche Berechnung folgt in der Funktion `pseudoinverse(v,g,r)`, die die Pseudoinverse `g` einer `r`× `r` -Matrix `v` zurückgibt.

```
//Berechnung der rxr-Pseudoinversen der Matrix v
void pseudoinverse(double*v, double*g, int r)
{
    //Hilfsvariablen für die LAPACK-Routine
    int info,rank, square_r=r*r, inc=1,lwork=6*r;
    double rcond=-1;
    double*work=new double[lwork];
    double*s=new double[r];
    double*one=new double[square_r];

    //Aufstellen der rxr-Einheitsmatrix
    for(int k=0;k<r;k++)
    {
        for(int i=0;i<r;i++)
        {
            if (k==i) one[k*r+i]=1.;
            else one[k*r+i]=0.;
        }
    }
    //Pseudoinverse wird berechnet und in one gespeichert
    dgelss_(&r, &r, &r, v, &r, one, &r, s, &rcond, &rank,
       work, &lwork, &info);

    //Pseudoinverse wird nach g kopiert
    dcopy_(&square_r,one,&inc,g,&inc);

    delete[] s;
```

```
    delete[] one;
    delete[] work;
    return;
}
```

Für die Iterationsstartwerte wurden die Funktionen `dgemm`, `rearrange_23` und `matrix_M` aus dem Tucker-Algorithmus übernommen und eine zusätzliche Funktion geschrieben, die den Startwert für $\boldsymbol{C}$ berechnet:

```
void startwert_C(double*b,double* &c,double*x, int r, int n0, int n1, int n2)
{
    double* ergebnis=new double[r*n2*n0];
    double* ergebnis2=new double[r*n2*n0];

    //Berechne b^T*x
    dgemm('T','N',r,n0*n2,n1,b,x,ergebnis);

    //sortiere um
    rearrange_23(n0,r,n2,ergebnis,ergebnis2);

    //Berechne die r ersten linken Singulärvektoren von (b^T*x)_(3)
    matrix_M(n0,r,n2,ergebnis2,c,r);

    //Speicher freigeben
    delete[] ergebnis;
    delete[] ergebnis2;
    return;
}
```

Als Alternative ist noch die Initialisierung mit Zufallszahlen gegeben:

```
void randommatrix(double*x, int n)
{
    for(int i=0;i<n;i++)
    {
        x[i]=(double)rand()/RAND_MAX;
    }
    return;
}
```

Es folgen die wesentlichen Hilfsfunktionen für den Algorithmus:

```
//Berechnung des Hadamard-Produkts (elementweise Multiplikation) zweier rxr-Matrizen a,b mit rxr-Matrix ergebnis als Resultat
void hadamard (double*a, double*b, double*ergebnis, int r)
{
    for(int k=0;k<r;k++)
    {
    for(int i=0;i<r;i++)
```

```
    {
        ergebnis[k*r+i]=a[k*r+i]*b[k*r+i];
    }
    }
    return;
}

//Berechnung des Khatri-Rao-Produkts von nxr-Matrix a, mxr-Matrix b und (n*m)xr-Matrix ergebnis als Resultat
void khatrirao (double*a, double*b, double*ergebnis, int n, int m, int r)
{
    for(int k=0;k<r;k++)
    {
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
        ergebnis[n*m*k+i*m+j]=a[k*n+i]*b[k*m+j];
        }
    }
    }
    return;
}

//Normalisierung der Spalten der mxr-Matrix a, Längen werden im r-dim. Vektor lambda gespeichert
void normalization (double*a, double*lambda, int m, int r)
{
    double*s;
    double alpha;
    int incx=1;

    for(int k=0;k<r;k++)
    {
        s=&a[m*k];
        lambda[k]=dnrm2_(&m,s,&incx);
        if(lambda[k]>0.00000000001)
        {
            alpha=1./lambda[k];
            dscal_(&m,&alpha,s,&incx);
        }
        else cout<<"Stop!"<<endl;
    }
    return;
}
```

In `calculate` erfolgt die Berechnung einer neuen Faktormatrix `null` für zwei fixierte andere Richtungen `eins` und `zwei`:

```
//Ausführen des Algorithmus
void calculate(double*x, double*null, double*eins, double*
```

```
    zwei, double*lambda, int r, int n0, int n1, int n2)
{
    //Hilfsvariablen
    double*c1=new double[r*r];
    double*c2=new double[r*r];
    double*ergebnis=new double[n1*n2*r];
    double*h=new double[n1*n2*r];
    char normal='N'; char transp='T';
    double alpha=1., beta=0.;
    int n=n1*n2;

    //eins^T*eins und zwei^T*zwei berechnen und in c1 und c2
       abspeichern
    dgemm_(&transp, &normal, &r, &r, &n1, &alpha, eins, &n1,
       eins, &n1, &beta, c1, &r);
    dgemm_(&transp, &normal, &r, &r, &n2, &alpha, zwei, &n2,
       zwei, &n2, &beta, c2, &r);

    hadamard(c2,c1,c1,r);
    pseudoinverse(c1,c2,r);

    //Khatri-Rao Produkt von eins und zwei, der Vektor
       ergebnis ist das Resultat
    khatrirao(eins,zwei,ergebnis,n1,n2,r);

    //Matrixmultiplikation x*ergebnis*c2 ergibt die neue
       Matrix null
    dgemm_(&normal, &normal, &n, &r, &r, &alpha, ergebnis, &n
       , c2, &r, &beta, h, &n);
    dgemm_(&normal, &normal, &n0, &r, &n, &alpha, x, &n0, h,
       &n, &beta, null, &n0);

    //Normalisierung der Spalten von null
    normalization(null,lambda,n0,r);

    //Speicher freigeben
    delete[] c1;
    delete[] c2;
    delete[] ergebnis;
    delete[] h;
    return;
}
```

Die eigentliche Iteration wird in `ALS` ausgeführt. Dabei sind `x`, `r`, `a`, `b`, `c` und die gewünschte Anzahl an Iterationen `iterationen` vorgegeben und es werden `a0`, `a1`, `a2` und `lambda` berechnet:

```
//Alternating Least Squares - Algorithmus
void ALS (double*x, double* &a0, double* &a1, double* &a2,
    double* &lambda, int r, int a, int b, int c, int
    iterationen)
{
```

```
    //Startwerte
    matrix_M(a,b,c,x,a1,r);
    startwert_C(a1,a2,x,r,a,b,c);
    //oder
    //randommatrix(a1,n1*r); randommatrix(a2,n2*r);

    //Iteration
    for(int k=0;k<iterationen;k++)
    {
        calculate(x,a0,a2,a1,lambda,r,a,c,b);
        calculate(x0,a1,a2,a0,lambda,r,b,c,a);
        calculate(x1,a2,a1,a0,lambda,r,c,b,a);
    }
    return;
}
```

Um die Norm des Fehlers zu bestimmen, muss der approximative Tensor wieder zusammengesetzt werden; dabei gibt die Funktion `composition` die berechnete Fehlernorm zurück:

```
double composition(double* tensor,double*a0, double*a1,
   double*a2, double*lambda,int r, int n0, int n1, int n2)
{
    double norm;

    //Hilfsvariablen für die Matrixmultiplikation mit BLAS
    double*ergebnis=new double[n1*n2*r];
    double*neu=new double[n0*n1*n2];
    char nein='N'; char transp='T';
    int incx=1,dim=n0*n1*n2;
    int g=n1*n2;
    double alpha=1., beta=0.;
    double lamb;
    double*t;

    khatrirao(a1,a2, ergebnis, n1, n2, r);

    //die Normalisierung wird rückgängig gemacht
    for(int l=0;l<r;l++)
    {
        lamb=lambda[l];
        t=&ergebnis[l*g];
        dscal_(&g,&lamb,t,&incx);
    }

    //a0*ergebnis^T
    dgemm_(&nein,&transp,&n0,&g,&r,&alpha,a0,&n0,ergebnis,&g
       ,&beta,neu,&n0);

    int hilf=0;
    //Differenz zwischen Tensor und Approximation in der
       richtigen Sortierung
```

```
    for(int k=0;k<n0;k++)
    {
        for(int j=0;j<(n2*n1);j++)
        {
            tensor[hilf]=tensor[hilf]-neu[k+j*n0];
            hilf++;
        }
    }
    //Bestimmung der Norm
    norm=dnrm2_(&dim,tensor,&incx);

    //Speicher freigeben
    delete[] ergebnis;
    delete[] neu;
    return norm;
}
```

Der Algorithmus wird auf Funktion (15) angewandt:

```
void function(double*x, int n0, int n1, int n2)
{
    double X,Y,Z;
    for (int j=0; j<n1; j++)
    {
        for (int k=0; k<n2; k++)
        {
            for (int i=0; i<n0; i++)
            {
                X=(1.0/n0)*(i+0.5);
                Y=(1.0/n1)*(j+0.5);
                Z=(1.0/n2)*(k+0.5);
                x[i+n0*(k+n2*j)]=1./sqrt(1.+X*X+Y*Y+Z*Z);
            }
        }
    }
    return;
}
```

Schließlich das Hauptprogramm:

```
int main(int argc, char *argv[])
{
    //srand(time(NULL)); für Zufallszahlen
    //Eingabe der Parameter
    int n0=200; int n1=200; int n2=200; int r=2; int iter
       =100;

    //Die benoetigten Arrays
    double*original=new double[n0*n1*n2];
    double*a0=new double[n0*r];
    double*a1=new double[n1*r];
    double*a2=new double[n2*r];
```

```
    double*lambda=new double[r];

    //Hilfsvariablen
    int dim=n0*n1*n2,incx=1;
    double norm1, norm2;

    //Initialisierung des Tensors
    function(original,n0,n1,n2);

    //Zeit messen
    clock_t start,end;
    start=clock();

    //Ausführen des Algorithmus
    ALS(original,a0,a1,a2,lambda,r,n0,n1,n2,iter);

    //Zeit stoppen
    end=clock();

    //wieder zusammensetzen und Fehlernorm bestimmen
    norm1=dnrm2_(&dim,original,&incx);
    norm2=composition(original,a2,a1,a0,lambda,r,n2,n1,n0);

    //Ausgabe
    cout<<endl<<"Algorithmus-Zeit: "<<(double)(end-start)/
       CLOCKS_PER_SEC<<endl;
    cout<<endl<<"Fehler-Toleranz: "<<norm2/norm1<<endl;
    cout<<endl<<"Kompressionsrate: "<<(double)(n0+n1+n2)*r/(
       n0*n1*n2)<<endl;

    //Speicher freigeben
    delete[] original;
    delete[] a0;
    delete[] a1;
    delete[] a2;
    delete[] lambda;
    return 0;
}
```

A.3. CD mit Videos

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich von der vorliegenden Bachelorarbeit Unterkapitel 2.5 (bis auf Zeilen 4 - 13), 2.6, Kapitel 1, 4, 5 (bis auf die Tabellen), 6 (bis auf Zeilen 6 - 7 und 19 - 25), sowie Anhang A.2 selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Bonn, den 1. September 2010

Katrin Almon

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich von der vorliegenden Bachelorarbeit Unterkapitel 2.1, 2.2, 2.3, 2.4, 2.5 (nur Zeilen 4 - 13), Kapitel 3, Kapitel 5 (nur die Tabellen), Kapitel 6 (nur Zeilen 6 -7 und 19 - 25), sowie Anhang A.1 selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Rheinbreitbach, den 1. September 2010

Barbara Fuchs