Low-Rank-Tensor-Techniken im Jacobi-Davidson-Eigenwertlöser

Masterarbeit

vorgelegt von
Rebekka-Sarah Hennig
am 24. Februar 2021

am Mathematischen Institut der Universität zu Köln

Betreuer:
Prof. Dr. Axel Klawonn
Dr. Martin Lanser

externe Betreuer:
Dr. Jonas Thies
Melven Röhrig-Zöllner

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Simulation und Analyse hoch-dimensionaler Probleme ist eine numerische Herausforderung. Die hierbei auftretenden Tensoren (d-dimensionale Arrays) sind mit dem Fluch der Dimensionalität behaftet. Das bedeutet, dass die Anzahl an Elementen eines Tensors, und somit auch der Speicherverbrauch und die Anzahl an Rechenoperationen, exponentiell mit der Anzahl an Dimensionen d wächst. Hierdurch können numerische Standardmethoden bei zu großer Dimensionalität nicht mehr verwendet werden.

Ein Beispiel für ein solches Problem kommt aus der Quantenmechanik [1, Kapitel 6 und 7]. Hier werden Zustandsänderungen einer Kette von Elektronenspins analysiert. Die einzelnen Spins können die Zustände Spin-Up $\uparrow$ oder Spin-Down $\downarrow$ annehmen. Für die gesamte Kette mit L Spins gibt es somit 2^L mögliche Zustände. Die Suche nach dem Spinkettenzustand mit dem niedrigsten Energieniveau führt zu einem hoch-dimensionalen linearen Eigenwertproblem, welches für große L schwierig zu berechnen ist.

Um den exponentiell großen Aufwand in solchen Problemen zu vermeiden, werden Low-Rank-Tensor-Methoden verwendet. Hierbei werden die hoch-dimensionalen Tensoren in ein Produkt aus Tensoren von geringerer Dimension zerlegt. Wird die innere Dimension des Produkts, genannt Bonddimension oder Rang der Zerlegung, nicht zu groß, kann man hierdurch die Anzahl an Elementen reduzieren. Eine Übersicht über verschiedene Formate solch einer Tensorzerlegung findet sich in [2, Kapitel 3].

Mit den zerlegten Tensoren kann man übliche Rechenoperationen wie Additionen, Multiplikationen mit einem Skalar oder Skalarprodukte durchführen. Zu beachten ist, dass die Bonddimensionen durch Operationen wie beispielsweise Additionen wachsen [3]. Um die Anzahl an Elementen klein zu halten, muss hier approximiert werden, wodurch zusätzliche Ungenauigkeit bei der Berechnung hinzukommt. Während bei der Gleitkommaarithmetik die einzelnen Elemente gerundet werden, wird hier ein Genauigkeitskriterium für den Tensor als Ganzes erfüllt. Auch wenn mit den Tensorzerlegungen alle Operationen ausgeführt werden können, welche für die üblichen Algorithmen aus der dünnbesetzten linearen Algebra benötigt werden, ist hierdurch unklar, welche Auswirkungen die Verwendung der anderen Darstellung in diesen Algorithmen hat. Es stellt sich die Frage, ob ein gegebenes Verfahren weiterhin robust funktioniert, wenn alle Rechenoperationen mit Tensorzerlegungen approximiert werden.

Es gibt einige Arbeiten, die sich mit der Verwendung von Low-Rank-Tensorzerlegungen in iterativen Lösern beschäftigen. Eine Übersicht kann in [4, Kapitel 3.1] gefunden werden.

In dieser Arbeit werden Low-Rank-Tensor-Techniken als Vorkonditionierer für iterative Verfahren von dünnbesetzten Matrizen betrachtet. Als Testverfahren wählen wir den Jacobi-Davidson-Eigenwertlöser [5]. Dies ist eine Projektionsmethode, welche ihren Suchraum durch das Lösen eines Gleichungssystems erweitert. Als Gleichungslöser wol-

len wir hier das TT-GMRES-Verfahren [6] nutzen, welches das Tensor-Train-Format im GMRES-Algorithmus verwendet.

Ziel ist es, unsere Tensor-Train-Jacobi-Davidson-Variante mit dem herkömmlichen Jacobi-Davidson-Verfahren zu vergleichen, wobei der Fokus auf dem numerischen Verhalten des Verfahrens liegt. Wir stellen uns hierbei die Frage, ob wir aus dem TT-Format einen Nutzen ziehen können, oder ob es besser ist, die herkömmliche Methode zu verwenden. Hierzu betrachten wir verschiedene hoch-dimensionale lineare Eigenwertprobleme. Für unsere Implementation nutzen wir das Python-Paket ttpy [7], welches die nötigen Operationen im Tensor-Train-Format enthält.

Zusätzlich wollen wir unsere Tensor-Train-Jacobi-Davidson-Methode mit einem auf dem alternierenden linearen Schema (ALS) [8] basierenden Algorithmus vergleichen. Anstatt das Tensor-Train-Format in übliche iterative Verfahren einzubauen, wird bei der ALS-Methode das Problem als Optimierungsproblem im Tensor-Train-Format umformuliert und so die Struktur des Tensor-Train-Formats bei der Berechnung ausgenutzt. Für den Vergleich nutzen wir den eigb-Algorithmus [9] aus ttpy, welcher auf dem ALS-Verfahren basiert.

Diese Arbeit ist wie folgt aufgebaut:

In Kapitel 2 geben wir eine kleine Einführung in das Thema der Tensorzerlegungen und definieren wichtige Eigenschaften und Operationen von Tensoren und dem Tensor-Train-Format, welche im späteren Verlauf der Arbeit benötigt werden. Kapitel 3 beschäftigt sich mit dem Thema Eigenwertlöser. Hier werden sowohl der Jacobi-Davidson-Algorithmus als auch das ALS-Verfahren erklärt. Unsere Kombination von Tensor-Train-Format und Jacobi-Davidson-Algorithmus wird in Kapitel 4 erläutert. Kapitel 5 schätzt die Komplexität von Operationen und den Speicherverbrauch im Tensor-Train Format ab und vergleicht sie mit der Komplexität bei der Verwendung von voll- beziehungsweise dünnbesetzten Matrizen und Vektoren. In Kapitel 6 kommen wir dann zu den numerischen Experimenten des Tensor-Train-Jacobi-Davidson-Algorithmus und dem Vergleich zum ALS-Verfahren. Das Fazit befindet sich in Kapitel 7.

2 Tensoren und Tensorzerlegungen

In diesem Kapitel beschäftigen wir uns mit den theoretischen Grundlagen von Tensoren und Tensorzerlegungen, welche für diese Arbeit benötigt werden. Wir verwenden in unserer Arbeit die Notation aus [2] und werden im Folgenden die für uns notwendigen Definitionen aus [2, Kapitel 2 und 3] wiedergeben. Diese sind über $\mathbb{R}$ definiert, können aber analog auf $\mathbb{C}$ übertragen werden, da die benötigten Definitionen und Eigenschaften aus der linearen Algebra auch über $\mathbb{C}$ definiert sind. Da wir in dieser Arbeit hauptsächlich mit reellen Tensoren arbeiten, bleiben wir hier bei der Formulierung über $\mathbb{R}$ aus [2].

Wir gehen im Folgenden zunächst kurz auf Tensoren ein und geben dann eine kurze Einführung in das Thema der Tensorzerlegungen. Hierbei interessieren wir uns vor allem für das Tensor-Train-Format [3].

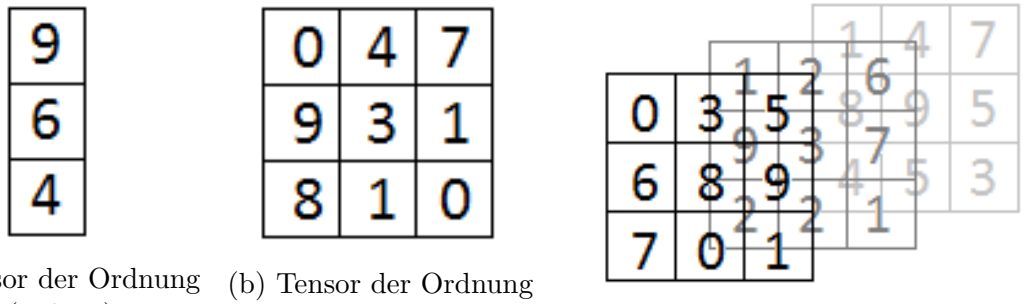
2.1 Tensoren

Unter einem Tensor verstehen wir in dieser Arbeit eine mehrdimensionale Verallgemeinerung einer Matrix. Dargestellt wird diese durch ein Array mit d Indizes:

$$\boldsymbol{T} \in \mathbb{R}^{n_1 \times \cdots \times n_d} = \mathbb{R}^N,$$

mit $n_i \in \mathbb{N}$ für $i = 1, \ldots, d$, $d \in \mathbb{N}$. Für eine kürzere Schreibweise wird die Indexmenge $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$ verwendet. Die verschiedenen Dimensionen n_i des Arrays nennen wir Moden und die Anzahl aller Moden d nennen wir die Ordnung des Tensors. Betrachten wir ein Element eines Tensors, so schreiben wir $\boldsymbol{T}_{x_1,\ldots,x_d}$ mit $1 \leq x_i \leq n_i \ \forall i$.

Damit ist ein Vektor ein Tensor der Ordnung 1 und eine Matrix ein Tensor der Ordnung 2. Einen Tensor der Ordnung 3 kann man sich wie mehrere Schichten von Matrizen

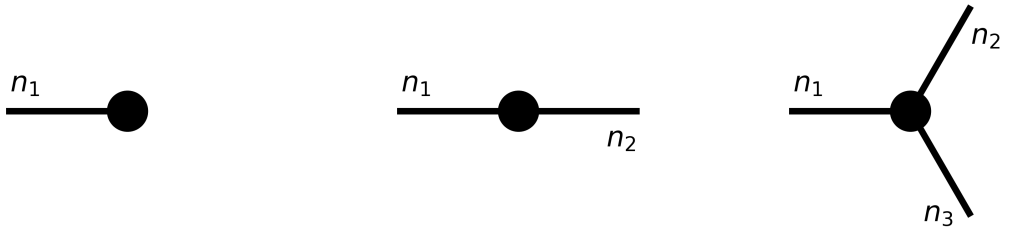


(a) Tensor der Ordnung 1 (Vektor)

(b) Tensor der Ordnung 2 (Matrix)

(c) Tensor der Ordnung 3

Abbildung 2.1: Beispiele für Tensoren



(a) Tensor der Ordnung 1 (Vektor) (b) Tensor der Ordnung 2 (Matrix) (c) Tensor der Ordnung 3

Abbildung 2.2: Graphische Darstellung von Tensoren

vorstellen oder wie einen Rubik's Cube, der in jedem kleinen Würfel eine Zahl enthält. Siehe Abbildung 2.1 für Beispiele.

Analog zu Matrizen, welche als lineare Operatoren auf Vektoren aufgefasst werden können, können wir auch hier lineare Operatoren definieren, welche auf den Tensoren operieren:

$$\boldsymbol{G} : \mathbb{R}^N \to \mathbb{R}^M, \boldsymbol{T} \to \boldsymbol{G} \cdot \boldsymbol{T}$$

mit $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$ und $M = (m_1, \ldots, m_d) \in \mathbb{N}^d$. Diese sogenannten Tensoroperatoren sind mehrdimensionale Verallgemeinerungen von Matrizen mit Paaren von Moden und werden als Tensoren in $\mathbb{R}^{M \times N} = \mathbb{R}^{(m_1 \times n_1) \times \cdots \times (m_d \times n_d)}$ betrachtet. Der Tensor $(\boldsymbol{G} \cdot \boldsymbol{T}) \in \mathbb{R}^M$ wird analog zum Matrix-Vektor-Produkt durch

$$(\boldsymbol{G} \cdot \boldsymbol{T})_{x_1,\ldots,x_d} = \sum_{y_1=1}^{n_1} \cdots \sum_{y_d=1}^{n_d} \boldsymbol{G}_{x_1,y_1,\ldots,x_d,y_d} \cdot \boldsymbol{T}_{y_1,\ldots,y_d} \tag{2.1.1}$$

definiert.

[2, Kapitel 2] gibt eine Übersicht über verschiedene Operationen für Tensoren und zeigt, dass man herkömmliche Matrix-Vektor-Operationen auf Tensoren und Tensoroperatoren übertragen kann. Ein Beispiel ist die Addition zweier Tensoren $\boldsymbol{T}$ und $\boldsymbol{U}$ des selben Tensorraums $\mathbb{R}^N$, welche analog zur Addition zweier Matrizen oder zweier Vektoren durch elementweise Addition verläuft:

$$(\boldsymbol{T} + \boldsymbol{U})_{x_1,\ldots,x_d} = \boldsymbol{T}_{x_1,\ldots,x_d} + \boldsymbol{U}_{x_1,\ldots,x_d}.$$

Ebenso wird die Multiplikation mit einem Skalar $a \in \mathbb{R}$ elementweise durchgeführt:

$$(a \cdot \boldsymbol{T})_{x_1,\ldots,x_d} = a \cdot \boldsymbol{T}_{x_1,\ldots,x_d}.$$

Die meisten anderen Operationen können als Indexkontraktionen formuliert werden, weshalb wir im Folgenden genauer auf diese Operation eingehen und zeigen, wie diese mit anderen Operationen zusammenhängt. Zusätzlich betrachten wir noch die Matrizisierung und die Vektorisierung eines Tensors, da wir diese später noch benötigen.

2.1.1 Matrizisierung und Vektorisierung

Bei der Matrizisierung und Vektorisierung ist das Ziel, den Tensor als Matrix beziehungsweise Vektor zu schreiben.

Dies wird standardmäßig bei vielen mehrdimensionalen Problemen in der Numerik verwendet. Betrachtet man beispielsweise eine partielle Differentialgleichung auf einem Würfel-Gitter in einer Finite-Differenzen-Methode, so werden hier für gewöhnlich die Gitterpunkte dimensionsweise durchnummeriert und mit dieser Reihenfolge die Werte auf dem Gitter als Vektor geschrieben. Dies entspricht der Vektorisierung des Tensors der Ordnung 3, welcher uns von dem Gitter vorgegeben wird und bei dem jede Gitterdimension für eine Mode steht.

Bevor wir zur Definition der Matrizisierung kommen, definieren wir eine Bijektion ϕ_N der Indexmenge $N = (n_1, \dots, n_d) \in \mathbb{N}^d$. Wir wählen ϕ_N so, dass die kleinstmögliche Komponente zuerst genannt wird. Das heißt, es gilt

$$\phi_N : \{1, \dots, n_1\} \times \cdots \times \{1, \dots, n_d\} \to \{1, \dots, \prod_{k=1}^{d} n_k\}$$

$$\phi_N(x_1, \dots, x_d) = \overline{x_1, \dots, x_d} = 1 + \sum_{k=1}^{d} (x_k - 1) \prod_{l=1}^{k-1} n_l.$$

Hiermit definieren wir die Matrizisierung wie folgt.

Definition 2.1.1 (Matrizisierung)**.** *Sei $N = (n_1, \dots, n_d) \in \mathbb{N}^d$ eine Indexmenge und $\boldsymbol{T} \in \mathbb{R}^N$ ein Tensor. Seien $N' = (n_{k_1}, \dots, n_{k_e})$ und $N'' = (n_{l_1}, \dots, n_{l_f})$ geordnete Teilmengen von N, sodass N die disjunkte Vereinigung von N' und N'' ist. Die Matrizisierung von $\boldsymbol{T}$ bezüglich N' und N'' ist gegeben durch*

$$\left(\boldsymbol{T}|_{N'}^{N''}\right)_{\overline{x_{k_1}, \dots, x_{k_e}}, \overline{x_{l_1}, \dots, x_{l_f}}} = \boldsymbol{T}_{x_1, \dots, x_d}.$$

Wählen wir $N'' = \emptyset$ und für N' eine Umsortierung von N, so kommen wir zum Begriff der Vektorisierung.

Definition 2.1.2 (Vektorisierung)**.** *Sei $N = (n_1, \dots, n_d) \in \mathbb{N}^d$ eine Indexmenge und $\boldsymbol{T} \in \mathbb{R}^N$ ein Tensor. Für eine Umsortierung $N' = (n_{k_1}, \dots, n_{k_d})$ von N ist die Vektorisierung von $\boldsymbol{T}$ gegeben durch*

$$\left(\boldsymbol{T}|_{N'}\right)_{\overline{x_{k_1}, \dots, x_{k_d}}} = \boldsymbol{T}_{x_1, \dots, x_d}.$$

Betrachten wir einen Tensoroperator $\boldsymbol{G} \in \mathbb{R}^{M \times N} = \mathbb{R}^{(m_1 \times n_1) \times \cdots \times (m_d \times n_d)}$, so definieren wir die natürliche Matrizisierung des Operators als $\mathrm{mat}(\boldsymbol{G}) = \boldsymbol{G}|_M^N$. Analog definieren wir die natürliche Vektorisierung eines Tensors $\boldsymbol{T} \in \mathbb{R}^N$ als $\mathrm{vek}(\boldsymbol{T}) = \boldsymbol{T}|_N$.

Da wir in dieser Arbeit sowohl mit Tensoren als auch mit ihren Matrizisierungen beziehungsweise Vektorisierungen arbeiten, nutzen wir fettgedruckte Buchstaben für die Tensoren. Hierdurch können wir klarer unterscheiden, wann wir Tensoren verwenden und wann übliche Matrizen und Vektoren beziehungsweise die Darstellungen der Tensoren als Matrizen und Vektoren.

2.1.2 Indexkontraktion und graphische Darstellung

Eine nützliche Operation in Bezug auf Tensoren ist die Indexkontraktion. Hierbei betrachten wir zwei Tensoren, welche sich einen Teil ihrer Indizes teilen:

$$\boldsymbol{T} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times p_1 \times \cdots \times p_f} \text{ und } \boldsymbol{U} \in \mathbb{R}^{n_1 \times \cdots \times n_e \times p_1 \times \cdots \times p_f}.$$

Die Reihenfolge der Indizes spielt an dieser Stelle keine Rolle.

Definition 2.1.3 (Indexkontraktion)**.** *Die Kontraktion der Moden $p_1, \ldots, p_f$ von $\boldsymbol{T}$ und $\boldsymbol{U}$ ergibt einen Tensor $\boldsymbol{V} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_e}$ mit*

$$\boldsymbol{V}_{x_1,\ldots,x_d,y_1,\ldots,y_e} = \sum_{z_1=1}^{p_1} \cdots \sum_{z_f=1}^{p_f} \boldsymbol{T}_{x_1,\ldots,x_d,z_1,\ldots,z_f} \cdot \boldsymbol{U}_{y_1,\ldots,y_e,z_1,\ldots,z_f}.$$

Eine solche Indexkontraktion schreiben wir kurz als $\boldsymbol{V} = \langle \boldsymbol{T}, \boldsymbol{U} \rangle_{p_1,\ldots,p_f}$, da es eine Generalisierung des inneren Produkts zweier Vektoren ist.

Viele Tensoroperationen entsprechen der Indexkontraktion einer speziellen Menge von zu kontrahierender Moden. Das Produkt (2.1.1), welches das Analogon zur Matrix-Vektor-Multiplikation ist, kann als Indexkontraktion der Moden N der Tensoren $\boldsymbol{G} \in \mathbb{R}^{M \times N}$ und $\boldsymbol{T} \in \mathbb{R}^{N}$ betrachtet werden:

$$(\boldsymbol{G} \cdot \boldsymbol{T}) = \langle \boldsymbol{G}, \boldsymbol{T} \rangle_N. \tag{2.1.2}$$

Ebenso kann das Skalarprodukt zweier Tensoren $\boldsymbol{T}, \boldsymbol{U} \in \mathbb{R}^{N}$ als Indexkontraktion der kompletten Indexmenge N definiert werden:

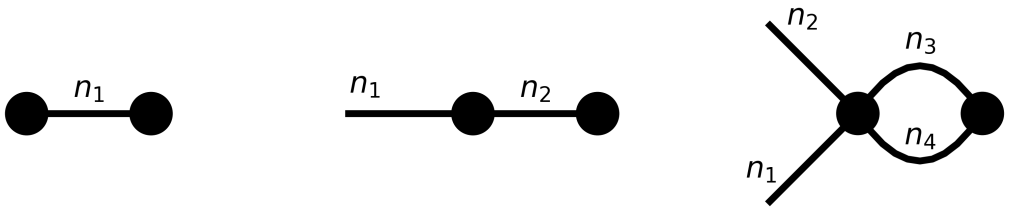
$$\operatorname{dot}(\boldsymbol{T}, \boldsymbol{U}) = \langle \boldsymbol{T}, \boldsymbol{U} \rangle_N. \tag{2.1.3}$$

Auch die Multiplikation zweier Tensoroperatoren $\boldsymbol{G} \in \mathbb{R}^{M \times N}$ und $\boldsymbol{F} \in \mathbb{R}^{N \times P}$, welche als Verallgemeinerung einer Matrix-Matrix-Multiplikation betrachtet werden kann, ist ein Spezialfall einer Indexkontraktion:

$$(\boldsymbol{G} \cdot \boldsymbol{F}) = \langle \boldsymbol{G}, \boldsymbol{F} \rangle_N.$$

Aufgrund der vielen Indizes ist die Beschreibung von Tensoroperationen über Formeln unübersichtlich. Ein hilfreiches Werkzeug zur Veranschaulichung von Operationen und Algorithmen ist die graphische Darstellung von Tensoren. Diese wurde erstmals von Penrose [10] vorgestellt und wird hier in vereinfachter Form genutzt. Hierbei wird ein Tensor $\boldsymbol{T} \in \mathbb{R}^{N}$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, der Ordnung d als ein Knoten mit d Kanten dargestellt. Die Kanten stehen für die Moden $n_1, \ldots, n_d$ des Tensors. Als Beispiel kann Abbildung 2.2 betrachtet werden, welche die graphische Darstellung der Tensoren aus Abbildung 2.1 zeigt.

Bei der Indexkontraktion zweier Tensoren werden in der graphischen Darstellung die Kanten der jeweiligen kontrahierten Moden miteinander verbunden, sodass Kanten entsteht, welche die zugehörigen Knoten verbindet. Die Moden des Ergebnistensors sind dann die nach der Operation nicht verbundenen Kanten.



(a) Skalarprodukt

(b) Matrix-Vektor-Multiplikation

(c) Kontraktion von zwei Dimensionen

Abbildung 2.3: Graphische Darstellung von Indexkontraktionen

In Abbildung 2.3 sehen wir einige Beispiele für solche Operationen. Abbildung 2.3a zeigt das Skalarprodukt $\langle v, w \rangle_{n_1}$ zweier Vektoren $v \in \mathbb{R}^{n_1}$ und $w \in \mathbb{R}^{n_1}$, Abbildung 2.3b eine Matrix-Vektor-Multiplikation $\langle A, v \rangle_{n_2}$ einer Matrix $A \in \mathbb{R}^{n_1 \times n_2}$ und eines Vektors $v \in \mathbb{R}^{n_2}$ und Abbildung 2.3c die Kontraktion $\langle \boldsymbol{U}, \boldsymbol{T} \rangle_{n_3,n_4}$ der zwei Indizes n_3 und n_4 der Tensoren $\boldsymbol{U} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ und $\boldsymbol{T} \in \mathbb{R}^{n_3 \times n_4}$. Fasst man $\boldsymbol{U}$ als Tensoroperator mit $M = \{n_1, n_2\}$ und $N = \{n_3, n_4\}$ auf, so entspricht dies der Matrix-Vektor-Multiplikation von $\mathrm{mat}(\boldsymbol{U})$ und $\mathrm{vek}(\boldsymbol{T})$.

2.2 Tensorzerlegungen

Die Anzahl der Elemente eines Tensors $\boldsymbol{T} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ kann durch $\mathcal{O}(n^d)$ abgeschätzt werden, wobei $n = \max_i n_i$. Das bedeutet, dass die Anzahl der Elemente exponentiell mit der Ordnung d wächst. Hierdurch wachsen auch der Speicherbedarf und die Komplexität exponentiell mit d. Dieser sogenannte Fluch der Dimensionalität sorgt dafür, dass Berechnungen mit Tensoren bei wachsender Dimension d zu aufwendig werden.

Dies wollen wir vermeiden, indem wir die vielen Moden eines hoch-dimensionalen Tensors mit Hilfe des Tensorprodukts auf mehrere Tensoren niedriger Ordnung verteilen.

Definition 2.2.1 (Tensorprodukt)**.** *Das Tensorprodukt $\boldsymbol{T} \otimes \boldsymbol{U} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_e}$ zweier Tensoren $\boldsymbol{T} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$ und $\boldsymbol{U} \in \mathbb{R}^{n_1 \times \cdots \times n_e}$ wird definiert durch:*

$$(\boldsymbol{T} \otimes \boldsymbol{U})_{x_1,\ldots,x_d,y_1,\ldots,y_e} = \boldsymbol{T}_{x_1,\ldots,x_d} \cdot \boldsymbol{U}_{y_1,\ldots,y_e},$$

mit $1 \leq x_i \leq m_i$ für $i = 1, \ldots, d$ und $1 \leq y_j \leq n_j$ für $j = 1, \ldots, e$.

Dies entspricht der Indexkontraktion der Tensoren $\boldsymbol{T}$ und $\boldsymbol{U}$, wenn man als Menge der zu kontrahierenden Moden die leere Menge wählt:

$$\boldsymbol{T} \otimes \boldsymbol{U} = \langle \boldsymbol{T}, \boldsymbol{U} \rangle_{\emptyset}.$$

Das einfachste Beispiel für eine Tensorzerlegung ist das dyadische Produkt. Hier werden zwei Vektoren $v \in \mathbb{R}^m$ und $w \in \mathbb{R}^n$ so miteinander multipliziert, dass eine Matrix entsteht: $v \cdot w^T = A$, $A \in \mathbb{R}^{m \times n}$. Dies entspricht dem Tensorprodukt der zwei Vektoren $v \otimes w = v \cdot w^T = A$.

Lässt sich eine Matrix als dyadisches Produkt schreiben, so können ohne Informationsverlust die zwei Vektoren statt der Matrix gespeichert werden. Hierdurch reduzieren sich die $m \cdot n$ Einträge der Matrix auf die $n + m$ Einträge der zwei Vektoren.

Diese Darstellung der Matrix ist nicht eindeutig. Skalieren wir v mit einem Wert $a \neq 0 \in \mathbb{R}$ und w mit $1/a$, so ergibt dies ebenso eine Zerlegung der Matrix. Wir können eine bis auf Vorzeichen eindeutige Zerlegung definieren, wenn wir fordern, dass $\|v\|_2 = 1$ gilt.

Das Prinzip des dyadischen Produkts lässt sich auf Tensoren beliebiger Ordnung verallgemeinern, indem für jede Dimension des Tensors ein weiterer Vektor zum Tensorprodukt hinzukommt. Diese Tensoren nennen wir Rank-One-Tensoren.

Definition 2.2.2 (Rank-One-Tensor)**.** *Ein Tensor $\boldsymbol{T} \in \mathbb{R}^N$ mit $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$ heißt Rank-One-Tensor, wenn er als Tensorprodukt von Vektoren $\boldsymbol{T}^{(i)} \in \mathbb{R}^{n_i}$, $i = 1, \ldots, d$, geschrieben werden kann:*

$$\boldsymbol{T} = \bigotimes_{i=1}^{d} \boldsymbol{T}^{(i)} = \boldsymbol{T}^{(1)} \otimes \ldots \otimes \boldsymbol{T}^{(d)}.$$

Für ein Element des Tensors gilt: $\boldsymbol{T}_{x_1,\ldots,x_d} = \prod_{i=1}^{d} \boldsymbol{T}^{(i)}_{x_i} = \boldsymbol{T}^{(1)}_{x_1} \cdot \ldots \cdot \boldsymbol{T}^{(d)}_{x_d}$.

Verwenden wir diese Schreibweise, so erhalten wir $\mathcal{O}(dn)$ Elemente statt der $\mathcal{O}(n^d)$ Elemente des Tensors, wobei $n = \max_i n_i$. Allerdings lässt sich nicht jeder Tensor als Tensorprodukt von Vektoren darstellen. Das folgende Beispiel zeigt, dass schon die Summe zweier Rank-One-Tensoren kein Rank-One-Tensor sein muss.

Beispiel 2.2.3. *Betrachte $A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \end{pmatrix}$ und $A_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \end{pmatrix}$.*
Dann gilt $A_1 + A_2 = I$.
Suche zwei Vektoren v, w, sodass $\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \cdot \begin{pmatrix} w_1 & w_2 \end{pmatrix} = \begin{pmatrix} v_1 w_1 & v_1 w_2 \\ v_2 w_1 & v_2 w_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.
$\Rightarrow v_1 w_2 = 0 \Rightarrow (v_1 = 0$ oder $w_2 = 0) \Rightarrow (v_1 w_1 = 0$ oder $v_2 w_2 = 0)$.
$\Rightarrow I$ nicht als dyadisches Produkt darstellbar (kein Rank-One-Tensor).

Um einen allgemeinen Tensor zu zerlegen reicht das einfache Tensorprodukt von Vektoren also nicht aus. Allerdings kann hierfür die Summe von Rank-One-Tensoren genutzt werden.

Lemma 2.2.4. *Jeder Tensor $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, ist als Summe von Rank-One-Tensoren darstellbar.*

Beweis. Sei $\boldsymbol{T} \in \mathbb{R}^N$ ein Tensor mit nur einem einzigen Nicht-Null-Eintrag $\boldsymbol{T}_{k_1,\ldots,k_d} = t$, $k_i \in \{1, \ldots, n_i\}$ $\forall i$, das heißt

$$\boldsymbol{T}_{x_1,\ldots,x_d} = \begin{cases} t, & x_1 = k_1, \ldots, x_d = k_d \\ 0, & \text{sonst} \end{cases}, \tag{2.2.1}$$

mit $1 \leq x_i \leq n_i$ für $i = 1, \ldots, d$. Dann ist $\boldsymbol{T}$ ein Rank-One-Tensor mit $\boldsymbol{T}^{(1)} = t \cdot e^{(k_1)}$ und $\boldsymbol{T}^{(j)} = e^{(k_j)}$ für $j = 2, \ldots, d$, wobei $e^{(k_i)}$, $i = 1, \ldots, d$, Einheitsvektoren sind:

$$e_x^{(k_i)} = \begin{cases} 1 & x = k_i \\ 0, & \text{sonst} \end{cases},$$

mit $1 \leq x \leq n_i$.

Da jeder Tensor als Summe von Tensoren der Form (2.2.1) geschrieben werden kann, indem die Einträge des Tensors einzeln auf die Summanden aufgeteilt werden, ist jeder Tensor als Summe von Rank-One-Tensoren darstellbar. □

Auf dieser Idee basiert das kanonische Format. Hierbei fügen wir den Vektoren $\boldsymbol{T}^{(i)}$ einen sogenannten Bondindex hinzu, entlang welchem wir die verschiedenen Vektoren einer Mode zusammenfügen. Über diesen Index summieren wir dann. Somit schreiben wir den Tensor als endliche Summe von Rank-One-Tensoren.

Definition 2.2.5 (Kanonisches Tensorformat)**.** *Ein Tensor $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, ist im kanonischen Tensorformat, wenn Folgendes gilt:*

$$\boldsymbol{T} = \sum_{k=1}^{r} \bigotimes_{i=1}^{d} \boldsymbol{T}_{k,:}^{(i)} = \sum_{k=1}^{r} \boldsymbol{T}_{k,:}^{(1)} \otimes \ldots \otimes \boldsymbol{T}_{k,:}^{(d)}.$$

Wir nennen $\boldsymbol{T}^{(i)} \in \mathbb{R}^{r \times n_i}$, $i = 1, \ldots, d$, die kanonischen Kerne und r den kanonischen Rang oder die Bonddimension der Zerlegung.

Für ein Element des Tensors gilt: $\boldsymbol{T}_{x_1,\ldots,x_d} = \sum_{k=1}^{r} \prod_{i=1}^{d} \boldsymbol{T}_{k,x_i}^{(i)} = \sum_{k=1}^{r} \boldsymbol{T}_{k,x_1}^{(1)} \cdot \ldots \cdot \boldsymbol{T}_{k,x_d}^{(d)}$.

Wir verwenden die Doppelpunktnotation aus Matlab, um freie Moden zu kennzeichnen, wenn bestimmte Indizes festgehalten werden. Somit gibt uns $\boldsymbol{T}_{k,:}^{(i)} \in \mathbb{R}^{n_i}$ die k-te Zeile von $\boldsymbol{T}^{(i)} \in \mathbb{R}^{r \times n_i}$.

Beispiel 2.2.6 zeigt eine solche kanonische Zerlegung. Der erste und dritte Vektor der Summe werden als Zeilen von $\boldsymbol{T}^{(1)}$ zusammengefasst und der zweite und vierte Vektor als Zeilen von $\boldsymbol{T}^{(2)}$.

Beispiel 2.2.6.

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}^T + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}^T = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \sum_{k=1}^{r} \bigotimes_{i=1}^{d} \boldsymbol{T}_{k,:}^{(i)}$$

mit $\boldsymbol{T}^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, $\boldsymbol{T}^{(2)} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, $r = 2$ *und* $d = 2$.

Das folgende Lemma zeigt, dass der Speicherverbrauch im kanonischen Format nur linear von d abhängt.

Lemma 2.2.7. *Sei $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, ein Tensor im kanonischen Format mit Bonddimension r und maximaler Modengröße $n = \max_i n_i$. Dann liegt der Speicherverbrauch von $\boldsymbol{T}$ in $\mathcal{O}(dnr)$.*

Beweis. Siehe [2, Lemma 3.2.3]. □

Wir können durch Tensorzerlegungen also tatsächlich die exponentielle Abhängigkeit von d vermeiden. Allerdings treten auch einige Schwierigkeiten bei der Verwendung von Tensorzerlegungen auf.

Beispielsweise werden sowohl der Speicherverbrauch als auch die Darstellbarkeit eines Tensors im jeweiligen Format von der Bonddimension r beeinflusst. Ist r groß, so bringt uns die Tensorzerlegung eventuell keinen Vorteil bei der Anzahl der benötigten Elemente. In Beispiel 2.2.6 haben die Kerne zusammen mehr Elemente als die Ergebnismatrix. Allerdings werden die zwei Summanden gebraucht, um die Matrix im kanonischen Format darzustellen. Würde man die Bonddimension beschränken, so können einige Tensoren nicht mehr dargestellt werden und müssen approximiert werden.

Zusätzlich kann die Bonddimension eines zerlegten Tensors durch Operationen wachsen. Führen wir beispielsweise eine Addition zweier Tensoren oder eine Multiplikation eines Tensors mit einem Operator in einem Zerlegungsformat durch, so können sich die Bonddimensionen addieren beziehungsweise multiplizieren (siehe [2, Kapitel 3.4.2]). In einem Algorithmus würde die Bonddimension somit immer weiter anwachsen und die Rechnung ineffizient machen.

Um dies zu vermeiden, müssen Low-Rank-Tensor-Approximationen berechnet werden. Den approximierten Tensor $\boldsymbol{B}$ von $\boldsymbol{T}$ nennen wir Truncation von $\boldsymbol{T}$. Die Genauigkeit der Truncation wird in der Frobeniusnorm $\|\cdot\|_F$ angegeben. Diese ist für einen Tensor $\boldsymbol{T} \in \mathbb{R}^N$ wie folgt definiert:

$$\|\boldsymbol{T}\|_F = \sqrt{\sum_{x_1=1}^{n_1} \cdots \sum_{x_d=1}^{n_d} |\boldsymbol{T}_{x_1,\ldots,x_d}|^2} = \sqrt{\langle \boldsymbol{T}, \boldsymbol{T} \rangle_N}.$$

Die im Folgenden verwendete Notation $\mathrm{bond}(\boldsymbol{T})$ steht für die geordnete Menge der Bonddimensionen eines zerlegten Tensors $\boldsymbol{T}$. Im kanonischen Format ist dies einfach der kanonische Rang. Wir werden aber noch Formate kennenlernen, welche mehrere Bondindizes mit verschiedenen Bonddimensionen besitzen.

Definition 2.2.8 (Truncation)**.** *Wir nennen einen Tensor $\boldsymbol{B} = \mathcal{T}_{\epsilon,R}(\boldsymbol{T})$ die Truncation von $\boldsymbol{T}$ oder die Approximation mit niedrigerer Bonddimension, wenn $\|\boldsymbol{T}-\boldsymbol{B}\|_F \leq \epsilon\|\boldsymbol{T}\|_F$ mit Toleranz ϵ für die Genauigkeit und* $\max(\mathrm{bond}(\boldsymbol{B})) = R < \max(\mathrm{bond}(\boldsymbol{T}))$ *mit maximaler Bonddimension R.*

Bei der Berechnung einer solchen Truncation fordert man in der Regel entweder eine gewünschte Toleranz für die Genauigkeit $\epsilon \approx \epsilon_{desired}$ oder beschränkt die maximale Bonddimension $R \leq R_{max}$. Wird sowohl $\epsilon_{desired}$ als auch R_{max} vorgegeben, so besteht die Möglichkeit, dass keine geeignete Truncation existiert. Ist die Schranke R_{max} der maximalen Bonddimension zu klein, um den Tensor mit der gewünschten Toleranz $\epsilon_{desired}$ zu approximieren, wird alternativ eine Lösung mit $\epsilon \gg \epsilon_{desired}$ berechnet, was zu einem großen Genauigkeitsverlust führt. Wir nennen einen Tensor low-rank-approximierbar, wenn eine Truncation des Tensors mit hoher Genauigkeit, das bedeutet mit einem kleinem $\epsilon_{desired}$, und einer kleinen Schranke R_{max} für die maximale Bonddimension existiert.

Nicht für alle Formate existieren robuste Algorithmen, die eine solche Truncation bestimmen können. Insbesondere ist die Menge der Tensoren $\boldsymbol{B}$ mit $\max(\text{bond}(\boldsymbol{B})) \leq R_{max}$ in einigen Formaten, wie beispielsweise dem kanonischen Format, nicht abgeschlossen. Das Problem, die beste Approximation mit maximaler Bonddimension R_{max} in der Frobeniusnorm zu bestimmen, ist hier nicht wohlgestellt [11], wodurch eine Lösung eventuell nicht existiert. Hierdurch lässt sich kein zuverlässiger Algorithmus für die Berechnung einer Truncation finden.

Ein Format, welches algorithmisch stabil ist, das heißt, für welches die beste Approximation immer existiert, ist das Tucker-Format [12] (vergleiche [2, S.27]). Hierbei wird der Tensor in mehrere Faktormatrizen und einen Kerntensor zerlegt. Der Kerntensor hat zwar die gleiche Ordnung wie der gegebene Tensor, allerdings kleinere Modengröße.

Definition 2.2.9 (Tucker-Format)**.** *Ein Tensor $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, ist im Tucker-Format, wenn Folgendes gilt:*

$$\boldsymbol{T} = \sum_{k_1=1}^{r_1} \cdots \sum_{k_d=1}^{r_d} (\boldsymbol{T}^{(1)}_{:,k_1} \otimes \cdots \otimes \boldsymbol{T}^{(d)}_{:,k_d}) \cdot \boldsymbol{U}_{k_1,\ldots,k_d} = (\boldsymbol{T}^{(1)} \otimes \ldots \otimes \boldsymbol{T}^{(d)}) \cdot \boldsymbol{U}.$$

Wir nennen $\boldsymbol{U} \in \mathbb{R}^{r_1 \times \cdots \times r_d}$ den Tucker-Kern, $\boldsymbol{T}^{(i)} \in \mathbb{R}^{n_i \times r_i}$, $i = 1, \ldots, d$, die Tucker-Faktoren und r_i die Tucker-Ränge oder Bonddimensionen.

Hier ist nun die interessante Frage, ob dieses Format ebenfalls dem Fluch der Dimensionalität entgeht.

Lemma 2.2.10. *Sei $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, ein Tensor im Tucker-Format mit maximaler Bonddimension $r = \max_i r_i$ und maximaler Modengröße $n = \max_i n_i$. Dann kann der Speicherbedarf von $\boldsymbol{T}$ durch $\mathcal{O}(r \cdot n \cdot d + r^d)$ abgeschätzt werden.*

Beweis. Siehe [2, Lemma 3.3.2]. □

Bei diesem Format bleibt also eine exponentielle Abhängigkeit von d bestehen, weshalb es nur für kleine Bonddimensionen sinnvoll ist. Das Tucker-Format löst also das Problem mit der Truncation, allerdings nicht das Problem mit dem Fluch der Dimensionalität.

2.3 Tensor-Train-Format (TT-Format)

Ein sehr vielversprechendes Format ist das Tensor-Train-Format. Hierbei wird der Tensor wieder in mehrere Kerne zerlegt, ähnlich dem kanonischen Format. Im Tensor-Train-Format haben die Kerne jeweils zwei Bondindizes, wobei sich zwei benachbarte Kerne jeweils einen Bondindex teilen. Es wird dann über alle Bondindizes von allen Kernen summiert.

Definition 2.3.1 (Tensor-Train-Format (TT-Format))**.** *Ein Tensor $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, ist im Tensor-Train-Format, wenn Folgendes gilt:*

$$\boldsymbol{T} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \bigotimes_{i=1}^{d} \boldsymbol{T}^{(i)}_{k_{i-1},:,k_i} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{T}^{(1)}_{k_0,:,k_1} \otimes \ldots \otimes \boldsymbol{T}^{(d)}_{k_{d-1},:,k_d}. \tag{2.3.1}$$

Wir nennen die Tensoren $\boldsymbol{T}^{(i)} \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$, $i = 1, \ldots, d$, *von Ordnung 3 die TT-Kerne und* r_i *die TT-Ränge oder Bonddimensionen. Es gilt* $r_0 = r_d = 1$ *und* $r_i \geq 1$ *für* $i = 1, \ldots, d-1$.

Für ein Element des Tensors gilt: $\boldsymbol{T}_{x_1,\ldots,x_d} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{T}^{(1)}_{k_0,x_1,k_1} \cdot \ldots \cdot \boldsymbol{T}^{(d)}_{k_{d-1},x_d,k_d} = \boldsymbol{T}^{(1)}_{:,x_1,:} \cdot \ldots \cdot \boldsymbol{T}^{(d)}_{:,x_d,:}$.

Durch die vielen Bondindizes wird vermieden, dass ein in mehreren Summanden gleichbleibender Faktor mehrfach gespeichert wird. Somit können die TT-Ränge kleiner werden als der kanonische Rang.

Beispiel 2.3.2. *Ein Tensor der Form* $v \otimes w \otimes w + w \otimes v \otimes w + w \otimes w \otimes v$ *mit* v, $w \in \mathbb{R}^n$ *muss im kanonischen Format als* $\boldsymbol{T}^{(1)} = (v, w, w)^T$, $\boldsymbol{T}^{(2)} = (w, v, w)^T$ *und* $\boldsymbol{T}^{(3)} = (w, w, v)^T$ *gespeichert werden. Im TT-Format zerlegen wir denselben Tensor in* $\boldsymbol{T}^{(1)} = (v, w)$, $\boldsymbol{T}^{(2)} = \begin{pmatrix} w & 0 \\ v & w \end{pmatrix}$ *und* $\boldsymbol{T}^{(3)} = (w, v)^T$. $\boldsymbol{T}^{(1)}$ *und* $\boldsymbol{T}^{(3)}$ *speichern also einen Faktor weniger und wir erhalten die TT-Ränge 1,2,2,1 statt des kanonischen Rangs 3.*

Zu Bonddimensionen in verschiedenen Formaten gibt es Paper wie [13], welche die Bonddimensionen eines Tensors in einem Format durch die Bonddimensionen des Tensors in einem anderen Format abschätzen. Hierauf gehen wir allerdings nicht genauer ein.

Wir können auch Tensoroperatoren im TT-Format darstellen.

Definition 2.3.3 (Tensor-Train-Operator)**.** *Ein Tensoroperator* $\boldsymbol{G} \in \mathbb{R}^{M \times N}$ *ist im TT-Format, wenn Folgendes gilt:*

$$\boldsymbol{G} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \bigotimes_{i=1}^{d} \boldsymbol{G}^{(i)}_{k_{i-1},:,:,k_i} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{G}^{(1)}_{k_0,:,:,k_1} \otimes \ldots \otimes \boldsymbol{G}^{(d)}_{k_{d-1},:,:,k_d},$$

mit TT-Kernen $\boldsymbol{G}^{(i)} \in \mathbb{R}^{r_{i-1} \times m_i \times n_i \times r_i}$ *für* $i = 1, \ldots, d$ *und* $r_0 = r_d = 1$.

Für ein Element von $\boldsymbol{G}$ *gilt:* $\boldsymbol{G}_{x_1,y_1,\ldots,x_d,y_d} = \boldsymbol{G}^{(1)}_{:,x_1,y_1,:} \cdot \ldots \cdot \boldsymbol{G}^{(d)}_{:,x_d,y_d,:}$.

Graphisch können wir uns einen Tensor im Tensor-Train-Format $\boldsymbol{T} \in \mathbb{R}^N$ oder einen TT-Operator $\boldsymbol{G} \in \mathbb{R}^{M \times N}$ wie in Abbildung 2.4 vorstellen. Hierbei steht jeder Knoten für einen Kern, die vertikalen Kanten für die auf die Kerne verteilten Moden des Tensors und die horizontalen Kanten für die Bonddimensionen, welche jeweils zwei benachbarte Kerne durch Indexkontraktion miteinander verbinden. Hier sehen wir leicht, dass die Bondindizes nur zwischen den Kernen operieren und nach außen nicht sichtbar sind. Diese Darstellung erinnert an aneinandergekoppelte Wagons eines Zuges, woher dieses Format seinen Namen hat.

Auch für dieses Format betrachten wir wieder den Speicherbedarf.

Lemma 2.3.4. *Sei* $\boldsymbol{T} \in \mathbb{R}^N$ *ein Tensor im TT-Format und* $\boldsymbol{G} \in \mathbb{R}^{M \times N}$ *ein TT-Operator mit* $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$ *und* $M = (m_1, \ldots, m_d) \in \mathbb{N}^d$. *Dann liegt der Speicherverbrauch von* $\boldsymbol{T}$ *in* $\mathcal{O}(dnr^2)$ *und der Speicherverbrauch von* $\boldsymbol{G}$ *in* $\mathcal{O}(dmnr_{Op}^2)$, *wobei* $n = \max_i n_i$, $m = \max_i m_i$, $r = \max(\mathrm{bond}(\boldsymbol{T}))$ *die maximale Bonddimension von* $\boldsymbol{T}$ *und* $r_{Op} = \max(\mathrm{bond}(\boldsymbol{G}))$ *die maximale Bonddimension von* $\boldsymbol{G}$ *ist.*

(a) TT-Tensor (b) TT-Operator

Abbildung 2.4: Graphische Darstellung im Tensor-Train-Format

Beweis. Siehe [2, Lemma 3.4.3]. □

Das bedeutet, dass wir im TT-Format keine exponentielle Abhängigkeit von d haben. Zusätzlich können in diesem Format Truncations berechnet werden, wie folgendes Lemma zeigt.

Lemma 2.3.5. *Die beste TT-Approximation $\boldsymbol{T}^{best}$ eines Tensors $\boldsymbol{T}$ in der Frobeniusnorm mit Bonddimensionen $r_k \leq R_{max}$ existiert immer und es existiert ein Algorithmus (TT-SVD-Algorithmus), welcher eine quasi-optimale Lösung $\boldsymbol{B}$ berechnet:*

$$\|\boldsymbol{T} - \boldsymbol{B}\|_F \leq \sqrt{d-1}\|\boldsymbol{T} - \boldsymbol{T}^{best}\|_F.$$

Beweis. Siehe [3, Korollar 2.4]. □

Da dieses Format die beiden Vorteile des kanonischen und des Tucker-Formats vereint, haben wir es für unsere Experimente ausgewählt.

Im Folgenden gehen wir kurz auf die Definition für Orthonormalität ein und betrachten die Truncation im TT-Format genauer. Auf die Berechnung einiger Standardoperationen wie Additionen oder Matrix-Vektor-Multiplikationen bei der Verwendung des TT-Formats wird in Kapitel 5 eingegangen.

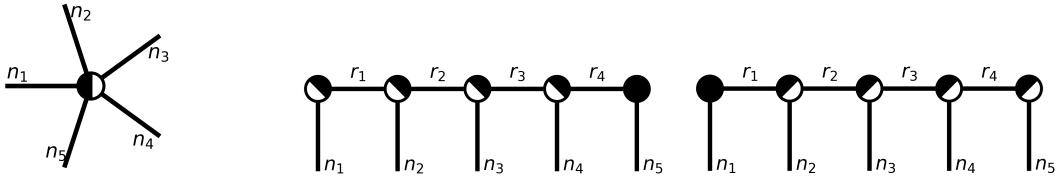
2.3.1 Orthonormalität

Hier wollen wir uns mit der Orthonormalität von Tensoren und der TT-Zerlegung beschäftigen, indem wir die folgenden Begriffe für eine Rechteckmatrix verallgemeinern.

Eine Matrix $A \in \mathbb{R}^{m\times n}$ ist orthonormal bezüglich der Zeilen, wenn gilt $A \cdot A^T = I \in \mathbb{R}^{m\times m}$ und orthonormal bezüglich der Spalten, wenn gilt $A^T \cdot A = I \in \mathbb{R}^{n\times n}$.

Definition 2.3.6. *Sei $\boldsymbol{T} \in \mathbb{R}^N$ ein Tensor, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$ eine Indexmenge und $N' = (n_{k_1}, \ldots, n_{k_e})$ und $N'' = (n_{l_1}, \ldots, n_{l_f})$ mit $e + f = d$ Teilmengen von N, sodass N disjunkte Vereinigung von N' und N'' ist. $\boldsymbol{T}$ heißt orthonormal bezüglich N', wenn die Matrizisierung von $\boldsymbol{T}$ bezüglich N' und N'' folgende Bedingung erfüllt:*

$$\boldsymbol{T}|_{N'}^{N''} \cdot \left(\boldsymbol{T}|_{N'}^{N''}\right)^T = \boldsymbol{T}|_{N'}^{N''} \cdot \boldsymbol{T}|_{N''}^{N'} = \boldsymbol{I} \in \mathbb{R}^{N'\times N'}.$$



(a) orthonormaler Tensor bezüglich $N' = (n_1, n_2, n_5)$

(b) links-orthonormaler TT-Tensor

(c) rechts-orthonormaler TT-Tensor

Abbildung 2.5: Graphische Darstellung der Orthonormalität

Hierbei ist der Identitätstensor $\boldsymbol{I} \in \mathbb{R}^N$ *definiert als* $\boldsymbol{I}_{x_1,y_1,\ldots,x_d,y_d} = \delta_{x_1,y_1} \cdot \ldots \cdot \delta_{x_d,y_d}$ *mit dem Kronecker-Delta*

$$\delta_{x_i,y_i} = \begin{cases} 1, & x_i = y_i \\ 0, & sonst \end{cases},$$

für $i = 1, \ldots, d$.

Dies wollen wir auf TT-Zerlegungen erweitern. Wir definieren uns für einen TT-Kern $\boldsymbol{T}^{(i)} \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$ die Matrizisierungen $\mathcal{L}(\boldsymbol{T}^{(i)}) = \boldsymbol{T}^{(i)}|^{r_i}_{r_{i-1},n_i}$, genannt linke Entfaltung von $\boldsymbol{T}^{(i)}$, und $\mathcal{R}(\boldsymbol{T}^{(i)}) = \boldsymbol{T}^{(i)}|^{n_i,r_i}_{r_{i-1}}$, genannt rechte Entfaltung von $\boldsymbol{T}^{(i)}$.

Ein TT-Kern ist links-orthonormal, wenn seine linke Entfaltung orthonormal bezüglich der Spalten ist, das heißt $(\mathcal{L}(\boldsymbol{T}^{(i)}))^T \cdot \mathcal{L}(\boldsymbol{T}^{(i)}) = I \in \mathbb{R}^{r_i \times r_i}$. Ein TT-Kern ist rechts-orthonormal, wenn seine rechte Entfaltung orthonormal bezüglich der Zeilen ist, das heißt $\mathcal{R}(\boldsymbol{T}^{(i)}) \cdot (\mathcal{R}(\boldsymbol{T}^{(i)}))^T = I \in \mathbb{R}^{r_{i-1} \times r_{i-1}}$.

Wir nennen einen Tensor-Train-Tensor links-orthonormal, wenn alle TT-Kerne ausgenommen der letzte links-orthonormal sind. Analog ist eine TT-Zerlegung rechts-orthonormal, wenn bis auf den ersten TT-Kern alle rechts-orthonormal sind.

Da in vielen Algorithmen die Orthonormalität eine Rolle spielt, wollen wir auch diese in der Visualisierung sichtbar machen. Hierzu stellen wir den Knoten als halb-gefüllten Kreis dar. Die Kanten werden so angeordnet, dass die zu N' gehörigen Kanten am ausgefüllten Teil des Knotens hängen, während die zu N'' gehörigen Kanten, welche bezüglich N' orthonormalisiert wurden, mit dem nicht ausgefüllten Teil verbunden sind (vergleiche Abbildung 2.5).

2.3.2 Truncation

Da die Truncation eine wichtige Rolle bei der Verwendung von Tensorzerlegungen spielt, werden wir uns hier mit der Idee hinter Truncation-Verfahren im Tensor-Train-Format beschäftigen. Hierbei betrachten wir zwei verschiedene Verfahren. Der TT-SVD-Algorithmus [3, Kapitel 2] wird genutzt, um die TT-Darstellung eines noch nicht zerlegten Tensors zu berechnen, wobei gleichzeitig eine Truncation ausgeführt werden kann. Liegt der Tensor bereits im TT-Format vor, so wird der TT-rounding-Algorithmus [3, Kapitel 3] verwendet, um die Truncation direkt im TT-Format durchzuführen und hierdurch die Bonddimension zu reduzieren.

Bevor wir zur Tensor-Approximation kommen, betrachten wir zunächst die Approximation einer Matrix.

Definition 2.3.7. *Eine Singulärwertzerlegung (SVD) einer Matrix $A \in \mathbb{R}^{m\times n}$ ist ein Produkt $A = U\Sigma V^T$, wobei U und V orthonormal bezüglich ihrer Spalten sind und Σ eine Diagonalmatrix der Singulärwerte von A ist.*

Betrachten wir nur die s größten Singulärwerte, so erhalten wir die abgeschnittene (truncated) Singulärwertzerlegung $\tilde{A} = U_t\Sigma_t V_t^T$ mit $U_t \in \mathbb{R}^{m\times s}$, $V_t \in \mathbb{R}^{n\times s}$, $\Sigma_t \in \mathbb{R}^{s\times s}$.

Satz 2.3.8 (Eckart-Young-Theorem)**.** *Sei $A \in \mathbb{R}^{m\times n}$ eine Matrix und $\tilde{A} = U_t\Sigma_t V_t$ die abgeschnittene Singulärwertzerlegung der s größten Singulärwerte von A. Dann ist $\tilde{A}$ die beste Approximation von A in der Frobeniusnorm mit Rang höchstens s:*

$$\min_{\operatorname{rang}(B)\leq s} \|A - B\|_F = \|A - \tilde{A}\|_F.$$

Beweis. Siehe [14]. □

Somit ist klar festgelegt, welche die beste Approximation mit beschränktem Rang einer Matrix ist. In höheren Dimensionen gibt es eine entsprechende Aussage für die beste Approximation eines Tensors nicht, da von den verschiedenen möglichen Zerlegungen keine eindeutig optimal ist. Dennoch will man sich die abgeschnittenen Singulärwertzerlegungen bei der Truncation im TT-Format zunutze machen. Die Idee hierbei liegt darin, die Approximation des Tensors durch sukzessive, abgeschnittene Singulärwertzerlegungen zu berechnen.

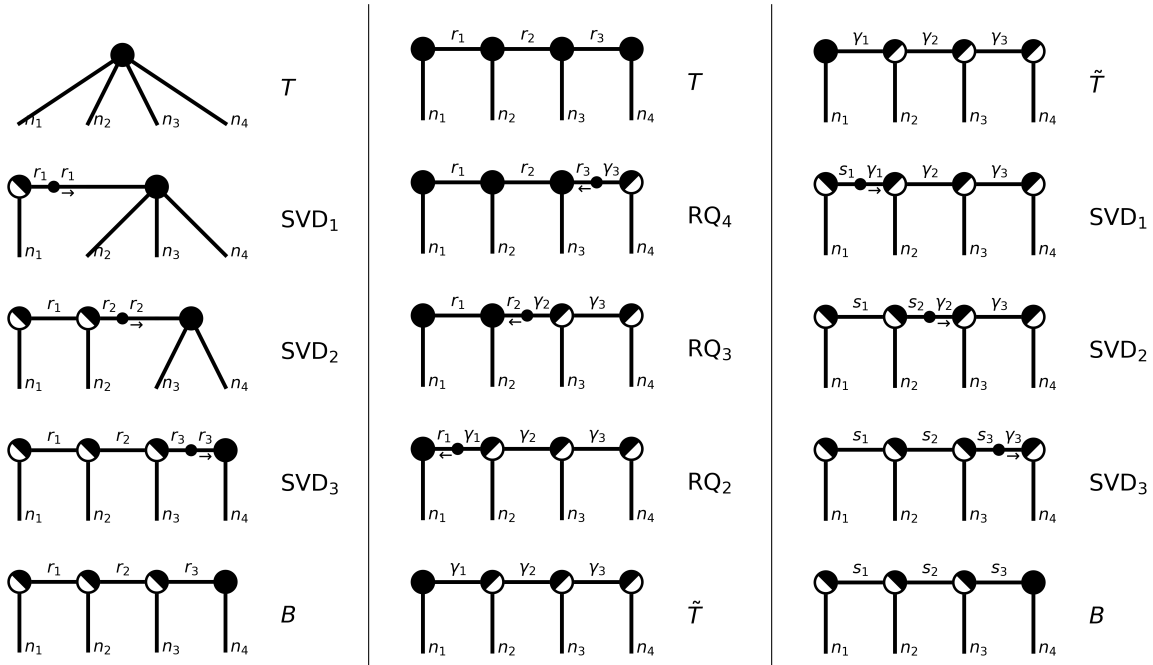
Betrachten wir zunächst die Berechnung der (approximierten) Tensor-Train-Zerlegung eines vollen Tensors $\boldsymbol{T} \in \mathbb{R}^{n_1\times\cdots\times n_d}$ mittels dem TT-SVD-Algorithmus [3, Kapitel 2]. Dieser geht Mode für Mode vor. Abbildung 2.6a zeigt eine graphische Darstellung des TT-SVD-Algorithmus zur Veranschaulichung der folgende Beschreibung des Algorithmus.

Wir betrachten zunächst die Matrizisierung $A = \boldsymbol{T}|_{r_0,n_1}^{n_2,\ldots,n_d}$ von $\boldsymbol{T}$, wobei $r_0 = 1$. Für die Matrix A können wir nun die abgeschnittene Singulärwertzerlegung $A = U_t\Sigma_t V_t^T$ berechnen. Hierbei enthält Σ_t nur die r_1 Singulärwerte, die über dem Schwellwert $\delta = \frac{\epsilon}{\sqrt{d-1}}\|\boldsymbol{T}\|_F$ liegen, wobei ϵ die Toleranz der zu erreichende Genauigkeit der Truncation ist. Soll die TT-Darstellung des Tensors nicht approximiert werden, berechnet man die gewöhnliche SVD von A.

Wir definieren uns den ersten Kern $\boldsymbol{T}^{(1)} \in \mathbb{R}^{r_0\times n_1\times r_1}$ so, dass $\boldsymbol{T}^{(1)}|_{r_0,n_1}^{r_1} = U_t$. Den Rest der Zerlegung definieren wir als unser neues $\tilde{\boldsymbol{T}} \in \mathbb{R}^{r_1\times n_2\times\cdots\times n_d}$: $\tilde{\boldsymbol{T}}|_{r_1}^{n_2,\ldots,n_d} = \Sigma_t V_t^T$. Somit haben wir die erste Mode unseres Tensors abgespalten.

Wenden wir obiges sukzessiv für die weiteren Moden $n_2, \ldots, n_{d-1}$ auf den Tensor $\tilde{\boldsymbol{T}}$ an, so erhalten wir die Kerne $\boldsymbol{T}^{(2)}, \ldots, \boldsymbol{T}^{(d-1)}$ der (approximierten) Tensor-Train-Zerlegung von $\boldsymbol{T}$. Den letzten Kern der TT-Zerlegung setzen wir auf den Rest der Zerlegung nach $d-1$ Iterationen, das heißt $\boldsymbol{T}^{(d)} = \tilde{\boldsymbol{T}} = \Sigma_t V_t^T \in \mathbb{R}^{r_{d-1}\times n_d\times r_d}$ mit $r_d = 1$. Somit erhalten wir die TT-Darstellung der Truncation $\boldsymbol{B}$ des vollen Tensors $\boldsymbol{T}$ beziehungsweise die TT-Darstellung von $\boldsymbol{T}$, wenn wir nicht approximiert haben.

Den Grund für die obige Wahl von δ zeigt das folgende Lemma.



(a) TT-SVD-Algorithmus

(b) Schritt 1 des TT-rounding-Algorithmus

(c) Schritt 2 des TT-rounding-Algorithmus

Abbildung 2.6: Graphische Darstellung der Truncation-Algorithmen

Lemma 2.3.9. *Damit die durch den TT-SVD-Algorithmus berechnete Truncation* $\boldsymbol{B}$ *eines Tensors* $\boldsymbol{T}$ *die Bedingung* $\|\boldsymbol{T} - \boldsymbol{B}\|_F \leq \epsilon \|\boldsymbol{T}\|_F$ *aus Definition 2.2.8 erfüllt, muss für den Schwellwert* δ *der Singulärwertzerlegung Folgendes gelten:*

$$\delta = \frac{\epsilon}{\sqrt{d-1}} \|\boldsymbol{T}\|_F.$$

Beweis. Siehe [3, S. 2300]. □

Nun kommen wir zum TT-rounding-Algorithmus [3, Kapitel 3]. Dieses Verfahren wird genutzt, wenn ein Tensor bereits im Tensor-Train-Format dargestellt ist $\boldsymbol{T} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{T}^{(1)}_{k_0,:,k_1} \otimes \cdots \otimes \boldsymbol{T}^{(d)}_{k_{d-1},:,k_d}$, die Darstellung allerdings große Bonddimensionen r_k hat. Der Algorithmus berechnet die TT-Darstellung einer Truncation von $\boldsymbol{T}$ mit kleineren Bonddimensionen. Waren die Bonddimensionen r_k nicht optimal, das bedeutet, es existiert eine TT-Darstellung von $\boldsymbol{T}$ mit der Genauigkeit der ursprünglichen Zerlegung aber kleineren Bonddimensionen, so verringert der Algorithmus entsprechend die Bonddimensionen.

Auch bei diesem Verfahren werden sukzessive, abgeschnittene SVDs verwendet, um die Bonddimensionen zu verkleinern. Damit der Algorithmus stabil funktioniert, muss allerdings zuvor die TT-Darstellung auf eine gewisse Weise orthonormalisiert werden (für mehr Details zur Herleitung siehe [3, S.2302-2304]). Abbildung 2.6b zeigt eine Vi-

sualisierung des Orthonormalisierungsprozesses und Abbildung 2.6c die anschließenden SVDs.

Wir starten bei der Orthonormalisierung beim letzten Kern $\boldsymbol{T}^{(d)} \in \mathbb{R}^{r_{d-1} \times n_d \times r_d}$. Wir berechnen für die Matrizisierung $\boldsymbol{T}^{(d)}|_{r_{d-1}}^{n_d, r_d}$ die RQ-Zerlegung $\boldsymbol{T}^{(d)}|_{r_{d-1}}^{n_d, r_d} = RQ$, wobei $Q \in \mathbb{R}^{\gamma_{d-1} \times n_d r_d}$ orthonormal bezüglich der Zeilen und $R \in \mathbb{R}^{r_{d-1} \times \gamma_{d-1}}$ eine rechte obere Dreiecksmatrix ist. Wir definieren $\tilde{\boldsymbol{T}}^{(d)} \in \mathbb{R}^{\gamma_{d-1} \times n_d \times r_d}$ so, dass $\tilde{\boldsymbol{T}}^{(d)}|_{\gamma_{d-1}}^{n_d, r_d} = Q$ und heften die Matrix R durch Indexkontraktion an den vorherigen Kern: $\bar{\boldsymbol{T}}^{(d-1)} = \langle \boldsymbol{T}^{(d-1)}, R \rangle_{r_{d-1}}$. Hierdurch haben wir nun einen rechts-orthonormalen letzten Kern.

Diesen Prozess führen wir nun sukzessiv für die Kerne $\bar{\boldsymbol{T}}^{(k)}$ für $d > k > 1$ durch, wodurch wir eine rechts-orthonormale TT-Darstellung mit Kernen $\tilde{\boldsymbol{T}}^{(k)} \in \mathbb{R}^{\gamma_{k-1} \times n_k \times \gamma_k}$ erhalten.

Nun berechnen wir die abgeschnittenen Singulärwertzerlegungen, wobei wir beim ersten Kern $\tilde{\boldsymbol{T}}^{(1)} \in \mathbb{R}^{\gamma_0 \times n_1 \times \gamma_1}$ starten. Für die Matrizisierung $\tilde{\boldsymbol{T}}^{(1)}|_{\gamma_0, n_1}^{\gamma_1}$ berechnen wir die abgeschnittene Singulärwertzerlegung, wobei wir wieder nur die s_1 Singulärwerte betrachten, die über dem Schwellwert $\delta = \frac{\epsilon}{\sqrt{d-1}} \|\boldsymbol{T}\|_F$ liegen:

$$\tilde{\boldsymbol{T}}^{(1)}|_{\gamma_0, n_1}^{\gamma_1} = U_t \Sigma_t V_t^T$$

mit $U_t \in \mathbb{R}^{\gamma_0 n_1 \times s_1}$, $\Sigma_t \in \mathbb{R}^{s_1 \times s_1}$ und $V_t \in \mathbb{R}^{\gamma_1 \times s_1}$.

Definieren wir $\hat{\boldsymbol{T}}^{(1)} \in \mathbb{R}^{\gamma_0 \times n_1 \times s_1}$ so, dass $\hat{\boldsymbol{T}}^{(1)}|_{\gamma_0, n_1}^{s_1} = U_t$, so erhalten wir einen Kern mit niedrigerer Bonddimension $s_1 \leq r_1$. Der Rest $\Sigma_t V_t^T$ der abgeschnittenen Singulärwertzerlegung wird durch Indexkontraktion dem nächsten Kern mitgegeben: $\bar{\boldsymbol{T}}^{(2)} = \langle \Sigma_t V_t^T, \tilde{\boldsymbol{T}}^{(2)} \rangle_{\gamma_1}$. Hierdurch erhalten wir einen Kern $\bar{\boldsymbol{T}}^{(2)} \in \mathbb{R}^{s_1 \times n_1 \times \gamma_2}$ und haben somit die Bonddimension r_1 auf s_1 reduziert.

Führen wir dies sukzessiv für die Kerne $\bar{\boldsymbol{T}}^{(i)}$ für $1 < i < d$ durch, erhalten wir den Truncation-Tensor $\boldsymbol{B} = \sum_{k_0=1}^{s_0} \cdots \sum_{k_d=1}^{s_d} \hat{\boldsymbol{T}}_{k_0,:,k_1}^{(1)} \otimes \cdots \otimes \hat{\boldsymbol{T}}_{k_{d-1},:,k_d}^{(d)}$ mit $s_i \leq r_i \ \forall i$ und $\|\boldsymbol{T} - \boldsymbol{B}\|_F \leq \epsilon \|\boldsymbol{T}\|_F$.

Lemma 2.3.10. *Die Anzahl benötigter Rechenoperationen des TT-rounding-Algorithmus zur Berechnung einer Truncation eines TT-Tensors $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, kann durch $\mathcal{O}(dnr^3)$ abgeschätzt werden, wobei $n = \max_i n_i$ und $r = \max(\text{bond}(\boldsymbol{T}))$.*

Beweis. Siehe [3, S. 2305]. □

Wir haben nun gezeigt, wie man im TT-Format eine Truncation berechnen kann und haben eine Abschätzung der Kosten des Algorithmus. Wir interessieren uns nun für eine obere Schranke der Bonddimensionen nach einer solchen Truncation.

Hierzu benötigen wir den folgenden Hilfssatz über den δ-Rang einer Matrix A. Hierbei ist der δ-Rang einer Matrix A definiert als der minimale Rang aller Matrizen $\tilde{A}$, die $\|A - \tilde{A}\|_F \leq \delta$ erfüllen.

Hilfssatz 2.3.11. *Der δ-Rang einer Matrix A ist beschränkt durch den Rang der Matrix A.*

Beweis. Da die Matrix $\tilde{A} = A$ die Bedingung $\|A - \tilde{A}\|_F \leq \delta$ für alle $\delta \geq 0$ erfüllt, kann der δ-Rang nach oben durch den Rang der Matrix A beschränkt werden. □

Satz 2.3.12. *Sei $\boldsymbol{B}$ die durch einen der obigen Algorithmen berechnete Truncation im TT-Format eines Tensors $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$. Dann können die Bonddimensionen r_k, $k = 0, \ldots, d$, von $\boldsymbol{B}$ wie folgt abgeschätzt werden:*

$$r_k \leq n^{d/2 - |d/2 - k|},$$

mit $n = \max_i n_i$.

Beweis. Die Bonddimension r_k, welche wir durch die obigen Algorithmen für die TT-Darstellung von $\boldsymbol{B}$ erhalten, entspricht dem δ-Rang der Matrizisierung $T_k = \boldsymbol{T}|_{n_1,\ldots,n_k}^{n_{k+1},\ldots,n_d}$ des Tensors $\boldsymbol{T}$, $k = 0, \ldots, d$ (siehe [3, S. 2300 und Algorithmus 2]).

Nach Hilfssatz 2.3.11 können wir die Bonddimension r_k durch den Rang von T_k beschränken. Für den Rang einer Matrix $A \in \mathbb{R}^{m \times n}$ gilt, dass $\operatorname{rang}(A) \leq \min(m, n)$. T_k ist eine Matrix der Größe $(\prod_{s=1}^{k} n_s) \times (\prod_{s=k+1}^{d} n_s)$, $k = 0, \ldots, d$.

Somit können wir die Bonddimensionen r_k für $k = 0, \ldots, d$ wie folgt abschätzen:

$$r_k \leq \operatorname{rang}(T_k) \leq \min(\prod_{s=1}^{k} n_s, \prod_{s=k+1}^{d} n_s) \leq \min(n^k, n^{d-k}) = n^{d/2 - |d/2 - k|}. \tag{2.3.2}$$

□

Hiernach haben wir am Rand der Tensor-Train-Darstellung nach der Truncation die Bonddimensionen $r_0 = r_d = 1$, was mit der Definition des Formats übereinstimmt. Mit jedem Schritt ins Innere der Tensor-Train-Darstellung erhöht sich der mögliche Wert für die Bonddimension um den Faktor n. Aus Satz 2.3.12 ergibt sich direkt folgende Folgerung.

Folgerung 2.3.13. *Der maximal mögliche Wert für eine Bonddimension der Truncation $\boldsymbol{B}$ ist $n^{\lfloor d/2 \rfloor}$. Dieser kann für d gerade nur von der mittleren Bonddimension $r_{d/2}$ und für d ungerade von den beiden mittleren Bonddimensionen $r_{(d+1)/2}$ und $r_{(d-1)/2}$ angenommen werden.*

Im Folgenden schreiben wir r_{worst} für diese Worst-Case-Bonddimension $r_{worst} = n^{\lfloor d/2 \rfloor}$.

Zusammengefasst können wir im TT-Format jeden Tensor mit hoher Genauigkeit approximieren. Allerdings erhalten wir für $r = r_{worst}$ wieder eine exponentielle Abhängigkeit von d.

3 Eigenwertlöser

Wir betrachten in dieser Arbeit hoch-dimensionale lineare Eigenwertprobleme. Iterative Verfahren benötigen nur das Ergebnis eines linearen Operators angewandt auf einen gegebenen Vektor sowie Vektoroperationen, um das Problem zu lösen. Für gewöhnlich wird der lineare Operator als dünnbesetzte Matrix gespeichert. Diesen Fall wollen wir im Folgenden betrachten.

Obwohl wir uns in dieser Arbeit mit reellen Operatoren befassen, verwenden wir hier die allgemeinere Formulierung in $\mathbb{C}$. Für die symmetrischen Operatoren können wir diese o.B.d.A. auf $\mathbb{R}$ beschränken. Ist der Operator unsymmetrisch, so können komplex-konjugierte Eigenpaare auftreten. Hier verwenden wir dann die komplexe Formulierung. Bei der Eigenwertproblematik im TT-Format, auf welche wir in Abschnitt 3.2 genauer eingehen, betrachten wir einen Algorithmus, welcher nur symmetrische Probleme löst. Demnach können wir uns dort auf den reellen Fall beschränken.

Beim Eigenwertproblem werden für eine gegebene Matrix $A \in \mathbb{C}^{n\times n}$ Eigenwerte (EW) $\lambda_i \in \mathbb{C}$ und zugehörige Eigenvektoren (EV) $v_i \in \mathbb{C}^n$ gesucht, sodass die folgende Gleichung erfüllt ist:

$$Av_i = \lambda_i v_i. \tag{3.0.1}$$

Zum Lösen des Eigenwertproblems bei kleinen und mittleren Problemgrößen werden für gewöhnlich Matrixzerlegungen verwendet. Hierdurch erhält die Matrix eine Form, mit der die Eigenwerte und Eigenvektoren leichter zu bestimmen sind; [15, Kapitel 1.8] beschäftigt sich mit diesem Thema. Ein Beispiel für eine Matrixzerlegung ist die Schurzerlegung.

Definition 3.0.1 (Schurzerlegung)**.** *Sei $A \in \mathbb{C}^{n\times n}$ eine Matrix. Die Zerlegung der Form*

$$AQ = QR$$

mit einer unitären Matrix $Q \in \mathbb{C}^{n\times n}$ und einer oberen Dreiecksmatrix $R \in \mathbb{C}^{n\times n}$ nennen wir Schurzerlegung von A.

Für eine Schurzerlegung gilt, dass die Diagonaleinträge von R die Eigenwerte von A sind ($R_{i,i} = \lambda_i$). Ist A normal, das heißt $A^H A = AA^H$, dann ist R eine Diagonalmatrix und die Spalten von Q sind die Eigenvektoren von A. Eine solche Schurzerlegung kann mittels sukzessiven QR-Zerlegungen mit $\mathcal{O}(n^3)$ Operationen berechnet werden (siehe [16, Kapitel 7.4 und 7.5]). Ist n groß, so sind die Kosten der Schurzerlegung oder auch die Kosten anderer Matrixzerlegungen zu hoch und es müssen andere Methoden betrachtet werden.

Im Folgenden geben wir basierend auf [17, S. 133-136] und [18, S. 148-149] eine kurze Einführung in die Projektionsmethode. Diese bildet die Grundlage der Jacobi-Davidson-Methode (Jada), mit welcher wir uns in dieser Arbeit beschäftigen wollen.

Für einen genaueren Einblick in die Eigenwertproblematik siehe [15], [18] oder [16].

3.1 Projektionsmethode

Viele Standardverfahren zur Berechnung von großen Eigenwertproblemen gehören zur Gruppe der Projektionsmethoden. Diese haben zum Ziel, eine gute Approximation der Eigenvektoren von A in einem k-dimensionalen Unterraum $\mathcal{W} \subset \mathbb{C}^n$, dem sogenannten Suchraum, zu finden.

Hierzu wird das Problem unter einer Petrov-Galerkin-Bedingung auf den Suchraum $\mathcal{W}$ projiziert, sodass das Residuum im orthogonalen Kompliment eines zweiten k-dimensionalen Unterraums $\tilde{\mathcal{W}} \subset \mathbb{C}^n$, dem sogenannten Testraum, liegt. Das heißt, es wird eine Lösung $\lambda \in \mathbb{C}$ und $v \in \mathcal{W}$ gesucht, welche folgende Bedingung erfüllt:

$$Av - \lambda v \perp \tilde{\mathcal{W}}.$$

Für den Fall $\tilde{\mathcal{W}} = \mathcal{W}$, welchen wir im Folgenden betrachten wollen, erhält man die Ritz-Galerkin-Bedingung

$$Av - \lambda v \perp \mathcal{W}. \tag{3.1.1}$$

Stellt man v in der orthogonalen Basis $W = (w_1\, w_2\, \ldots\, w_k)$ des Unterraums $\mathcal{W}$ dar, das heißt $v = Ws$ mit $s \in \mathbb{C}^k$, so kann die Ritz-Galerkin-Bedingung (3.1.1) zu folgender Gleichung umformuliert werden:

$$\begin{aligned} & W^*(AWs - \lambda Ws) = 0 \\ \Leftrightarrow \qquad & (W^*AW)s - \lambda s = 0. \end{aligned} \tag{3.1.2}$$

Gleichung (3.1.2) ergibt ein Eigenwertproblem kleinerer Dimension der Matrix $H = W^*AW \in \mathbb{C}^{k\times k}$, welches kostengünstig durch eine Schurzerlegung lösbar ist. Aus dem Eigenpaar (λ_i, s_i) von H können die Ritzwerte und Ritzvektoren (λ_i, Ws_i) von A bestimmt werden, welche eine Approximation der Eigenpaare von A liefern.

Oft wird die Projektionsmethode in Kombination mit einer Unterraum-Iteration verwendet. Hierbei wird die Projektionsmethode mehrmals hintereinander ausgeführt, wobei sich in jeder Iteration der Suchraum $\mathcal{W}$ abhängig von der aktuellen Approximation der Eigenpaare ändert. Dies soll die Approximation schrittweise verbessern.

Der Unterschied zwischen verschiedenen Projektionsmethoden liegt hauptsächlich in der Wahl des Suchraum $\mathcal{W}$ beziehungsweise in der Berechnung von Basisvektoren dieses Raumes. Bekannte Projektionsverfahren wie Lanczos und Arnoldi verwenden Krylov-Unterräume $K_k = \{x, Ax, A^2x, \ldots, A^{k-1}x\}$ für die Projektion (siehe [15, S.125]). Die Idee hierbei ist, dass mit wachsendem k A^kx zum dominanten Eigenvektor konvergiert, das heißt zum Eigenvektor mit dem betragsmäßig größten Eigenwert $|\lambda_i|$ (siehe [18, S.149]).

Auch der Jacobi-Davidson-Eigenwertlöser gehört zur Gruppe der Projektionsverfahren. Dieser bestimmt die nächste Suchrichtung durch das Lösen einer linearen Gleichung. Eine hinreichend genaue Lösung dieser führt zu quadratischer Konvergenz für allgemeine Matrizen (siehe [19]). Dieses Verfahren wollen wir im nächsten Abschnitt genauer betrachten.

3.1.1 Jacobi-Davidson-Methode (Jada)

Beim Jacobi-Davidson-Verfahren wird der Unterraum $\mathcal{W}$ mittels einem inexaktem Newton-Prozess konstruiert. Die Idee hierfür hatten Sleijpen und Van der Vorst in [5]. Eine schöne Herleitung ist in [15, Kapitel 8.4] zu finden. Diese fassen wir im Folgenden zusammen.

Sei (σ, v) eine Approximation eines Eigenpaares von A. Dann ist das Ziel, Korrekturen $\delta\sigma$ und δv von σ und v zu finden, welche die folgende Gleichung erfüllen:

$$A(v + \delta v) = (\sigma + \delta\sigma)(v + \delta v). \tag{3.1.3}$$

Vernachlässigung des quadratischen Terms $\delta\sigma \cdot \delta v$ führt zur Korrekturgleichung

$$(A - \sigma I)\delta v - \delta\sigma v = -res, \tag{3.1.4}$$

wobei $res = (A - \sigma I)v$ das Residuum ist.

Da das System unterbestimmt ist, wird noch die Bedingung $v^*\delta v = 0$ hinzugefügt, welche dafür sorgt, dass unser neuer Vektor $v + \delta v$ normiert ist. Damit ist folgendes Gleichungssystem zu lösen:

$$\begin{pmatrix} A - \sigma I & -v \\ v^* & 0 \end{pmatrix} \begin{pmatrix} \delta v \\ \delta\sigma \end{pmatrix} = \begin{pmatrix} -res \\ 0 \end{pmatrix}. \tag{3.1.5}$$

Die selbe Formulierung erhält man auch mit einem Newton-Verfahren für das Optimierungsproblem

$$\min_{\|v\|_2^2=1} \frac{1}{2}\|(A - \nu I)v\|_2^2.$$

Für das entsprechende Gleichungssystem aus der Eigenwertgleichung $(A-\nu I)u = 0$ und der Orthonormalitätsbedingung $\frac{1}{2}u^*u - \frac{1}{2} = 0$ erhält man durch einen Newton-Schritt mit den Unbekannten u und ν und den momentanen Approximationen v und σ folgendes System:

$$\begin{pmatrix} u_{neu} \\ \nu_{neu} \end{pmatrix} = \begin{pmatrix} v \\ \sigma \end{pmatrix} - \begin{pmatrix} (A - \sigma I) & -v \\ v^* & 0 \end{pmatrix}^{-1} \begin{pmatrix} res \\ 0 \end{pmatrix}. \tag{3.1.6}$$

Mit $u_{neu} = v + \delta v$ und $\nu_{neu} = \sigma + \delta\sigma$ können wir dies umformulieren zu (3.1.5).

(3.1.5) kann auch als folgendes Problem betrachtet werden:

$$\text{Finde } \delta v = (A - \sigma I)^{-1}(-res + \delta\sigma v), \text{ sodass } v^*\delta v = 0. \tag{3.1.7}$$

Liegt σ nah an einem Eigenwert λ von A, so ist $(A - \sigma I)$ schlecht konditioniert. Um die Kondition des linearen Problems zu verbessern, kann man Projektionen nutzen. Projiziert man den zum Eigenwert gehörenden Eigenvektor raus, erhält man ein Problem ohne den nahezu singulären Eigenwert $\lambda - \sigma$ von $(A - \sigma I)$. Diesen Effekt erhofft man sich auch, wenn man nur eine Approximation dieses Eigenvektors hat.

Angenommen v sei normiert ($\|v\|_2 = 1$). Dann ist die orthogonale Projektion auf v $P_v = I - vv^*$ und es gilt $P_v v = 0$. Durch die Ritz-Galerkin-Bedingung (3.1.1) gilt $res \perp v$ und somit $P_v res = res$. Multiplizieren der Gleichung (3.1.4) mit P_v ergibt:

$$P_v(A - \sigma I)\delta v = -res. \tag{3.1.8}$$

Die Nebenbedingung des Problems (3.1.7) sagt aus, dass $\delta v \perp v$, womit δv in der Gleichung durch $P_v \delta v$ ersetzt werden kann und das Gleichungssystem reduziert sich zu folgender Gleichung:

$$P_v(A - \sigma I)P_v\delta v = -res, \tag{3.1.9}$$

wobei $P_v\delta v$ die gesuchte Lösung ist. Dies ergibt die Jacobi-Davidson-Korrekturgleichung für v:

$$(I - vv^*)(A - \sigma I)(I - vv^*)\delta v = -res. \tag{3.1.10}$$

Das Jacobi-Davidson-Verfahren ist ein Projektionsverfahren, welches die Lösung der Korrekturgleichung für ein approximiertes Eigenpaar (σ, v) der aktuellen Iteration verwendet, um den Suchraum in der nächsten Iteration zu erweitern. Fügen wir die Korrektur δv des betrachteten approximierten Eigenvektors v dem Suchraum hinzu, so können wir in der nächsten Iteration eine bessere Approximation dieses Eigenpaars finden. Hierdurch konvergiert in der Regel jedoch nur das in der Korrekturgleichung betrachtete Eigenpaar. Um mehrere Eigenpaare einer Matrix A zu berechnen, kann man den Jacobi-Davidson-QR-Algorithmus [20] verwenden. Dieser verläuft sukzessiv für die verschiedenen Eigenvektoren.

Die Berechnung des ersten Eigenpaares geschieht analog zum Jacobi-Davidson-Verfahren. Für die Berechnung der weiteren Eigenpaare nehmen wir an, dass Q die bereits konvergierten Eigenvektoren von A enthält. Diese projizieren wir durch $\tilde{A} = (I - QQ^*)A(I - QQ^*)$ aus unserem Problem heraus. Wenden wir nun den Jacobi-Davidson-Algorithmus auf $\tilde{A}$ an, so berechnet dieser ein noch nicht konvergiertes Eigenpaar (σ, v) von A. Die zu lösende Korrekturgleichung (3.1.10) für den Eigenvektor v verändert sich zu

$$(I - \tilde{Q}\tilde{Q}^*)(A - \sigma I)(I - \tilde{Q}\tilde{Q}^*)\delta v = -r\tilde{e}s, \tag{3.1.11}$$

wobei $\tilde{Q} = [Q, v]$ und $r\tilde{e}s = (I - QQ^*)(A - \sigma I)(I - QQ^*)v$. Außerdem soll δv die Orthogonalitätsbedingung $\tilde{Q}^*\delta v = 0$ erfüllen.

Zusätzlich ist zu beachten, dass bei der Berechnung der vielen Eigenpaare der Suchraum W immer weiter anwächst, was einen Restart erforderlich macht. Um W zu verkleinern ohne die wichtigen Informationen zu verlieren, wird der Suchraum gefiltert. Hierzu sortieren man die Schurzerlegung $HQ_H = Q_H R_H$ der Matrix H des projizierten Eigenwertproblems nach der für die Eigenwerte gewünschte Eigenschaft. Suchen wir

beispielsweise die Eigenwerte mit kleinstem Realteil, so sortieren wir die Ritzwerte aufsteigend nach ihrer Größe $\mathrm{Re}(R_H(1,1)) \leq \cdots \leq \mathrm{Re}(R_H(k,k))$ für $k = \dim(W)$. Soll der Suchraum auf $i \leq k$ Suchrichtungen reduziert werden, so wird $W = WQ_H(:, 1:i)$ gesetzt. Für nähere Details zum Jacobi-Davidson-QR-Algorithmus siehe [20, Kapitel 2].

Wir betrachten in unseren numerischen Experimenten stets nur einen Eigenwert. Für uns ergibt sich somit Algorithmus 1, dessen allgemeinen Ablauf wir im Folgenden kurz beschreiben. Er beruht auf einem Innen-Außen-Iterations-Schema.

Der Algorithmus startet mit einem leeren Suchraum. In der äußeren Iteration wird die neue Suchrichtung t bezüglich des Suchraums W orthonormalisiert und dann dem Suchraum hinzugefügt. Die Projektion H der Matrix A wird bezüglich des vergrößerten Suchraums aktualisiert, bevor eine sortierte Schurzerlegung genutzt wird, um die Ritzwerte und Ritzvektoren von A zu berechnen. Mit Hilfe des Residuums prüfen wir, wie gut die bisherige Approximation des Eigenpaars ist. Wurde die gewünschte Toleranz für das Residuum noch nicht erreicht, so kommen wir zur inneren Iteration, welche mittels des GMRES-Verfahrens die Korrekturgleichung (3.1.10) löst, wobei hier Q unsere Approximation des Eigenvektors ist. Die berechnete Korrektur t dient dann in der nächsten äußeren Iteration als neue Suchrichtung.

Algorithm 1 Jacobi-Davidson-QR-Algorithmus für ein Eigenpaar

Input: Matrix A, Startvektor t, Toleranz tol_{Jada} für das Abbruchkriterium
Output: Eigenpaar (Q, σ) von A

1: $H = \emptyset, W = \emptyset, W_A = \emptyset$
2: **for** $nIter = 1, \ldots, maxIter$ **do**
3: $\quad v = (I - WW^*)t / \|(I - WW^*)t\|_2$
4: $\quad v_A = Av$
5: $\quad H = \begin{pmatrix} H & W^*v_A \\ v^*W_A & v^*v_A \end{pmatrix}$
6: $\quad W = (W, v), W_A = (W_A, v_A)$
7: $\quad$ Berechne sortierte Schur-Zerlegung $HQ_H = Q_H R_H$
8: $\quad \sigma = R_H[0,0], Q = WQ_H[:, 0]$
9: $\quad res = AQ - Q\sigma$
10: $\quad$ **if** $\|res\|_2 < tol_{Jada}$ **then return** (Q,σ)
11: $\quad$ **end if**
12: $\quad$ // Löse Korrekturgleichung approximativ
13: $\quad t = \mathrm{GMRES}((I - QQ^*)(A - \sigma I)(I - QQ^*), -res)$
14: $\quad t = (I - QQ^*)t$
15: **end for**

3.2 Eigenwertproblem im TT-Format

Ist unser Operator $\boldsymbol{A}$ im TT-Format gegeben, so sind Verfahren basierend auf dem alternierenden linearen Schema (ALS) [8] üblich, um das Eigenwertproblem

$$\boldsymbol{A} \cdot \boldsymbol{U} = \lambda \cdot \boldsymbol{U} \tag{3.2.1}$$

zu lösen, wobei wir Eigenvektoren $\boldsymbol{U}$ im TT-Format suchen. Ist Gleichung (3.2.1) erfüllt, so gilt ebenfalls $\mathrm{mat}(\boldsymbol{A}) \cdot \mathrm{vek}(\boldsymbol{U}) = \lambda \cdot \mathrm{vek}(\boldsymbol{U})$, was Gleichung (3.0.1) entspricht.

Für unsere numerischen Experimente verwenden wir den eigb-Algorithmus aus dem Python-Paket ttpy, welcher in [9] beschrieben wird. Dieser Eigenwertlöser basiert auf dem ALS-Verfahren und kann mittels eines Block-TT-Formats mehrere Eigenwerte gleichzeitig berechnen. Dies spielt bei uns jedoch keine Rolle, da wir in unseren numerischen Experimenten nur die Approximation eines Eigenwerts testen.

Das ALS-Verfahren dient dem Lösen von Optimierungsproblemen im TT-Format. Der eigb-Algorithmus betrachtet als Optimierungsproblem die Minimierung des Rayleigh-Quotienten $R_A(v) = \frac{v^* A v}{v^* v}$ für die Matrix $A = \mathrm{mat}(\boldsymbol{A})$. Der Satz von Courant-Fischer zeigt die Beziehung zwischen dem Rayleigh-Quotienten und den Eigenwerten einer symmetrischen Matrix.

Satz 3.2.1 (Satz von Courant-Fischer)**.** *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix mit Eigenwerten $\lambda_1 \leq \cdots \leq \lambda_n$. Sei X_k die Menge der k-dimensionalen Untervektorräume von $\mathbb{R}^n$. Dann ist folgende Gleichung erfüllt:*

$$\lambda_k = \max_{X \in X_{n-k+1}} \min_{x \neq 0 \in X} R_A(x) = \min_{X \in X_k} \max_{x \neq 0 \in X} R_A(x).$$

Beweis. Siehe [17, Theorem 1.21]. □

Folgerung 3.2.2. *Für eine symmetrische Matrix $A \in \mathbb{R}^{n\times n}$ mit Eigenwerten $\lambda_1 \leq \cdots \leq \lambda_n$ gilt $\lambda_1 = \min_{x\neq 0} R_A(x)$ und $\lambda_n = \max_{x\neq 0} R_A(x)$.*

Beweis. Siehe siehe [17, S. 26]. □

Die Minimierung des Rayleigh-Quotienten ist im symmetrischen Fall also äquivalent zu der Suche nach dem kleinsten Eigenwert. Da dies für unsymmetrische Matrizen nicht der Fall ist, kann der eigb-Algorithmus unsymmetrische Eigenwertprobleme nicht lösen.

Im Folgenden gehen wir genauer auf den Ablauf des ALS-Verfahrens aus [8] ein.

3.2.1 Alternierendes Lineares Schema (ALS)

Das alternierende lineare Schema löst Optimierungsprobleme für Tensoren im TT-Format $\boldsymbol{U} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \bigotimes_{i=1}^{d} \boldsymbol{U}^{(i)}_{k_{i-1},:,k_i}$ mit TT-Kernen $\boldsymbol{U}^{(i)} \in \mathbb{R}^{r_{i-1}\times n_i \times r_i}, i = 1, \ldots, d$.

Die Strategie hierbei ist es, alle bis auf einen TT-Kern festzuhalten und den freien TT-Kern zu optimieren. Führen wir dies nacheinander für alle TT-Kerne durch, so reduziert man das multi-lineare Problem auf mehrere lineare Probleme im jeweils freien Index n_i ([8, S. 2]).

Der Algorithmus läuft in sogenannten Sweeps ab. Eine graphische Darstellung eines halben Sweeps ist in Abbildung 3.1 zu sehen. Diese dient zur Veranschaulichung der folgenden Beschreibung.

Zu Beginn eines Sweeps sind die TT-Kerne $\boldsymbol{U}^{(2)}, \dots, \boldsymbol{U}^{(d)}$ rechts-orthogonal.

Es wird mit dem nicht-orthogonalen TT-Kern $\boldsymbol{U}^{(1)}$ begonnen. Dieser wird bezüglich des Minimierungsproblems optimiert, während die anderen TT-Kerne festgehalten werden. Das bedeutet, wir berechnen den TT-Kern $\boldsymbol{V}^{(1)}$, für den das betrachtete Optimierungsproblem unter allen TT-Tensoren der Form $\sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{V}^{(1)}_{k_0,:,k_1} \otimes \boldsymbol{U}^{(2)}_{k_1,:,k_2} \otimes \cdots \otimes \boldsymbol{U}^{(d)}_{k_{d-1},:,k_d}$ minimiert wird.

Der optimierte TT-Kern $\boldsymbol{V}^{(1)}$ kann durch eine QR-Zerlegung $\boldsymbol{V}^{(1)} = \tilde{\boldsymbol{U}}^{(1)} R^{(1)}$ in einen links-orthogonalen Teil $\tilde{\boldsymbol{U}}^{(1)}$ und eine obere Dreiecksmatrix $R^{(1)}$ zerlegt werden.

Setzt man nun $\boldsymbol{U}^{(1)} = \tilde{\boldsymbol{U}}^{(1)}$, erhält man einen links-orthogonalen optimierten ersten TT-Kern. Die Dreiecksmatrix $R^{(1)}$, in Abbildung 3.1 dargestellt durch den kleinen Punkt zwischen den TT-Kernen, wird zum nächsten TT-Kern geschoben $\boldsymbol{U}^{(2)} = R^{(1)} \boldsymbol{U}^{(2)}$, welcher nun nicht mehr orthogonal ist.

Die für $\boldsymbol{U}^{(1)}$ erfolgten Schritten werden nun der Reihe nach für die TT-Kerne $\boldsymbol{U}^{(2)}, \dots, \boldsymbol{U}^{(d-1)}$ durchgeführt. Dies ergibt den halben Sweep, an dessen Ende die TT-Kerne $\boldsymbol{U}^{(1)}, \dots, \boldsymbol{U}^{(d-1)}$ links-orthogonal und optimiert sind, der nicht-orthogonale TT-Kern $\boldsymbol{U}^{(d)}$ jedoch noch nicht optimiert wurde.

Nun wird der halbe Sweep in umgekehrter Richtung für die TT-Kerne $\boldsymbol{U}^{(d)}, \dots, \boldsymbol{U}^{(2)}$ durchgeführt. Am Ende des gesamten Sweeps sind die TT-Kerne $\boldsymbol{U}^{(2)}, \dots, \boldsymbol{U}^{(d)}$ wieder rechts-orthogonal und alle TT-Kerne wurden optimiert.

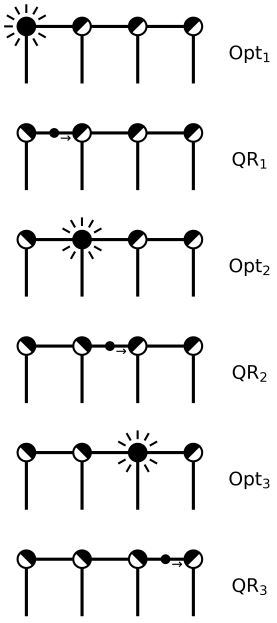


Abbildung 3.1: Graphische Darstellung eines halben Sweeps des ALS-Verfahrens

Ist die gewünschte Genauigkeit noch nicht erreicht, führt man weitere Sweeps durch, um die TT-Kerne weiter zu optimieren.

Um die rechts-Orthogonalität beim ersten Sweep zu gewährleisten, muss zu Beginn des Algorithmus eventuell eine Orthogonalisierung durchgeführt werden.

4 Tensor-Train-Jacobi-Davidson-Methode (TT-Jada)

In diesem Kapitel beschäftigen wir uns mit der Verwendung des Tensor-Train-Formats im Jacobi-Davidson-Algorithmus.

Im Tensor-Train-Format können wir die gleichen Operationen durchführen wie mit einer dünnbesetzten Matrix und Vektoren. Der Unterschied liegt im Aufwand und der Genauigkeit der Operationen in den verschiedenen Darstellungen.

Verwenden wir dünnbesetzte Matrizen und Vektoren, so ist der Rechenaufwand exponentiell Abhängig von der Dimension des Problems. Das Tensor-Train-Format ist so konzipiert, dass es diese exponentielle Abhängigkeit des Aufwands bei kleiner Bonddimension vermeiden kann (siehe Kapitel 5). Aufgrund dessen erhoffen wir uns von der Verwendung des Tensor-Train-Formats eine Laufzeitersparnis für große Dimensionen.

Zu einem Problem könnten die verschiedenen Genauigkeiten der Operationen werden. Bei Arrays mit Fließkommazahlen schneiden wir beim Runden entsprechende Nachkommastellen ab. Hierdurch erhält man einen relativen Fehler in allen einzelnen Einträgen. Beim TT-Format nutzen wir hierfür Truncations, welche auf abgeschnittenen Singulärwertzerlegungen basieren. Hierbei bezieht sich die Toleranz auf die Frobeniusnorm des gesamten TT-Vektors und sie ist üblicherweise gröber gewählt, als die Toleranz von $\sim 10^{-15}$, welche man für doppelt genaue Fließkommazahlen erhält. Die Robustheit vieler Algorithmen ist bezüglich der Gleitkommaarithmetik analysiert. Verwenden wir nun TT-Operationen, so ist nicht klar, welche Auswirkungen der andere Genauigkeitsbegriff auf die Robustheit des Verfahrens hat.

Um die Auswirkungen der Truncation einzudämmen, wollen wir das Tensor-Train-Format nur in der inneren Iteration des Jacobi-Davidson-Verfahrens verwenden, während in der äußeren Iteration weiter mit der dünnbesetzten Matrix und Vektoren gerechnet wird.

Der Grund hierfür ist, dass wir in der äußeren Iteration möglichst genaue Projektionen und Orthogonalisierungen benötigen, da die Genauigkeit der Ergebnisse des Algorithmus empfindlich von der Orthogonalität der Basisvektoren der äußeren Iteration abhängt. Dies wäre mit der benötigten Truncation schwer umzusetzen. Die innere Iteration löst eine Korrekturgleichung, um eine neue Suchrichtung zu liefern. Da wir die Lösung dieser Gleichung nur approximativ benötigen, sollte die andere Genauigkeit im Tensor-Train-Format hier keinen zu großen Effekt auf das Verfahren insgesamt haben.

Das Lösen der Korrekturgleichung in der inneren Iteration des Jacobi-Davidson-Verfahrens übernimmt der GMRES-Algorithmus. Die Verwendung des TT-Formats an dieser Stelle bedeutet konkret, dass wir hier nun die TT-Variante dieses Verfahrens, das heißt den TT-GMRES-Algorithmus aus [6], als Gleichungslöser verwenden.

Die Verwendung von beiden Darstellungen im Jacobi-Davidson-Verfahren bedeutet, dass der Algorithmus sowohl den Operator als dünnbesetzte Matrix als auch die TT-Darstellung des Operators benötigt. Hierdurch können wir nicht von dem geringeren Speicherverbrauch des TT-Operators (siehe Kapitel 5) profitieren. Deshalb werden wir noch eine weitere Variante des TT-Jacobi-Davidson-Algorithmus betrachten, welche zwar noch Vektoren in der äußeren Iteration verwendet, allerdings die dünnbesetzte Matrix nicht benötigt. Hierdurch wollen wir die Berechnung größerer Probleme ermöglichen.

Für die Programmierung verwenden wir das Python-Paket ttpy [7], welches eine Implementation des Tensor-Train-Formats und der wichtigsten Operationen für dieses Format enthält.

Im Folgenden erläutern wir kurz das GMRES-Verfahren und die Unterschiede im TT-GMRES-Verfahren. Anschließend gehen wir darauf ein, was wir beim Aufruf des TT-GMRES-Verfahrens im Jacobi-Davidson-Algorithmus zu beachten haben und stellen verschiedene Varianten unseres Tensor-Train-Jacobi-Davidson-Verfahrens (TT-Jada) vor, welche wir in unseren Experimenten in Kapitel 6 verwenden.

4.1 GMRES- und TT-GMRES-Verfahren

Das GMRES-Verfahren (generalized minimal residuum) löst ein lineares Gleichungssystem $Ax = b$, indem die Norm des Residuums $\|b - Ax\|_2$ im Krylov-Unterraum $K_k = \{b, Ab, A^2b, ..., A^{k-1}b\}$ minimiert wird.

Hierfür wird erst eine orthogonale Basis V_k von K_k und die Matrix $\bar{H}_k$ mit $AV_k = V_{k+1}\bar{H}_k$ berechnet. Nun soll eine Korrektur der Startlösung x_0 im Krylov-Unterraum berechnet werden, welche das Residuum in der euklidischen Norm minimiert. Dies führt zum Least-Squares-Problem $\min_{z \in K_k} \|res_0 - Az\|_2$ mit $res_0 = b - Ax_0$.

Mit der Darstellung $z = V_k y$ und einigen Umformulierungen, ergibt sich die folgende Gleichung für die Korrektur der Startlösung:

$$x_k = x_0 + V_k y, \quad y = \underset{y}{\operatorname{argmin}} \|\beta e^{(1)} - \bar{H}_k y\|_2, \tag{4.1.1}$$

wobei $\beta = \|res_0\|_2$ und $e^{(1)}$ der erste Einheitsvektor ist. Eine genauere Herleitung kann in [17, Kapitel 6.5] gefunden werden.

In dieser Arbeit wollen wir das bisher im Jacobi-Davidson-Algorithmus verwendeten GMRES-Verfahren (vergleiche Algorithmus 1) durch das TT-GMRES-Verfahren aus [6] ersetzen.

Das TT-GMRES-Verfahren verläuft analog zum GMRES-Verfahren. Der Unterschied liegt hauptsächlich in der Verwendung des Tensor-Train-Formats. Im TT-GMRES-Verfahren sind $\boldsymbol{A}$, $\boldsymbol{x}$ und $\boldsymbol{b}$ im TT-Format und auch die Operationen werden in diesem Format durchgeführt. Hierdurch sind auch die Zwischenergebnisse innerhalb des Algorithmus im Tensor-Train-Format gespeichert.

Anstatt dem Tensoroperator $\boldsymbol{A}$ wird dem TT-GMRES-Algorithmus eine Funktion $\mathcal{A}(\boldsymbol{x}[, \epsilon]) \rightarrow \boldsymbol{y}$ übergeben, welche das Matrix-Vektor-Produkt $\boldsymbol{Ax}$ im TT-Format mit

Algorithm 2 TT-GMRES-Algorithmus

Input: Operator $\mathcal{A}(\boldsymbol{x}[,\epsilon])$ zur Berechnung von $\boldsymbol{Ax}$ mit Toleranz ϵ für die Truncation-Genauigkeit , Startvektor $\boldsymbol{x_0}$ und rechte Seite $\boldsymbol{b}$ im TT-Format, Toleranz tol_{GMRES} für das Abbruchkriterium (optional: maximale Bonddimension R)

Output: Lösung $\boldsymbol{x}$ von $\boldsymbol{Ax} = \boldsymbol{b}$ im TT-Format

1: $\boldsymbol{res}_0 = \mathcal{T}_{tol_{GMRES},R}(\boldsymbol{b} - \mathcal{A}(\boldsymbol{x}_0)), \beta = \|\boldsymbol{res}_0\|_2, \boldsymbol{\nu}_1 = \boldsymbol{res}_0/\beta$
2: **for** j =1,. . . , m **do**
3: $\quad\delta = \frac{tol_{GMRES}}{\|\bar{\boldsymbol{res}}_{j-1}\|_2/\beta}$
4: $\quad\boldsymbol{w} = \mathcal{A}(\boldsymbol{v}_j, \delta)$
5: $\quad$**for** $i = 1, \ldots, j$ **do**
6: $\quad\quad h_{i,j} = (\boldsymbol{w}, \boldsymbol{\nu}_i), \boldsymbol{w} = \boldsymbol{w} - h_{i,j}\boldsymbol{\nu}_i$
7: $\quad\quad\boldsymbol{w} = \mathcal{T}_{\delta,R}(\boldsymbol{w})$
8: $\quad$**end for**
9: $\quad h_{j+1,j} = \|\boldsymbol{w}\|_2, \boldsymbol{\nu}_{j+1} = \boldsymbol{w}/h_{j+1,j}$
10: $\quad$Assembliere Matrix $\bar{H}_j = [h_{i,k}], k = 1, \ldots, j, i = 1, \ldots, j+1$
11: $\quad$Berechne $y_j = \text{argmin}_y \|\beta e^{(1)} - \bar{H}_j y\|_2$
12: $\quad$// Prüfe Residuum $\|\bar{\boldsymbol{res}}_j\|_2 = \|\beta e^{(1)} - \bar{H}_j y_j\|_2$
13: $\quad$**if** $\|\bar{\boldsymbol{res}}_j\|_2/\|\boldsymbol{b}\|_2 \leq tol_{GMRES}$ **then break**
14: $\quad$**end if**
15: **end for**
16: $\boldsymbol{x}_j = \boldsymbol{x}_0 + \sum_{i=1}^{j} y_j[i]\boldsymbol{\nu}_i$
17: $\boldsymbol{x}_j = \mathcal{T}_{tol_{GMRES},R}(\boldsymbol{x}_j)$
18: **if** $\|\bar{\boldsymbol{res}}_j\|_2/\|\boldsymbol{b}\|_2 > tol_{GMRES}$ **then** Setze $\boldsymbol{x}_0 = \boldsymbol{x}_j$ und starte in Zeile 1
19: **end if**

möglicher Truncation der Toleranz ϵ für die Genauigkeit berechnet, das heißt $\boldsymbol{y} = \mathcal{T}_\epsilon(\boldsymbol{Ax})$.

Da sich durch Additionen und Matrix-Vektor-Multiplikationen die Bonddimensionen von TT-Vektoren addieren beziehungsweise multiplizieren, ist es wichtig, im TT-GMRES-Verfahren nach jeder solchen Operation eine Truncation auszuführen, um die Kosten des TT-Formats gering zu halten. Die entscheidende Frage hierbei ist, wie klein die Toleranz der Truncation gewählt werden muss, um die Konvergenz des Verfahrens nicht zu stören.

Lemma 4.1.1. *Das TT-GMRES-Verfahren konvergiert, wenn in jeder Iteration die zum aktuellen Residuum $\boldsymbol{res}_{j-1}$ umgekehrt proportionale Toleranz*

$$\delta = \frac{tol_{GMRES}}{\|\boldsymbol{res}_{j-1}\|_2/\beta}$$

für die Truncation-Genauigkeit verwendet wird, wobei $\beta = \|\boldsymbol{res}_0\|_2$ und tol_{GMRES} die Toleranz für die im TT-GMRES-Verfahren zu erreichende Genauigkeit ist.

Beweis. Siehe [6]. □

Algorithmus 2 zeigt den Pseudocode für das TT-GMRES-Verfahren. In rot markiert sind die Änderungen, welche gegenüber dem normalen GMRES-Verfahren ausgeführt werden müssen. Dies sind die Truncations und die Berechnung der Truncation-Genauigkeit in jeder Iteration. Die fettgedruckten Buchstaben verdeutlichen wieder, dass wir hier nun mit Tensoren arbeiten.

4.2 Aufruf des TT-GMRES-Verfahrens im Jacobi-Davidson-Algorithmus

Kommen wir nun zur konkreten Umsetzung des Tensor-Train-Jacobi-Davidson-Algorithmus.

Verwendet man den GMRES-Algorithmus um im Jacobi-Davidson-Verfahren die Korrekturgleichung (3.1.10)

$$(I - QQ^*)(A - \sigma I)(I - QQ^*)t = -res$$

zu lösen, so können einfach die benötigten Vektoren und Matrizen zwischen den beiden Algorithmen ausgetauscht werden. Verwendet man nun den TT-GMRES-Algorithmus, so hat man das Problem, dass mit zwei verschiedenen Formaten gearbeitet wird. Auf Grund dessen muss man sich mit dem Aufruf des TT-GMRES-Algorithmus im Jacobi-Davidson-Verfahren beschäftigen.

Bei der zu lösenden Korrekturgleichung benötigt der TT-GMRES-Algorithmus die rechte Seite der Gleichung $-res$ und den Operator $(I - QQ^*)(A - \sigma I)(I - QQ^*)$ im TT-Format, um damit weiterzuarbeiten. Die Lösung t gibt uns der Algorithmus im TT-Format ($\boldsymbol{t}_{TT}$) zurück. Diese muss für die äußere Iteration des Jacobi-Davidson-Verfahrens in einen Vektor transformiert werden.

Betrachten wir zunächst die Formattransformationen eines Vektors v in seine TT-Darstellung $\boldsymbol{v}_{TT}$ und umgekehrt. Das Python-Paket ttpy enthält Routinen, um die TT-Zerlegung eines Tensors mittels sukzessiver, abgeschnittener SVDs zu berechnen (siehe Kapitel 2.3.2) und um aus der TT-Zerlegung durch das Ausrechnen der Format-Formel (2.3.1) wieder den vollen Tensor zu erzeugen. Fügen wir diesen eine Transformation vom Vektor zum Tensor oder umgekehrt vom Tensor zum Vektor hinzu, so kommen wir zu unseren beiden Routinen Vec2TT($v, n_{TT}[, \epsilon]$) und TT2Vec($\boldsymbol{v}_{TT}$), für die Formattransformationen zwischen Vektor und TT-Darstellung. n_{TT} gibt hierbei die gewünschten Moden des Tensors an, da diese nicht eindeutig aus der Länge des Vektors zu bestimmen sind.

Bei der Transformation vom Vektor ins TT-Format, wählt man eine Toleranz ϵ, damit die Bonddimensionen im Laufe des Algorithmus nicht zu sehr anwachsen. Wir verwenden hier die Toleranz $\epsilon = tol_{Jada} \cdot 1e - 2$, wobei tol_{Jada} die Toleranz der zu erreichenden Genauigkeit für die Residuumsnorm im Jacobi-Davidson-Algorithmus ist.

Mit diesen Routinen kann die Lösung $\boldsymbol{t}_{TT}$ des TT-GMRES-Verfahrens in einen Vektor umgewandelt werden.

Dies könnten wir auch verwenden, um die rechte Seite $-res$ ins TT-Format umzuwandeln. Bei unseren numerischen Experimenten ist uns allerdings aufgefallen, dass wir

Algorithm 3 Operator der Korrekturgleichung im TT-Format

Input: TT-Matrix $\boldsymbol{A}$, $\boldsymbol{x}$ und $\boldsymbol{Q}$ im TT-Format, Skalar σ, Toleranz ϵ für die Truncation-Genauigkeit (optional: maximale Bonddimension R)
Output: $\boldsymbol{sol} = (\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}^*)(\boldsymbol{A} - \sigma\boldsymbol{I})(\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}^*)\boldsymbol{x}$ im TT-Format

function CorrFunc($\boldsymbol{A}, \boldsymbol{x}, \boldsymbol{Q}, \sigma, \epsilon = 1e-14$)
  $y = \text{tt.dot}(\boldsymbol{Q}, \boldsymbol{x})$
  $\boldsymbol{z} = \boldsymbol{x} - \boldsymbol{Q}y$
  $\boldsymbol{z} = \mathcal{T}_{\epsilon,R}(\boldsymbol{z})$
  $\boldsymbol{sol} = \text{tt.matvec}(\boldsymbol{A}, \boldsymbol{z}) - \sigma * \boldsymbol{z}$
  $\boldsymbol{sol} = \mathcal{T}_{\epsilon,R}(\boldsymbol{sol})$
  $y = \text{tt.dot}(\boldsymbol{Q}, \boldsymbol{sol})$
  $\boldsymbol{sol} = \boldsymbol{sol} - \boldsymbol{Q}y$
  return $\mathcal{T}_{\epsilon,R}(\boldsymbol{sol})$
end function

Algorithm 4 Tensor-Train-Jacobi-Davidson-Algorithmus

Input: Matrix A und die TT-Darstellung $\boldsymbol{A}_{TT}$, Startvektor t, Toleranz tol_{Jada} für das Abbruchkriterium
Output: Eigenpaar (Q, σ) von A

$H = \emptyset, W = \emptyset, W_A = \emptyset$
for $nIter = 1, \ldots, maxIter$ **do**
  $v = (I - WW^*)t / \|(I - WW^*)t\|_2$
  $v_A = Av$
  $H = \begin{pmatrix} H & W^*v_A \\ v^*W_A & v^*v_A \end{pmatrix}$
  $W = (W, v), W_A = (W_A, v_A)$
  Berechne sortierte Schur-Zerlegung $HQ_H = Q_H R_H$
  $\sigma = R_H[0,0], Q = WQ_H[:,0]$
  $\boldsymbol{Q}_{TT} = \text{Vec2TT}(Q)$
  $\boldsymbol{res}_{TT} = \mathcal{T}_{\epsilon,R}(\boldsymbol{A}_{TT}\boldsymbol{Q}_{TT} - \boldsymbol{Q}_{TT}\sigma)$
  if $\|\boldsymbol{res}_{TT}\|_2 < tol_{Jada}$ **then return** (Q, σ)
  end if
  // Löse Korrekturgleichung approximativ
  $\boldsymbol{t}_{TT} = \text{TTGMRES}(\text{CorrFunc}(\boldsymbol{A}_{TT}, \boldsymbol{x}, \boldsymbol{Q}_{TT}, \sigma, \epsilon), -\boldsymbol{res}_{TT})$
  $t = \text{TT2Vec}(\boldsymbol{t}_{TT})$
  $t = (I - QQ^*)t$
end for

für das Residuum im TT-Format in der Regel niedrigere Bonddimension erhalten, wenn wir das Residuum direkt im TT-Format berechnen, als wenn wir das Residuum mit der dünnbesetzten Matrix und dem Vektor berechnen und dann das Ergebnis ins TT-Format transformieren.

Da wir möglichst niedrige Bonddimensionen im TT-Format haben wollen, nutzen wir dies hier und berechnen das Residuum im TT-Format, das heißt $\boldsymbol{res}_{TT} = \boldsymbol{A}_{TT}\boldsymbol{Q}_{TT} - \boldsymbol{Q}_{TT}\sigma$. Hierzu müssen wir nur Q mit unseren Routinen ins TT-Format transformieren. $\boldsymbol{A}_{TT}$ haben wir gegeben und σ ist ein Skalar, welches wir nicht umwandeln müssen. Hier benötigen wir auch noch eine Truncation, da wir eine Addition und eine Matrix-Vektor-Multiplikation im TT-Format ausführen.

Setzen wir uns nun mit dem Operator $(I - QQ^*)(A - \sigma I)(I - QQ^*)$ auseinander. Der TT-GMRES-Algorithmus erwartet hier eine Funktion $\mathcal{A}(\boldsymbol{x}[, \epsilon])$ mit $\boldsymbol{x}$ im TT-Format, wobei auch innerhalb der Funktion im TT-Format gerechnet werden soll. Hierzu nutzen wir Algorithmus 3.

Die Darstellung unserer Matrix A im TT-Format ($\boldsymbol{A}_{TT}$) geben wir dem Jacobi-Davidson-Algorithmus im Input mit. σ ist ein Skalar, sodass wir direkt $(\boldsymbol{A} - \sigma\boldsymbol{I})\boldsymbol{x}$ im TT-Format berechnen können. Für die Projektionen $(I - QQ^*)$ verwenden wir den für das Residuum bereits transformierten Tensor $\boldsymbol{Q}_{TT}$. Hiermit können wir das Skalarprodukt y von $\boldsymbol{Q}$ und $\boldsymbol{x}$ und anschließend $\boldsymbol{x} - \boldsymbol{Q}y$ im TT-Format berechnen, um die Projektion auszuführen. Zu beachten ist noch, dass wir nach Additionen und Matrix-Vektor-Multiplikationen eine Truncation ausführen müssen, um die Bonddimension niedrig zu halten. Wir geben dann

$$\text{CorrFunc}(\boldsymbol{A}_{TT}, \boldsymbol{x}, \boldsymbol{Q}_{TT}, \sigma, \epsilon) \tag{4.2.1}$$

als Funktion $\mathcal{A}(\boldsymbol{x}[, \epsilon])$ in $\boldsymbol{x}$ und ϵ mit festgesetzten $\boldsymbol{A}_{TT}$, $\boldsymbol{Q}_{TT}$ und σ aus der äußeren Iteration an den TT-GMRES-Algorithmus.

Mit diesen Anpassungen ergibt sich unser Tensor-Train-Jacobi-Davidson-Algorithmus 4. Dies ist die erste unserer Varianten, welche wir im Folgenden mit TT-Jada abkürzen werden. In rot sind die Änderungen gegenüber dem Jacobi-Davidson-QR-Algorithmus 1 markiert, welchen wir im Folgenden die Jada-Variante nennen.

Wir wählen für den TT-GMRES-Algorithmus in der TT-Jada-Variante beziehungsweise für den GMRES-Algorithmus in der Jada-Variante eine adaptive Toleranz $tol_{GMRES} = \|res\|_2 \cdot 1e - 4$ als Abbruchkriterium, wobei res das aktuelle Residuum in der äußeren Iteration des (TT-)Jacobi-Davidson-Algorithmus ist. Maximal sollen jedoch nur 20 GMRES-Schritte pro äußere Iteration durchgeführt werden. Der Grund hierfür ist, dass der Krylov-Unterraum im GMRES-Algorithmus mit jeder Iteration wächst. Hierdurch werden die Orthogonalisierungen zu teuer, weshalb man für gewöhnlich nach 20-50 Iterationen einen Restart benötigt. Anstelle eines Restarts führt man im Jacobi-Davidson-Algorithmus üblicherweise eine äußere Iteration durch, da dies schneller zum Ergebnis führt.

Nun kommen wir zu den weiteren Varianten des Tensor-Train-Jacobi-Davidson-Algorithmus, welche wir in unseren Experimenten verwenden werden.

Bei der TT-Jada-Variante führen wir in der äußeren Iteration noch immer eine Matrix-

Vektor-Multiplikation mit der dünnbesetzten Matrix durch (siehe Zeile 4 im Algorithmus 4). Hierdurch benötigt unser Algorithmus den Operator in beiden Darstellungen (A und $\boldsymbol{A}_{TT}$).

Um dies zu umgehen, betrachten wir die fullTT-Jada-Variante. Hierbei ersetzen wir die dünnbesetzte Matrix-Vektor-Multiplikation $v_A = Av$ aus Zeile 4 in Algorithmus 4 durch eine TT-Matrix-Vektor-Multiplikation und zwei Formattransformationen

$$v_A = \text{TT2Vec}(\boldsymbol{A}_{TT} \cdot \text{Vec2TT}(v)). \tag{4.2.2}$$

Der Rest des TT-Jada-Algorithmus bleibt gleich.

Die Variante fullTT verwendet somit bei allen Matrix-Vektor-Operationen den TT-Operator und benötigt die dünnbesetzte Matrix nicht mehr. Die anderen Berechnungen in der äußeren Iteration verwenden weiterhin gewöhnliche Vektoren.

Für das TT-Matrix-Vektor-Produkt wandeln wir den Vektor v mit hoher Genauigkeit (niedriger Toleranz $\epsilon = 1e-14$) ins TT-Format um. Wir führen hier nach der Multiplikation keine Truncation aus, sondern wandeln das Ergebnis direkt wieder in einen Vektor um, wodurch wir hier Truncation vermeiden können. Indem wir bei der fullTT-Variante nur die Matrix-Vektor-Multiplikation verändern, sollte der Genauigkeitsverlust minimal sein.

Bei den bisherigen Varianten haben wir immer direkt mit den Jacobi-Davidson-Iterationen gestartet. Hat man zu Beginn keine gute a-priori Schätzung des Eigenvektors des gesuchten Eigenwerts, so ergeben die ersten Iterationen eine teure Art, eine gute Startlösung zu finden. Hierbei wird der approximierte Eigenvektor des aktuell kleinsten Eigenwerts des Suchraums durch die Korrekturgleichung aufwendig verbessert. In der nächsten Iteration findet der Algorithmus dann einen kleineren Eigenwert im nun größeren Suchraum und die Approximation des Eigenvektors beginnt von vorne. Die quadratische Konvergenz des Jacobi-Davidson-Algorithmus setzt erst dann ein, wenn eine genügend genaue Approximation des kleinsten Eigenwerts gefunden wurde.

Um Kosten bei der Berechnung eines Anfangs-Suchraums des Jacobi-Davidson-Algorithmus zu sparen, ersetzt man üblicherweise die ersten Iterationen durch ein paar Iterationen des Arnoldi-Verfahrens. Somit startet der Jacobi-Davidson-Algorithmus mit der orthogonalen Basis W des Krylov-Unterraums K_k, wobei k die Anzahl der Arnoldi-Startschritte ist.

In der Implementation kann man die Arnoldi-Schritte direkt in die Jacobi-Davidson-Schleife einbauen. Hierzu wird in den ersten Iterationen die neue Suchrichtung durch $t = (I - QQ^*)v_A$ bestimmt, statt durch das Lösen der Korrekturgleichung.

Vergleichen wir das herkömmliche Jacobi-Davidson-Verfahren mit einer unserer Tensor-Train-Varianten, so spielt es keine entscheidende Rolle, ob wir beide Verfahren mit oder ohne Arnoldi-Startschritte betrachten. Beim Vergleich mit anderen Verfahren hingegen trägt dies zu einem besseren Abschneiden des Jacobi-Davidson-Verfahrens bei. Da wir unser Tensor-Train-Jacobi-Davidson-Verfahren auch mit einem ALS-Verfahren vergleichen wollen, haben wir die Option für eine feste Anzahl an Arnoldi-Startschritten eingebaut. Im Folgenden wird explizit darauf hingewiesen, falls in der Iteration mit k Arnoldi-Schritten gestartet wird.

5 Komplexitätsabschätzung

Die Grundoperationen im Tensor-Train-Format verhalten sich bezüglich der Laufzeit-Komplexität anders als die Grundoperationen mit dicht-gespeicherten Vektoren und voll- beziehungsweise dünnbesetzter Matrix. Betrachten wir beispielsweise die Multiplikation mit einem Skalar, so muss im TT-Format nur einer der TT-Kerne mit dem Skalar multipliziert werden und nicht wie bei einem Vektor jeder Eintrag. Die Kosten der Operationen hängen also nicht nur von der unterschiedlichen Elementanzahl eines Tensors in den beiden Darstellungen ab, sondern auch von der Darstellung selbst.

Deshalb wollen wir uns in diesem Kapitel mit den Komplexitätsabschätzungen einiger Operationen im TT-Format beschäftigen, um die Kosten in den beiden Formaten besser vergleichen zu können. Als Grundoperationen betrachten wir hier skalierte Additionen zweier Tensoren (axpy), Skalarprodukte (dot) sowie Matrix-Vektor-Multiplikationen beziehungsweise die Anwendung eines Tensor-Train-Operators, da diese Operationen im (TT-)Jacobi-Davidson-Algorithmus verwendet werden. Die Abschätzungen für diese und weitere Operationen können in [3] gefunden werden. Die folgenden Herleitungen sind konstruktiv und veranschaulichen, wie die TT-Operationen ausgeführt werden.

Satz 5.0.1. *Die skalierte Addition (axpy) von zwei Tensoren $\boldsymbol{T}$, $\boldsymbol{U} \in \mathbb{R}^N$ im TT-Format, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, hat eine Komplexität von $\mathcal{O}(dnr^3)$ Operationen, wobei $n = \max_i n_i$ und $r = \max(\text{bond}(\boldsymbol{T}), \text{bond}(\boldsymbol{U}))$.*

Beweis. Bei der skalierten Addition (axpy) von zwei Tensoren $\boldsymbol{T}$, $\boldsymbol{U} \in \mathbb{R}^N$ führen wir eine Skalarmultiplikation des Tensors $\boldsymbol{T}$ mit einem Skalar $a \in \mathbb{R}$ aus. Dies geschieht, wie oben erwähnt, indem wir einen der TT-Kerne $\boldsymbol{T}^{(i)}$ elementweise mit a multiplizieren. Dies benötigt $\mathcal{O}(nr^2)$ Operationen, welche wir auf $\mathcal{O}(nr)$ Operationen beschränken können, wenn wir hier den ersten TT-Kern $\boldsymbol{T}^{(1)}$ verwenden. Die Skalarmultiplikation verändert die Bonddimension des Tensors nicht: $\text{bond}(a \cdot \boldsymbol{T}) = \text{bond}(\boldsymbol{T})$.

Zusätzlich muss noch eine Addition ausgeführt werden. Im TT-Format werden hierzu für jede Mode die TT-Kerne $(a \cdot \boldsymbol{T})^{(i)}$ und $\boldsymbol{U}^{(i)}$ zu einem neuen TT-Kern zusammengesetzt (siehe [3, S. 2308, 2309] für mehr Details). Dies kostet an sich keine Floating-Point-Operationen, sondern nur das Kopieren von Daten, was hier vernachlässigbar ist. Allerdings erzeugen wir so einen Tensor mit erhöhten Bonddimensionen $\text{bond}((a \cdot \boldsymbol{T}) + \boldsymbol{U})_i = \text{bond}(a \cdot \boldsymbol{T})_i + \text{bond}(\boldsymbol{U})_i = \text{bond}(\boldsymbol{T})_i + \text{bond}(\boldsymbol{U})_i$, $i = 0, \ldots, d$, wodurch wir eine Truncation benötigen. Da $\max(\text{bond}((a \cdot \boldsymbol{T}) + \boldsymbol{U})) \leq 2r$ kann diese in $\mathcal{O}(dnr^3)$ berechnet werden (siehe Lemma 2.3.10).

Somit kommen wir hier insgesamt auf eine Komplexität von $\mathcal{O}(dnr^3)$ für die skalierte Addition zweier Tensoren im TT-Format. □

Hiermit haben wir eine Abschätzung für die Komplexität der skalierten Addition im TT-Format. Nun wollen wir uns mit dem Skalarprodukt (dot) von zwei Tensoren $\boldsymbol{T}$

und $\boldsymbol{U}$ beschäftigen. Dieses ist nach (2.1.3) definiert durch $\operatorname{dot}(\boldsymbol{T},\boldsymbol{U}) = \langle \boldsymbol{T},\boldsymbol{U}\rangle_N = \sum_{x_1,\dots,x_d} \boldsymbol{T}_{x_1,\dots,x_d} \cdot \boldsymbol{U}_{x_1,\dots,x_d}$.

Satz 5.0.2. *Das Skalarprodukt (dot) zweier Tensoren $\boldsymbol{T}$, $\boldsymbol{U} \in \mathbb{R}^N$ im TT-Format, $N = (n_1,\dots,n_d) \in \mathbb{N}^d$, hat eine Komplexität von $\mathcal{O}(dnr^3)$ Operationen, wobei $n = \max_i n_i$ und $r = \max(\operatorname{bond}(\boldsymbol{T}), \operatorname{bond}(\boldsymbol{U}))$.*

Beweis. Für $\boldsymbol{T}$ und $\boldsymbol{U}$ im TT-Format, das heißt $\boldsymbol{T} = \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \bigotimes_{i=1}^{d} \boldsymbol{T}^{(i)}_{k_{i-1},:,k_i}$ und $\boldsymbol{U} = \sum_{l_0=1}^{s_0} \cdots \sum_{l_d=1}^{s_d} \bigotimes_{i=1}^{d} \boldsymbol{U}^{(i)}_{l_{i-1},:,l_i}$, können wir aus dem Skalarprodukt durch geschicktes Umformen d verschachtelte Indexkontraktionen der Bondindizes k_i und l_i und der Moden n_i, $i = 1,\dots,d$ erzeugen:

$$
\begin{aligned}
&\operatorname{dot}(\boldsymbol{T},\boldsymbol{U}) \\
&= \sum_{x_1,\dots,x_d} \boldsymbol{T}_{x_1,\dots,x_d} \cdot \boldsymbol{U}_{x_1,\dots,x_d} \\
&= \sum_{x_1,\dots,x_d} \left(\sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{T}^{(1)}_{k_0,x_1,k_1} \cdots \boldsymbol{T}^{(d)}_{k_{d-1},x_d,k_d} \right) \cdot \left(\sum_{l_0=1}^{s_0} \cdots \sum_{l_d=1}^{s_d} \boldsymbol{U}^{(1)}_{l_0,x_1,l_1} \cdots \boldsymbol{U}^{(d)}_{l_{d-1},x_d,l_d} \right) \\
&= \sum_{x_1=1}^{n_1} \cdots \sum_{x_d=1}^{n_d} \sum_{k_0=1}^{r_0} \cdots \sum_{k_d=1}^{r_d} \sum_{l_0=1}^{s_0} \cdots \sum_{l_d=1}^{s_d} \left(\boldsymbol{T}^{(1)}_{k_0,x_1,k_1} \cdots \boldsymbol{T}^{(d)}_{k_{d-1},x_d,k_d} \boldsymbol{U}^{(1)}_{l_0,x_1,l_1} \cdots \boldsymbol{U}^{(d)}_{l_{d-1},x_d,l_d} \right) \\
&= \sum_{k_0,l_0} \left(\sum_{k_1,l_1,x_1} \boldsymbol{T}^{(1)}_{k_0,x_1,k_1} \boldsymbol{U}^{(1)}_{l_0,x_1,l_1} \left(\cdots \left(\sum_{k_d,l_d,x_d} \boldsymbol{T}^{(d)}_{k_{d-1},x_d,k_d} \boldsymbol{U}^{(d)}_{l_{d-1},x_d,l_d} \right) \cdots \right) \right). \qquad (5.0.1)
\end{aligned}
$$

Sei $\boldsymbol{V}^{(d)}_{k_{d-1},l_{d-1}} := \sum_{k_d,l_d,x_d} \boldsymbol{T}^{(d)}_{k_{d-1},x_d,k_d} \boldsymbol{U}^{(d)}_{l_{d-1},x_d,l_d}$. Da $r_d = s_d = 1$ müssen diese beiden Indizes nicht kontrahiert werden. Die Berechnung von $\boldsymbol{V}^{(d)}$ liegt somit in $\mathcal{O}(nr^2)$ mit $n = \max_i n_i$ und $r = \max_i(r_i, s_i) = \max(\operatorname{bond}(\boldsymbol{T}), \operatorname{bond}(\boldsymbol{U}))$. Dies gibt uns den Tensor für die nächste Indexkontraktion.

Nutzen wir die besondere Struktur in unserer Summe aus, so können wir die Indexkontraktion von k_{d-1}, l_{d-1} und n_{d-1} wie folgt aufteilen:

$$
\begin{aligned}
\boldsymbol{V}^{(d-1)}_{k_{d-2},l_{d-2}} &= \sum_{k_{d-1},l_{d-1},x_{d-1}} \boldsymbol{T}^{(d-1)}_{k_{d-2},x_{d-1},k_{d-1}} \boldsymbol{U}^{(d-1)}_{l_{d-2},x_{d-1},l_{d-1}} \boldsymbol{V}^{(d)}_{k_{d-1},l_{d-1}} \\
&= \sum_{k_{d-1},x_{d-1}} \boldsymbol{T}^{(d-1)}_{k_{d-2},x_{d-1},k_{d-1}} \sum_{l_{d-1}} \boldsymbol{U}^{(d-1)}_{l_{d-2},x_{d-1},l_{d-1}} \boldsymbol{V}^{(d)}_{k_{d-1},l_{d-1}}. \qquad (5.0.2)
\end{aligned}
$$

Definieren wir $\boldsymbol{W}^{(d-1)}_{l_{d-2},x_{d-1},k_{d-1}} = \sum_{l_{d-1}} \boldsymbol{U}^{(d-1)}_{l_{d-2},x_{d-1},l_{d-1}} \boldsymbol{V}^{(d)}_{k_{d-1},l_{d-1}}$, so liegt die Berechnung von $\boldsymbol{W}^{(d-1)}$ in $\mathcal{O}(nr^3)$. $\sum_{k_{d-1},x_{d-1}} \boldsymbol{T}^{(d-1)}_{k_{d-2},x_{d-1},k_{d-1}} \boldsymbol{W}^{(d-1)}_{l_{d-2},x_{d-1},k_{d-1}}$ können wir im Anschluss in $\mathcal{O}(nr^3)$ berechnen. Insgesamt erfolgt die Berechnung von $\boldsymbol{V}^{(d-1)}$ mittels $\boldsymbol{W}^{(d-1)}$ also in $\mathcal{O}(nr^3)$ Operationen.

Dies führen wir sequenziell für die d verschachtelten Indexkontraktionen aus. Da $r_0 = s_0 = 1$ fällt die äußerste Summe weg und wir kommen zur Abschätzung $\mathcal{O}(dnr^3)$ für das Skalarprodukt zweier Tensoren. □

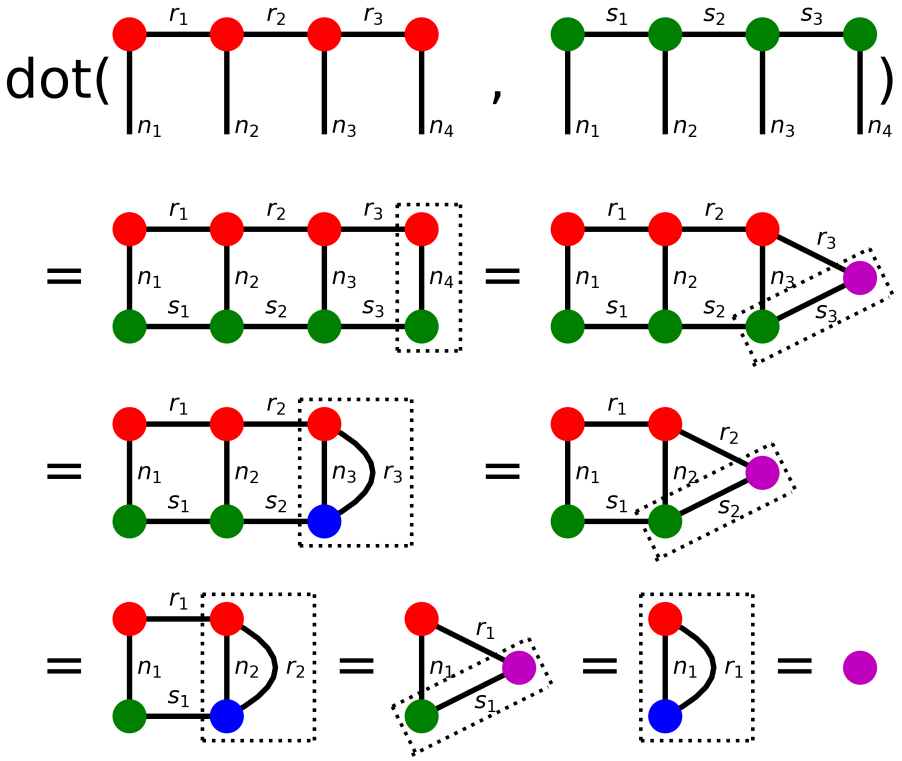


Abbildung 5.1: Skalarprodukt zweier TT-Tensoren $\boldsymbol{T}$ (rot) und $\boldsymbol{U}$ (grün). In den Zwischenschritten werden die „Kerne“ $\boldsymbol{V}^{(i)}$ (lila) und $\boldsymbol{W}^{(i)}$ (blau) aus den umrahmten Kerne berechnet.

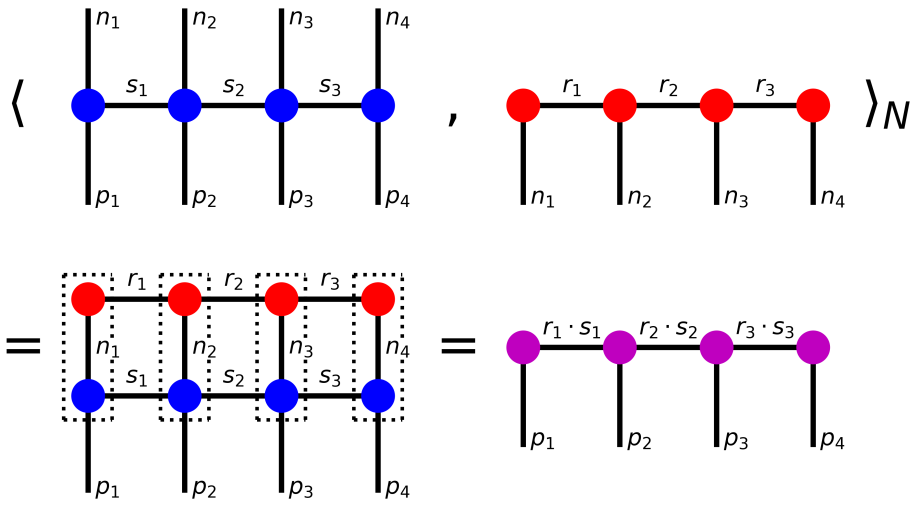


Abbildung 5.2: Matrix-Vektor-Produkt des TT-Operators $\boldsymbol{G}$ (blau) und des TT-Tensors $\boldsymbol{T}$ (rot). Die TT-Kerne des Ergebnistensors $(\boldsymbol{G} \cdot \boldsymbol{T})$ (lila) entsprechen dem Produkt der Kerne, welche von der jeweiligen gepunkteten Linie eingefasst werden.

Der Beweis des Satzes beschreibt konstruktiv, wie das Skalarprodukt im TT-Format berechnet werden kann. Abbildung 5.1 zeigt die graphische Darstellung dieses Prozesses. Es werden nacheinander die Indexkontraktionen der eingerahmten Knoten durchgeführt, um das Skalarprodukt zu berechnen. In den Zwischenschritten entstehen hierbei die blauen Knoten ($\boldsymbol{W}^{(i)}$) und die lila Knoten ($\boldsymbol{V}^{(i)}$).

Kommen wir nun zum Matrix-Vektor-Produkt. Analog zu diesem ist das Produkt $(\boldsymbol{G} \cdot \boldsymbol{T}) \in \mathbb{R}^P$ zwischen einem Tensor $\boldsymbol{T} \in \mathbb{R}^N$ und einem Tensoroperator $\boldsymbol{G} \in \mathbb{R}^{P \times N}$ nach (2.1.1) wie folgt definiert:

$$(\boldsymbol{G} \cdot \boldsymbol{T})_{x_1,\dots,x_d} = \sum_{y_1=1}^{n_1} \cdots \sum_{y_d=1}^{n_d} \boldsymbol{G}_{x_1,y_1,\dots,x_d,y_d} \cdot \boldsymbol{T}_{y_1,\dots,y_d}.$$

Satz 5.0.3. *Die Berechnung des Produkts eines Tensors $\boldsymbol{T} \in \mathbb{R}^N$ mit einem Tensoroperator $\boldsymbol{G} \in \mathbb{R}^{P \times N}$ im TT-Format, $N = (n_1, \dots, n_d) \in \mathbb{N}^d$, $P = (p_1, \dots, p_d) \in \mathbb{N}^d$, hat eine Komplexität von $\mathcal{O}(dnpr_{Op}^2 r^2 + dpr_{Op}^3 r^3)$ Operationen, wobei $n = \max_i n_i$, $p = \max_i p_i$, $r = \max(\text{bond}(\boldsymbol{T}))$ und $r_{Op} = \max(\text{bond}(\boldsymbol{G}))$.*

Beweis. Im TT-Format ergeben sich die Tensor-Train-Kerne des Produkts $(\boldsymbol{G} \cdot \boldsymbol{T})^{(i)} \in \mathbb{R}^{r_{i-1}s_{i-1} \times p_i \times r_i s_i}$ aus den lokalen Produkten der Tensor-Train-Kerne $\boldsymbol{T}^{(i)} \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$ und $\boldsymbol{G}^{(i)} \in \mathbb{R}^{s_{i-1} \times p_i \times n_i \times s_i}$ (siehe [3, S. 2312, 2313]). Somit reduziert sich das globale Produkt auf d lokale Produkte:

$$(\boldsymbol{G} \cdot \boldsymbol{T})^{(i)}_{:,x_i,:} = \sum_{y_i=1}^{n_i} \boldsymbol{G}^{(i)}_{:,x_i,y_i,:} \otimes \boldsymbol{T}^{(i)}_{:,y_i,:}, \; i = 1, \dots, d.$$

Zur Veranschaulichung kann Abbildung 5.2 betrachtet werden. Die jeweiligen lila Knoten ergeben sich aus dem Produkt der jeweils eingerahmten TT-Kerne.

Die Berechnung eines solchen TT-Kerns des Ergebnistensors benötigt $\mathcal{O}(npr_{Op}^2 r^2)$ Operationen, wobei $r = \max_i r_i = \max(\text{bond}(\boldsymbol{T}))$, $r_{Op} = \max_i s_i = \max(\text{bond}(\boldsymbol{G}))$, $n = \max_i n_i$ und $p = \max_i p_i$.

Da wir auf diese Weise einen Tensor mit Bonddimensionen $\text{bond}(\boldsymbol{G} \cdot \boldsymbol{T})_i = \text{bond}(\boldsymbol{G})_i \cdot \text{bond}(\boldsymbol{T})_i$ erhalten, dessen maximale Bonddimension durch $r_{Op} \cdot r$ abgeschätzt werden kann, benötigen wir zusätzlich eine Truncation mit Kosten $\mathcal{O}(dp(r_{Op} r)^3)$ und kommen so insgesamt zur Abschätzung $\mathcal{O}(dnpr_{Op}^2 r^2 + dpr_{Op}^3 r^3)$ für das Matrix-Vektor-Produkt im TT-Format. □

Da wir in dieser Arbeit nur Operatoren mit $P = N$ betrachten, ergibt sich für uns die Komplexität von $\mathcal{O}(dn^2 r_{Op}^2 r^2 + dnr_{Op}^3 r^3)$ Operationen für das Matrix-Vektor-Produkt im TT-Format.

Kommen wir nun zu den Komplexitätsabschätzungen der betrachten Operationen bei der Verwendung einer Matrix und Vektoren. Hierzu bestimmen wir zunächst, welche Dimension die Vektorisierung eines Tensors beziehungsweise die Matrizisierung eines Tensoroperators hat. Stellen wir einen Tensor als Vektor dar, so hat dieser eine Länge von $\mathcal{O}(n^d)$ Elementen. Aus dem Tensoroperator wird eine vollbesetzte (dense) Matrix der Dimension $\mathcal{O}(n^d) \times \mathcal{O}(n^d)$ beziehungsweise eine dünnbesetzte (sparse) Matrix mit

$\mathcal{O}(m \cdot n^d)$ Elementen, wobei m die maximale Anzahl der Nicht-Null-Einträge pro Zeile ist.

Für die Komplexität der betrachteten Operationen für Matrizen und Vektoren der genannten Dimension ergeben sich die Abschätzungen in Tabelle 5.1, welche auch die Abschätzungen im TT-Format noch einmal wiedergibt.

	axpy	dot	Matrix-Vektor-Multiplikation
dünnbesetzte Matrix + Vektor vollbesetzte Matrix + Vektor	$\mathcal{O}(n^d)$	$\mathcal{O}(n^d)$	$\mathcal{O}(mn^d)$ $\mathcal{O}(n^{2d})$
Tensor-Train-Format	$\mathcal{O}(dnr^3)$	$\mathcal{O}(dnr^3)$	$\mathcal{O}(dn^2r_{Op}^2r^2 + dnr_{Op}^3r^3)$

Tabelle 5.1: Komplexität verschiedener Rechenoperationen

Im Tensor-Train-Format entfällt die exponentielle Abhängigkeit von d, welche wir bei der Verwendung von Matrizen und Vektoren haben. Hier spielen allerdings die maximalen Bonddimensionen r und r_{Op} eine Rolle. Die maximale Bonddimension r_{Op} des TT-Operators ist problemabhängig, für unsere Probleme aber klein und unabhängig von der Dimension d. Er wird durch die betrachteten Operationen auch nicht verändert, wodurch r_{Op} bei uns eine problemabhängige Konstante ist.

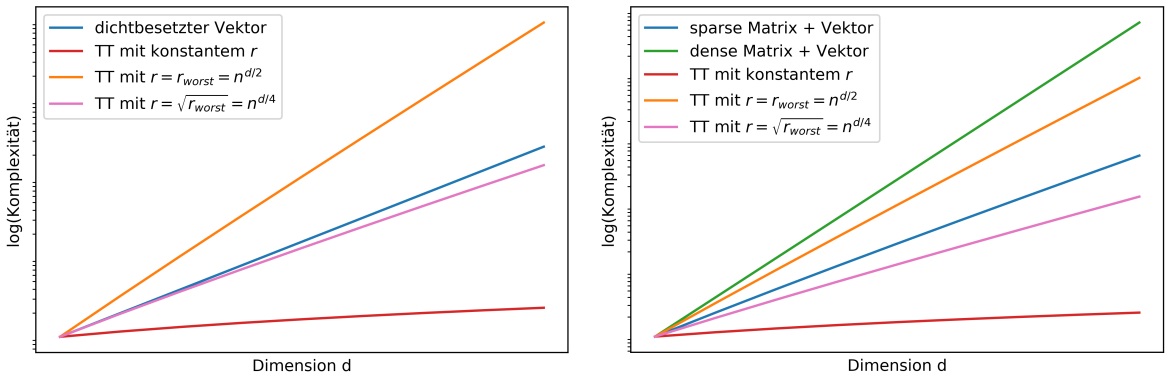
Anders sieht es bei der maximalen Bonddimension r eines TT-Tensors aus. Diese kann sich durch die Berechnungen vergrößern und Werte bis $r_{worst} = n^{\lfloor d/2 \rfloor}$ annehmen (siehe Folgerung 2.3.13). Hierdurch können wir wieder eine exponentielle Abhängigkeit von d in unserer Abschätzung erhalten.

Im Folgenden wollen wir betrachten, wie sich die Abschätzungen in Abhängigkeit der Dimension d verhalten, wenn wir den optimalen Fall einer konstanten (von d unabhängigen) maximalen Bonddimension r betrachten und wie sie für den Worst-Case aussieht, in welchem die maximale Bonddimension mit der Problemgröße wächst ($r = r_{worst}$).

Die zugehörigen Kurvenverläufe sind in Abbildung 5.3 mit logarithmisch-skalierter y-Achse visualisiert. Hierfür haben wir die Parameter $n = 2$, $m = 2d + 1$ und $r_{Op} = 8$ des Spinketten-Problems aus Kapitel 6.1 verwendet und die Kurven so verschoben, dass sie alle im selben Punkt starten, um die Verläufe besser vergleichen zu können.

Zu beachten ist, dass auf Grund der unbekannten Konstanten in der $\mathcal{O}$-Notation der tatsächliche Aufwand um große Konstanten von unseren Kurven abweichen kann. Dennoch können wir am Verlauf der Kurven abschätzen, für welche Methode wir bei großer Dimension d geringere Laufzeit erwarten. Hat eine Kurve gegenüber einer zweiten eine geringere Steigung, so kann es durch eine große Konstante sein, dass der tatsächliche Aufwand der ersten Kurve höher ist, als der der zweiten Kurve. Durch die geringere Steigung können wir allerdings darauf schließen, dass für d groß genug der tatsächliche Aufwand der ersten Kurve unter den der zweiten Kurve fällt und man ab dem Punkt bessere Laufzeiten für die erste Methode erhält.

Betrachten wir eine konstante maximale Bonddimension r, so bekommen wir die Dimension d des Problems bei den TT-Operationen nur als linearen Faktor in der Abschätzung. Die Matrix-Vektor-Operationen hingegen haben eine exponentielle Abhängigkeit von d. Damit ist der Verlauf der Kosten im TT-Format wesentlich flacher, als bei der Ver-



(a) axpy / dot

(b) Matrix-Vektor-Multiplikation

Abbildung 5.3: Verlauf der Komplexitätsabschätzungen verschiedener Operationen im TT-Format (für unterschiedliche Wahlen der maximalen Bonddimension r) und bei der Verwendung einer dünn- beziehungsweise vollbesetzten Matrix und Vektoren. Die möglichen Konstanten in der Abschätzung werden hierbei vernachlässigt.

wendung von Matrizen und Vektoren. Somit erzielen die TT-Operationen mit konstanter Bonddimension gegenüber den Matrix-Vektor-Operationen für d groß genug bessere Laufzeiten, selbst wenn in der Abschätzung noch eine große Konstante hinzukommt.

Im Fall $r = r_{worst} = n^{\lfloor d/2 \rfloor}$ erhalten wir auch bei den TT-Operationen eine exponentielle Abhängigkeit von d, welche wir durch das TT-Format eigentlich vermeiden wollen. Durch die r^3-Abhängigkeit und da wir bei dünnbesetzten Matrizen von einem kleinen m ausgehen, erhalten wir bei den TT-Varianten einen schlechteren, das heißt steileren, Kurvenverlauf als für dot/axpy mit Vektoren und für das dünnbesetzte Matrix-Vektor-Produkt. Die TT-Variante des Matrix-Vektor-Produkts erhält allerdings eine bessere Abschätzung als das vollbesetzte Produkt, da der Exponent im TT-Fall etwas niedriger ist als im vollbesetzten Fall. Demnach erwarten wir hier im TT-Format schlechtere Laufzeiten als bei der Verwendung einer Matrix und Vektoren, da man in der Regel mit dünnbesetzten Matrizen rechnet.

Zusätzlich ist in der Abbildung noch der Verlauf für den Aufwand im TT-Format mit $r = \sqrt{r_{worst}} = n^{d/4}$ dargestellt, da wir für diesen Fall wieder etwas flachere Kurvenverläufe im TT-Format erhalten als bei der Verwendung einer dünnbesetzten Matrix und Vektoren. Da hier die Steigung sehr ähnlich ist, erwarten wir erst für sehr große Werte von d, dass wir eine Zeitersparnis bei der Berechnung der Operationen im TT-Format erhalten.

Zusammengefasst stellen wir fest: Für die Worst-Case-Bonddimension erhalten wir höhere Komplexität als bei der Verwendung einer dünnbesetzten Matrix und vollbesetzten Vektoren. Ist die maximale Bonddimension wesentlich kleiner als die Worst-Case-Bonddimension, aber noch exponentiell abhängig von der Dimension d (Beispiel oben: $r = n^{d/4}$), so bleibt die Komplexität exponentiell abhängig von d. Allerdings kann diese, abhängig von den Konstanten, kleiner sein als im Fall mit einer dünnbesetzten Matrix und Vektoren. Ist unsere maximale Bonddimension konstant, das heißt unabhängig von

d, so bekommen wir nur eine lineare Abhängigkeit von d für die Operationen im TT-Format.

Es ist also entscheidend, eine im Verhältnis zur Problemgröße n^d sehr kleine und nach Möglichkeit konstante maximale Bonddimension zu haben, um durch das TT-Format Rechenaufwand bei großer Dimension d einzusparen.

Zusätzlich zur Komplexität der Rechenoperationen kann man sich auch noch mit der Speicherkomplexität beschäftigen.

Nach Lemma 2.3.4 hat ein Tensor $\boldsymbol{T} \in \mathbb{R}^N$ im TT-Format einen Speicherverbrauch von $\mathcal{O}(dnr^2)$ mit $n = \max_i n_i$ und $r = \max(bond(\boldsymbol{T}))$ und der Speicherbedarf eines TT-Operators $\boldsymbol{G} \in \mathbb{R}^{P \times N}$ kann durch $\mathcal{O}(dnpr_{Op}^2)$ abgeschätzt werden mit $p = \max_i p_i$ und r_{Op} der maximalen Bonddimension des TT-Operators. Da wir in dieser Arbeit nur Tensoroperatoren mit $P = N$ betrachten, erhalten wir für den Speicherverbrauch eines TT-Operators eine Abschätzung von $\mathcal{O}(dn^2r_{Op}^2)$.

Wie oben schon erwähnt, liegt der Speicherverbrauch bei $\mathcal{O}(n^d)$ für einen Vektor, $\mathcal{O}(n^{2d})$ für die vollbesetzte Matrix und $\mathcal{O}(mn^d)$ für die dünnbesetzte Matrix, wobei m die maximale Anzahl der Nicht-Null-Einträge pro Zeile ist.

Tabelle 5.2 fasst die Abschätzungen noch einmal zusammen.

	Tensor	Operator
dünnbesetzte Matrix + Vektor	$\mathcal{O}(n^d)$	$\mathcal{O}(mn^d)$
vollbesetzte Matrix + Vektor		$\mathcal{O}(n^{2d})$
Tensor-Train-Format	$\mathcal{O}(dnr^2)$	$\mathcal{O}(dn^2r_{Op}^2)$

Tabelle 5.2: Speicherkomplexität

Wie schon erwähnt, ist die maximale Bonddimension r_{Op} der TT-Operatoren in unseren Problemen klein und unabhängig von der Dimension d. Hiermit ist der Speicherverbrauch des TT-Operators tatsächlich nur linear von d abhängig und nicht exponentiell, wie dies für dünnbesetzte und vollbesetzte Matrizen der Fall ist. Wir sparen also bei großen Dimensionen d wesentlich an Speicherplatz, wenn wir den Operator im Tensor-Train-Format abspeichern.

Bei den TT-Tensoren hängt die Komplexität wieder stark von der maximalen Bonddimension ab. Betrachten wir wie beim Operator konstante maximale Bonddimension, so erhalten wir auch hier nur eine lineare Abhängigkeit und es würde sich lohnen, die Tensoren für große Dimension d im TT-Format zu speichern.

Betrachten wir wieder den Worst-Case mit $r = n^{\lfloor d/2 \rfloor}$, so bleibt die exponentielle Abhängigkeit von d bestehen. Durch die zusätzliche lineare Abhängigkeit von d in unserer obigen Abschätzung würden wir hier schlechtere Komplexität für den Speicherbedarf erhalten als mit einem dicht-gespeicherten Vektor. Zu beachten ist jedoch, dass r_{worst} nur von der mittlere Bonddimension angenommen werden kann und die anderen Bonddimensionen mit jedem Schritt zum Rand der Tensor-Train-Darstellung um den Faktor n abfallen (siehe Satz 2.3.12). Hierdurch ist der Speicherverbrauch im Worst-Case niedriger als die Abschätzung $\mathcal{O}(dnr^2)$ mit $r = n^{\lfloor d/2 \rfloor}$.

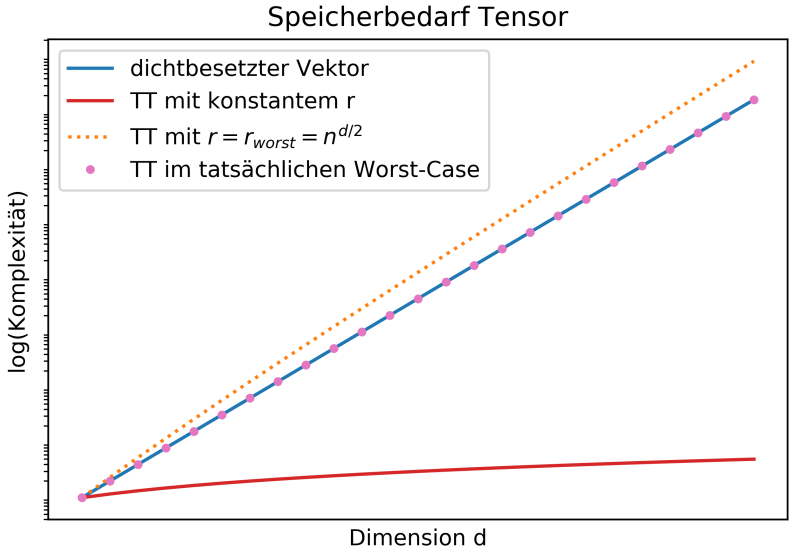


Abbildung 5.4: Verlauf der Komplexitätsabschätzungen des Speicherbedarfs eines Tensors im TT-Format (für verschiedene Fälle) und als dichtbesetzter Vektor. Die möglichen Konstanten in der Abschätzung werden hierbei vernachlässigt.

Satz 5.0.4. *Der Speicherverbrauch eines Tensors* $\boldsymbol{T} \in \mathbb{R}^N$, $N = (n_1, \ldots, n_d) \in \mathbb{N}^d$, *im TT-Format liegt im Worst-Case bei* $\mathcal{O}(n^d)$ *Elementen, wobei* $n = \max_i n_i$.

Beweis. Nach Satz 2.3.12 erfüllen die Bonddimensionen r_k der TT-Darstellung von $\boldsymbol{T}$ die Bedingung $r_k \leq n^{d/2-|d/2-k|}$. Hierdurch nehmen nicht nur die Bonddimensionen zum Rand der Zerlegung hin ab, sondern auch die Anzahl an Elementen der TT-Kerne. Die beiden mittleren Kerne haben jeweils höchstens $n^{d/2} \cdot n \cdot n^{d/2-1} = n^d$ Elemente, die zwei nächst kleineren Kerne liegen bei n^{d-2} Elementen und mit jedem weiteren Schritt zum Rand der TT-Darstellung verringert sich der Exponent um den Wert 2.

Für die gesamte TT-Darstellung des Tensors kommt man so auf $2\sum_{k=0}^{d/2} n^{d-2k}$ Elemente. Dies können wir mit Hilfe der geometrischen Reihe wie folgt abschätzen:

$$2\sum_{k=0}^{d/2} n^{d-2k} = 2n^d \sum_{k=0}^{d/2} (\frac{1}{n^2})^k \leq 2n^d \frac{n^2}{n^2-1}. \tag{5.0.3}$$

Da $n \geq 2$ gilt $\frac{n^2}{n^2-1} \leq 4/3$. Hiermit folgt $2\sum_{k=0}^{d/2} n^{d-2k} \leq 8/3n^d < 3n^d$. Somit erhalten wir für den Worst-Case tatsächlich eine Speicherkomplexität von $\mathcal{O}(n^d)$ mit einer etwas größeren Konstante als bei der Komplexität des Vektors. □

Abbildung 5.4 visualisiert analog zu Abbildung 5.3 die Komplexitätsverläufe des Speicherbedarfs des Tensors. Hier sehen wir noch einmal, dass die tatsächliche Worst-Case-Abschätzung des Speicherverbrauchs bessere Komplexität hat, als die einfache Annahme

mit $r = n^{d/2}$ für alle Bonddimensionen. Somit erhalten wir in der Abbildung die gleichen Kurven für den Vektor und die Worst-Case-Abschätzung, da hier die Konstanten vernachlässigt werden.

Zusammengefasst können wir also für unsere Operatoren, welche low-rank-approximierbar sind, bei großen Dimensionen d mit dem TT-Format Speicherplatz einsparen. Bei den Tensoren haben wir im TT-Format im schlechtesten Fall einen um eine kleine Konstante größeren Speicherbedarf. Hierdurch haben wir kaum Nachteile beim Speicherbedarf, wenn wir das TT-Format verwenden und können bei kleineren Bonddimensionen einiges an Speicherplatz sparen.

6 Numerische Experimente

In diesem Kapitel kommen wir zu den numerischen Experimenten. Durch diese wollen wir herausfinden, ob wir durch die Verwendung des Tensor-Train-Formats im Jacobi-Davidson-Verfahren den Rechenaufwand reduzieren können.

Hierbei vergleichen wir verschiedene Varianten des Jacobi-Davidson-Algorithmus aus Kapitel 4, sowie den eigb-Algorithmus aus ttpy. Beide Verfahren können mehrere Eigenwerte berechnen. Wir beschränken uns der Einfachheit halber auf die Berechnung nur eines Eigenpaars. Wir betrachten im Folgenden sowohl symmetrische als auch unsymmetrische Probleme.

6.1 Spinketten-Problem

Das Spinketten-Problem, welches wir in diesem Abschnitt betrachten, stammt vom Heisenberg-Modell. Dieses Modell wird in der Physik zur Beschreibung von Quantenmagnetismus und anderen quantenmechanischen Phänomenen genutzt ([21, S. 51]).

Betrachtet wird eine (periodische) Kette von L Elektronen, welche jeweils die Zustände Spin-Up $\uparrow$ oder Spin-Down $\downarrow$ haben können. Insgesamt kommen wir so auf 2^L Zustände Ψ für die gesamte Spinkette.

Durch Wechselwirkungen der Spins untereinander und Wechselwirkungen der Spins mit einem externen Magnetfeld kommt es zu Zustandsänderungen der Spinkette. Wir nehmen die Wechselwirkungen als kurzreichweitig und zeitlich getrennt an, sodass sich nur nächste Nachbarn gegenseitig beeinflussen und stets nur eine Wechselwirkung zur selben Zeit geschieht. Somit können die Zustandsänderungen durch einen großen dünnbesetzten Operator $\mathcal{H}$, Hamilton-Operator genannt, beschrieben werden.

Unser Ziel ist es, die Zustände mit niedrigstem Energieniveau $E \in \mathbb{R}$ zu ermitteln, was uns zum Eigenwertproblem $\mathcal{H}\Psi = E\Psi$ führt.

Wir gehen in dieser Arbeit nur sehr grob auf die physikalischen Hintergründe dieses Problems ein. Unsere Quellen hierfür sind [21, S. 16, 17, 51 und 52], [22, Kapitel 1.3, 1.4 und 2.1.4], [23, S. 42-53 und S. 246] und [1, Kapitel 6.1, 6.2 und 7.], welche einen genaueren Einblick in die physikalische Betrachtung dieses Problems geben.

Bevor wir mit den Experimenten starten, wollen wir zunächst darauf eingehen, wie wir von dem physikalischen Modell auf unser numerisches Modell für die Berechnung kommen. Hierfür betrachten wir die Repräsentation der Spinkettenzustände und des Hamilton-Operators im TT-Format und als Vektor und dünnbesetzte Matrix.

Darstellung in den verschiedenen Formaten

Zunächst gehen wir auf die Repräsentation der Spinkettenzustände ein, welche wir an folgendem Beispiel erläutern wollen:

$$\uparrow - \downarrow - \downarrow - \uparrow - \uparrow - \downarrow . \tag{6.1.1}$$

Für die Darstellung im TT-Format definieren wir den TT-Kern $\boldsymbol{S}_i$ für den i-ten Spinzustand als $\boldsymbol{S}_i = \begin{pmatrix}0\\1\end{pmatrix}$, wenn für das i-te Elektron $\downarrow$ gilt und $\boldsymbol{S}_i = \begin{pmatrix}1\\0\end{pmatrix}$, wenn für das i-te Elektron $\uparrow$ gilt. Der Zustandstensor $\boldsymbol{\Psi}$ der gesamten Spinkette wird dann durch das Tensorprodukt der TT-Kerne $\boldsymbol{S}_i$ der einzelnen Spinzustände im TT-Format dargestellt, das heißt $\boldsymbol{\Psi} = \bigotimes_{i=1}^{L} \boldsymbol{S}_i$. Für unser Beispiel (6.1.1) erhalten wir somit die folgende TT-Darstellung:

$$\boldsymbol{\Psi} = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix}. \tag{6.1.2}$$

Ersetzt man im Spinkettenzustand $\uparrow$ durch 1 und $\downarrow$ durch 0, so erhält man eine Binärzahl mit Dezimaldarstellung *dec*. Diese verwenden wir für die Darstellung des Zustandstensors $\boldsymbol{\Psi}$ als Vektor (Ψ). Wir nummerieren die Einträge von $\boldsymbol{\Psi}$ so durch, dass in Zeile *dec* des Vektors der Eintrag für den zu *dec* gehörigen Spinkettenzustand steht. In unserem Beispiel (6.1.1) erhalten wir aus dem Spinkettenzustand die Binärzahl 100110. Mit $dec = \text{dezimal}(100110) = 38$ erhalten wir folgenden Vektor:

$$\Psi \in \mathbb{R}^{2^L},\ \Psi[i] = \begin{cases}1, & i = dec\\ 0, & \text{sonst}\end{cases}. \tag{6.1.3}$$

Nun kommen wir zur Darstellung des Hamilton-Operators.

Der Hamilton-Operator $\mathcal{H}$ setzt sich aus verschiedenen Operationen zusammen. Diese Operationen kommen von den unterschiedlichen Wechselwirkungen, welche die Übergänge zwischen den Spinkettenzuständen herbeiführen.

Eine dieser Wechselwirkung ist die magnetische Wechselwirkung der Spins untereinander, welche sich, wie bereits erwähnt, nur zwischen nächsten Nachbarn abspielt. Benachbarte Elektronen mit entgegengesetzten Spins können hierdurch einen sogenannten Spinflip durchführen. Hierbei gehen beide Spins auf eins der Teilchen über, vertauschen sich dort und trennen sich wieder, wodurch die Spinzustände der beiden Elektronen vertauscht werden. Ein Spinflip macht aus einer Spin-Up Spin-Down oder Spin-Down Spin-Up Folge in der Spinkette also eine Spin-Down Spin-Up beziehungsweise Spin-Up Spin-Down Folge ($\uparrow\downarrow \to \downarrow\uparrow$, $\downarrow\uparrow \to \uparrow\downarrow$), wobei es hier zwei mögliche Austauschpfade gibt. Dies kann durch den Operator $2(\boldsymbol{S}^+ \otimes \boldsymbol{S}^- + \boldsymbol{S}^- \otimes \boldsymbol{S}^+)$ angewandt auf das jeweilige Spinpaar im TT-Format erzeugt werden. Hierbei ist $\boldsymbol{S}^+ = \left(\begin{smallmatrix}0&1\\0&0\end{smallmatrix}\right)$ und $\boldsymbol{S}^- = \left(\begin{smallmatrix}0&0\\1&0\end{smallmatrix}\right)$.

Zusätzlich ist es energetisch günstig, wenn benachbarte Spins gleichgerichtet sind. Dies führt dazu, dass Spinkettenzustände mit vielen parallelen Nachbarspins in ihrem Zustand verharren, während Spinkettenzustände mit vielen antiparallelen Nachbarspins zur Veränderung tendieren. Die Parallelität eines Nachbarpaars wird durch den TT-

Operator $\boldsymbol{S}^z \otimes \boldsymbol{S}^z$ angewandt auf das jeweilige Spinpaar im TT-Format ermittelt, wobei $\boldsymbol{S}^z = \left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$.

Betrachten wir nur diese Spin-Spin-Wechselwirkungen, so erhalten wir für zwei Spins in einer nicht-periodischen Spinkette den Hamilton-Operator $\boldsymbol{\mathcal{H}}_{Spin}(L=2) = \boldsymbol{S}^z \otimes \boldsymbol{S}^z + 2(\boldsymbol{S}^+ \otimes \boldsymbol{S}^- + \boldsymbol{S}^- \otimes \boldsymbol{S}^+)$.

Betrachten wir mehr Spins, so können wir den Hamilton-Operator berechnen, indem wir den $L = 2$ Fall über alle Nachbarpaare aufsummieren, wobei auf die gerade nicht betrachteten Spins die Identität angewandt wird. Somit ergibt sich der folgende TT-Operator:

$$\boldsymbol{\mathcal{H}}_{Spin} = \sum_{i=1}^{L-1} (\bigotimes_{j=1}^{i-1} \boldsymbol{I}) \otimes \boldsymbol{\mathcal{H}}_{Spin}(L=2) \otimes (\bigotimes_{j=i+2}^{L} \boldsymbol{I}). \tag{6.1.4}$$

Ist die Spinkette periodisch, so kommt noch der Summand L hinzu mit $\boldsymbol{S}_{L+1} = \boldsymbol{S}_1$, das heißt, die zweite Hälfte von $\boldsymbol{\mathcal{H}}_{Spin}(L=2)$ wird auf den ersten Spin angewandt.

Zusätzlich zu den Spin-Spin-Wechselwirkungen können die Spins noch mit einem äußeren Magnetfeld wechselwirken. Ein Magnetfeld in x-Richtung sorgt für spontane Spinumkehrung einzelner Spins, wodurch der Spinkettenzustand gewechselt wird. Ein Magnetfeld in z-Richtung sorgt, ähnlich wie die gleichgerichteten Nachbarspins, dafür, dass ein Spinkettenzustand mit vielen zum Magnetfeld parallelen Spins gleich bleibt, während sich ein Zustand mit vielen zum Magnetfeld antiparallelen Spins verändert.

Wir betrachten ein Magnetfeld sowohl in x- als auch in z-Richtung, wobei für die Stärke des Magnetfelds einfach 1 gewählt wurde. Für einen einzelnen Spin ergibt sich dann der TT-Operator $\boldsymbol{\mathcal{H}}_{Magn}(L=1) = \boldsymbol{S}^z + \boldsymbol{S}^x$ mit $\boldsymbol{S}^x = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ und $\boldsymbol{S}^z$ wie oben.

Betrachten wir mehrere Spins, so summieren wir analog zu $\boldsymbol{\mathcal{H}}_{Spin}$ über die einzelnen Spins, wobei wir auf die gerade nicht betrachteten Spins die Identität anwenden. Es ergibt sich folgender TT-Operator:

$$\boldsymbol{\mathcal{H}}_{Magn} = \sum_{i=1}^{L} (\bigotimes_{j=1}^{i-1} \boldsymbol{I}) \otimes \boldsymbol{\mathcal{H}}_{Magn}(L=1) (\bigotimes_{j=i+1}^{L} \boldsymbol{I}). \tag{6.1.5}$$

Insgesamt ergibt sich die folgende TT-Darstellung für den Hamilton-Operator:

$$\boldsymbol{\mathcal{H}} = \boldsymbol{\mathcal{H}}_{Spin} + \boldsymbol{\mathcal{H}}_{Magn}. \tag{6.1.6}$$

Diese TT-Darstellung ist noch nicht optimal, da wir sehr oft den Identitätsoperator speichern. Wenden wir mit Hilfe des Python-Pakets ttpy eine Truncation auf den Hamilton-Operator an, so erhalten wir eine TT-Darstellung mit maximaler Bonddimension 8 im periodischen und maximaler Bonddimension 5 im nicht-periodischen Fall.

Für die Darstellung des Hamilton-Operators als dünnbesetzte (sparse) Matrix würde es zu viel Speicherplatz beanspruchen, den vollen Operator zu erstellen, um dann die Einträge durchzunummerieren und in einer Matrix anzuordnen. Deshalb arbeiten wir hier mit den Binärzahlen der Spinkettenzustände.

Betrachten wir auch hier zunächst die Spin-Spin-Wechselwirkungen. Für den Spinflip gilt, dass dieser stattfindet, wenn sich die Binärzahlen des Zeilenindex und des Spaltenindex der Matrix durch genau eine Vertauschung eines 01 beziehungsweise 10 Nachbarpaars unterscheiden. In alle diese Einträge setzen wir eine 2 für die zwei Austauschpfade $|\uparrow,\downarrow\rangle \to |\uparrow\downarrow,0\rangle \to |\downarrow,\uparrow\rangle$ und $|\uparrow,\downarrow\rangle \to |0,\uparrow\downarrow\rangle \to |\downarrow,\uparrow\rangle$.

Für die Parallelität der Spinzustände zählen wir in der Binärzahl die Anzahl an 11 und 00 Paaren und die Anzahl an 10 oder 01 Paaren. Ziehen wir die Anzahl der antiparallelen Paare von der Anzahl der parallelen Paare ab, so erhalten wir unseren Diagonaleintrag.

Zu beachten ist, dass bei einer periodischen Spinkette die erste und letzte Stelle der Binärzahl auch ein Nachbarpaar bilden.

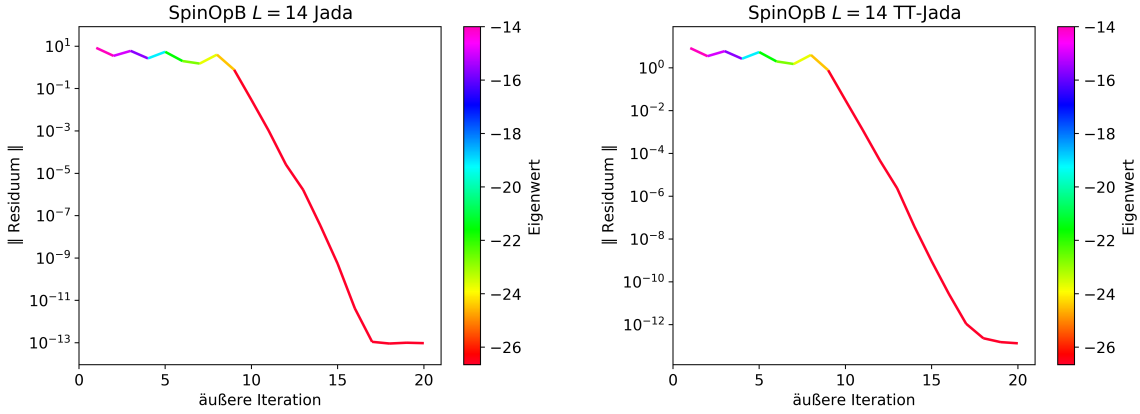
Dann fügen wir noch die Wechselwirkung mit dem Magnetfeld hinzu. Das x-Magnetfeld lässt einen einzelnen Spin umklappen, das heißt, wir erhalten hier überall eine 1 in der Matrix, wo sich die Binärzahlen des Spaltenindex und des Zeilenindex nur in genau einer Stelle unterscheiden. Das z-Magnetfeld zählt die Differenz der Anzahl zum Magnetfeld parallelen Spins zu der Anzahl zum Magnetfeld antiparalleler Spins und addiert diesen Wert zur Diagonalen hinzu.

Numerische Ergebnisse

Bei den folgenden Experimenten werden periodische Spinketten verschiedener Länge in einem äußeren Magnetfeld betrachtet. Wir vergleichen hier den Jacobi-Davidson-Algorithmus in der Jada-Variante mit der TT-Jada-Variante.

Abbildung 6.1 zeigt den Residuen-Verlauf ($\|res\|_2 = \|AQ - \sigma Q\|_2$) in den äußeren Iterationen der Jada- und der TT-Jada-Variante für den periodischen Spinketten-Operator mit Magnetfeld (SpinOpB) für $L = 14$ Spins.

Die verschiedenen Farben kennzeichnen die Eigenwert-Approximation, welche in der jeweiligen Iteration berechnet wurde. Diese spielt eine Rolle, da zu Beginn des Jacobi-Davidson-Verfahrens der Suchraum noch sehr klein ist und hier somit keine gute Ap-



(a) Jada (b) TT-Jada

Abbildung 6.1: Residuenverlauf von SpinOpB L=14 periodisch

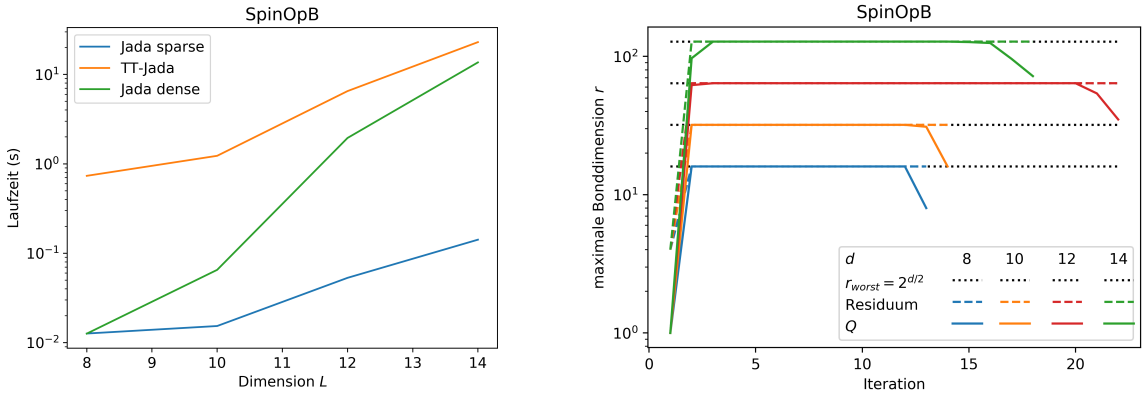
proximation des zu berechnenden Eigenvektors zum kleinsten Eigenwert enthalten ist. Aus diesem Grund bleibt das Residuum auf einem hohen Wert, da wir zwar die aktuelle Approximation des Eigenpaars durch das GMRES-Verfahren verbessern, doch in der nächsten äußeren Iteration wird ein neuer weiter außen liegender Eigenwert gefunden, mit dem der Algorithmus dann weiterarbeitet und welcher noch nicht verbessert wurde. Erst, wenn eine gute Approximation des kleinsten Eigenwerts gefunden wurde, haben die Iterationen Auswirkungen auf das Residuum, welches dann kontinuierlich abfällt bis die gewünschte Genauigkeit erreicht ist.

Wir können in Abbildung 6.1 erkennen, dass die Verläufe der Kurven sehr ähnlich sind und beide Verfahren eine Genauigkeit von etwa $\|res\|_2 = 1e-13$ erreichen. Die selben Beobachtungen konnten wir auch für die anderen getesteten Dimensionen L machen. Demnach können wir das GMRES-Verfahren durch das TT-GMRES-Verfahren im Jacobi-Davidson-Eigenwertlöser ersetzen, ohne dabei das Konvergenzverhalten des Algorithmus zu verändern.

Da wir uns von dem verwendeten TT-Format eine Zeitersparnis erhoffen, betrachten wir in Abbildung 6.2a nun die Laufzeiten der beiden Versionen für verschiedene Dimensionen. Hierbei betrachten wir zusätzlich noch die Jada-Varianten mit einer vollbesetzten (dense) Matrix statt der dünnbesetzten (sparse) Matrix. Die Abbildung ist in logarithmischer Darstellung, damit wir den asymptotischen Verlauf der Kurven besser einschätzen können. Als Abbruchkriterium wählen wir $\|res\|_2 = 1e-12$.

Wir können erkennen, dass die TT-Jada-Variante eine längere Laufzeit als die beiden anderen Versionen hat. Der asymptotische Verlauf der TT-Jada-Variante ist weniger steil, als der von der Jada-Variante mit der vollbesetzten Matrix. Diese Variante können wir demnach bei noch größeren Dimensionen in der Laufzeit schlagen.

Die Jada-Variante mit der dünnbesetzten Matrix, welche wir normalerweise beim Jacobi-Davidson-Algorithmus betrachten, hat allerdings eher einen flacheren asymptotischen Verlauf als die TT-Jada-Variante. Hier ist nicht damit zu rechnen, dass wir für größere



(a) Laufzeitvergleich

(b) maximale Bonddimension der Zwischenergebnisse

Abbildung 6.2: SpinOpB periodisch

Probleme Laufzeit durch das Tensor-Train-Format einsparen können.

Wir hatten gehofft, dass die TT-Jada-Variante zumindest asymptotisch schneller wird als die Jada-Variante. Um zu analysieren, wieso dies nicht der Fall ist, betrachten wir in Abbildung 6.2b die maximalen Bonddimensionen der Zwischenergebnisse $\boldsymbol{Q}$ und des Residuums im Verlauf der Iterationen. Diese haben einen Einfluss auf die Laufzeit des Algorithmus, da wir mit diesen TT-Tensoren im TT-GMRES-Verfahren weiter rechnen (vergleiche Algorithmus 4).

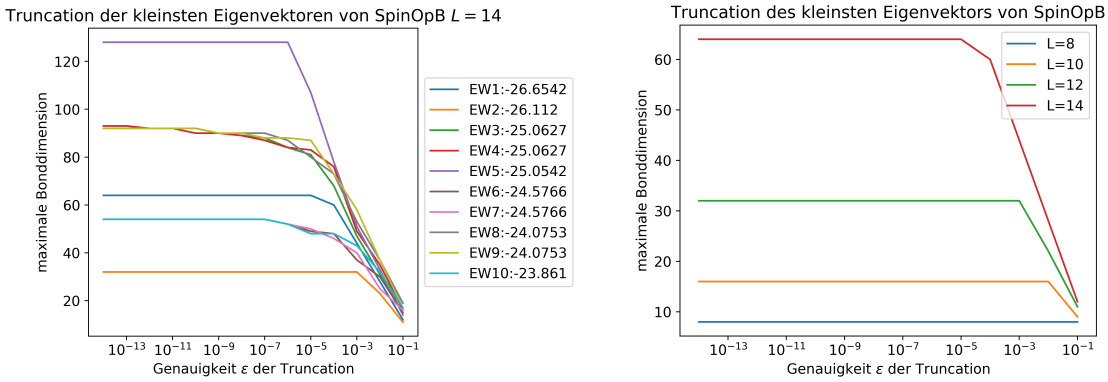
Wir können erkennen, dass beide TT-Tensoren sehr schnell hohe Bonddimensionen annehmen, welche der größtmöglichen Bonddimension $r_{worst} = 2^{\lfloor L/2 \rfloor}$ entsprechen. Wie in Kapitel 5 diskutiert, ergibt sich für den Worst-Case im TT-Format ein höherer Aufwand als mit dünnbesetzten Matrizen und Vektoren. Dies erklärt also unsere Beobachtungen.

Damit haben wir zwar den Grund für unsere Laufzeitergebnisse gefunden, allerdings bleibt die Frage, warum wir trotz Truncation und eines low-rank-approximierbaren Tensoroperators solch hohe Bonddimensionen bei den Zwischenergebnissen erhalten.

Um dies zu untersuchen, betrachten wir in Abbildung 6.3 die maximale Bonddimension unserer gesuchten Lösung, das heißt, die maximalen Bonddimensionen der Eigenvektoren der kleinsten Eigenwerte. Hierbei haben wir verschiedene Genauigkeiten der Truncation betrachtet.

Abbildung 6.3a zeigt die maximalen Bonddimensionen der Eigenvektoren der 10 kleinsten Eigenwerte vom Spinketten-Operator mit $L = 14$ Spins in Abhängigkeit der Toleranz ϵ der Truncation-Genauigkeit. Wir sehen hier, dass wir für die meisten Eigenvektoren sehr hohe Bonddimensionen erhalten. Erst bei einer Toleranz von etwa $\epsilon = 1e-5$ fallen die Bonddimensionen ab und erreichen bei etwa $\epsilon = 1e-2$ einen niedrigen Bereich. Die Eigenvektoren sind demnach für höhere Genauigkeiten nicht low-rank-approximierbar. Mit dieser Erkenntnis ergibt es Sinn, dass die Berechnung der Eigenvektoren auch nicht mit geringer Bonddimension geschieht.

Dies gilt nicht nur für den Spinketten-Operator mit $L = 14$ Spins, sondern auch für



(a) Für die Eigenvektoren der 10 kleinsten Eigenwerte von SpinOpB14

(b) Für den Eigenvektor des kleinsten Eigenwerts für verschiedene Problemgrößen L

Abbildung 6.3: maximale Bonddimension in Abhängigkeit der Toleranz ϵ der Truncation-Genauigkeit

die anderen Problemgrößen. In Abbildung 6.3b können wir erkennen, dass die maximale Bonddimension des Eigenvektors zum kleinsten Eigenwert mit der Problemgröße wächst. Tatsächlich haben wir hier die Abhängigkeit $r_{max} = 2^{(L/2)-1} = r_{worst}/2$.

Mit Berücksichtigung unserer Komplexitätsanalyse in Kapitel 5 sehen wir also, dass diese Bonddimensionen zu groß sind, damit das TT-Format Laufzeit gegenüber der Jada-Variante mit der dünnbesetzten Matrix spart. Die TT-Jada-Variante kann allerdings schneller werden als die Jada-Variante mit der vollbesetzten Matrix, was sich mit unseren Laufzeitergebnissen in Abbildung 6.2a deckt.

Wir haben in dieser Arbeit das Spinketten-Problem betrachtet, da in der Physik das TT-Format für dieses Problem bereits erfolgreich genutzt wird. Hier werden allerdings deutlich größere Spinketten mit Bonddimensionen im Bereich von ein paar 100 bis 1000 bei geringeren Genauigkeit von etwa 1e-5 betrachtet und üblicherweise ALS-artige Eigenwertlöser verwendet. In [24, Kapitel 7] werden beispielsweise Spinketten mit Länge $L = 28$, 32 und 60 betrachtet.

Um zu verstehen, weshalb das TT-Format für diese größeren Probleme in der Literatur gut funktioniert, fügen wir Abbildung 6.3b noch die maximalen Bonddimensionen von einigen größeren Problemen hinzu (siehe Abbildung 6.4). Hier können wir beobachten, dass die Bonddimension $r_{max} = r_{worst}/2$ mit steigendem L immer später und ab einem Punkt nicht mehr angenommen wird. Zusätzlich sehen wir, dass bei den größeren Spinketten die maximale Bonddimension im mittleren Genauigkeitsbereich deutlich unter der Worst-Case-Bonddimension liegt. Für sehr lange Spinketten verbessert sich

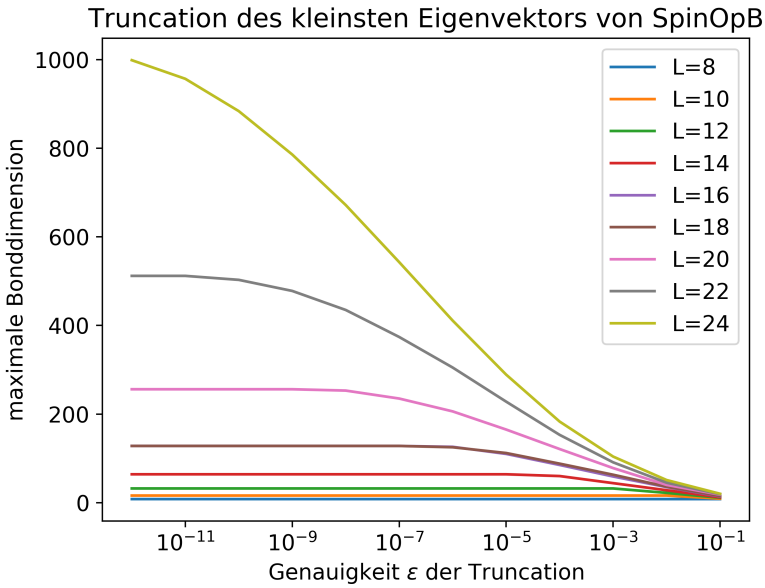


Abbildung 6.4: maximale Bonddimension in Abhängigkeit der Toleranz ϵ der Truncation-Genauigkeit für den Eigenvektor des kleinsten Eigenwerts für verschiedene Problemgrößen L

demnach das Verhalten der Bonddimensionen.

Wir nutzen in der äußeren Iteration unseres Algorithmus noch dicht-gespeicherte Vektoren und berechnen die Eigenvektoren in wesentlich höherer Genauigkeit. Deshalb ist es für uns ungünstig, dass die interessanten Fälle, mit kleinen Bonddimensionen bei großer Genauigkeit, erst bei etwa $L = 30$ starten, da wir für diese Problemgröße an die Grenzen des Arbeitsspeichers zur Speicherung der Vektoren kommen ($2^{30} \cdot 8$ Byte ≈ 9 GByte pro Vektor in doppelter Genauigkeit). Im Folgenden wollen wir uns mit Problemen beschäftigen, welche schon bei kleineren Problemgrößen ein interessantes numerisches Verhalten aufweisen.

6.2 d-dimensionaler Laplace-Operator

Nun betrachten wir den d-dimensionalen Laplace-Operator und seine Eigenwertgleichung $-\Delta_d u = \lambda u$, welche auch Helmholtz-Gleichung genannt wird.

Der eindimensionale Laplace-Operator kann mit dem zentralen Differenzenquotienten durch $\Delta = \text{tridiag}[-1, 2, -1]$ diskretisiert werden. Mit Hilfe des Tensorprodukts lässt sich die Diskretisierung des d-dimensionalen Laplace-Operators wie folgt darstellen:

$$\boldsymbol{\Delta}_d = \Delta \otimes I \otimes \cdots \otimes I + I \otimes \Delta \otimes I \otimes \cdots \otimes I + \cdots + I \otimes \cdots \otimes I \otimes \Delta. \tag{6.2.1}$$

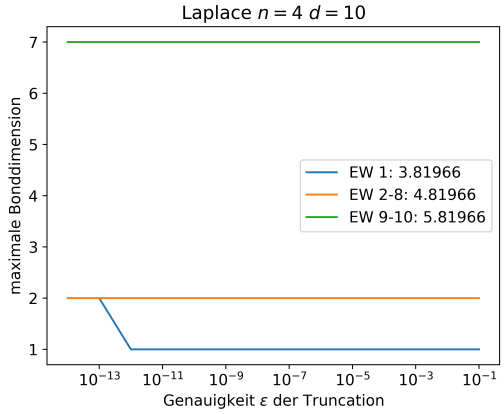
Dies gibt uns eine Tensorzerlegung des Operators. Auf Grund der besonderen Struktur der Diskretisierung des Laplace-Operators ergibt sich eine TT-Darstellung mit maximaler Bonddimension 2, da jeder TT-Kern nur die zwei Operatoren Δ und I speichern muss.

In Abbildung 6.5 betrachten wir die maximalen Bonddimensionen der Eigenvektoren der 10 kleinsten Eigenwerte. Für den Laplace-Operator erhalten wir mehrere identische Eigenwerte. Für zwei Eigenvektoren mit gleichem Eigenwert erhalten wir die selben Kurven in der Abbildung, welche sich gegenseitig überlagern. Deshalb stehen die Kurven hier für Gruppen von identischen Eigenwerten, um die Übersichtlichkeit zu gewährleisten.

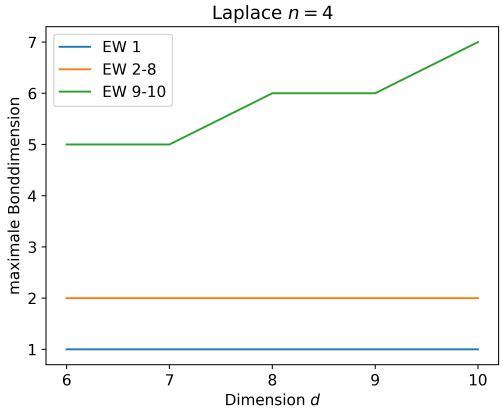
Abbildung 6.5a zeigt die maximalen Bonddimensionen dieser Eigenvektoren in Abhängigkeit der Toleranz ϵ der Truncation-Genauigkeit für den 10-dimensionalen Laplace-Operator mit $n = 4$ Elementen in jeder Dimension. Wir erhalten unabhängig von der Genauigkeit der Truncation sehr niedrige Werte für die Bonddimensionen.

Betrachten wir nun, wie sich die maximalen Bonddimensionen in Abhängigkeit der Dimension d für eine vorgegebene Toleranz $\epsilon = 1e-12$ der Truncation-Genauigkeit verhält (siehe Abbildung 6.5b). Für den kleinsten Eigenwert und die Gruppe der zweitkleinsten Eigenwerte sehen wir, dass diese einen konstanten, von d unabhängigen Wert annehmen. Die maximale Bonddimension der dritt kleinsten Eigenwertgruppe steigt nur leicht mit der Dimension an, sodass sie in einem niedrigen Bereich liegt.

Die selben Ergebnisse erhalten wir auch, wenn wir eine feinere Diskretisierung wählen (vergleiche Abbildung 6.6). Die Eigenvektoren der kleinsten Eigenwerte des d-dimensionalen Laplace-Operators sind demnach low-rank-approximierbar, unabhängig von der Problemgröße.

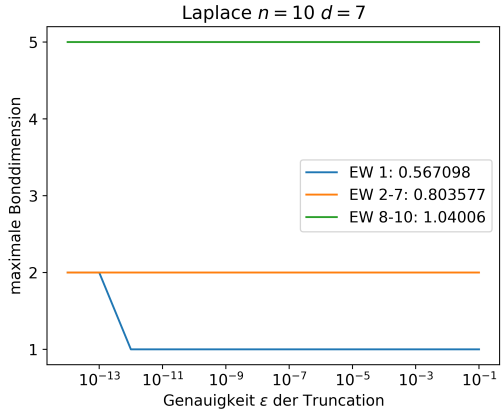


(a) maximale Bonddimension abhängig von der Toleranz ϵ der Truncation-Genauigkeit für $d = 10$

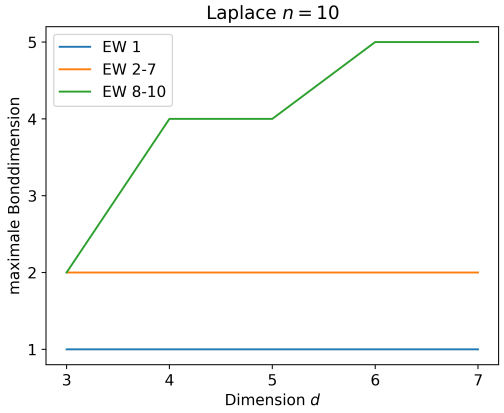


(b) maximale Bonddimension abhängig von der Dimension d mit $\epsilon = 1e - 12$

Abbildung 6.5: maximale Bonddimension der Eigenvektoren der 10 kleinsten Eigenwerte für den d-dimensionalen Laplace-Operator mit einer Diskretisierung von $n = 4$ Elementen in jeder Dimension

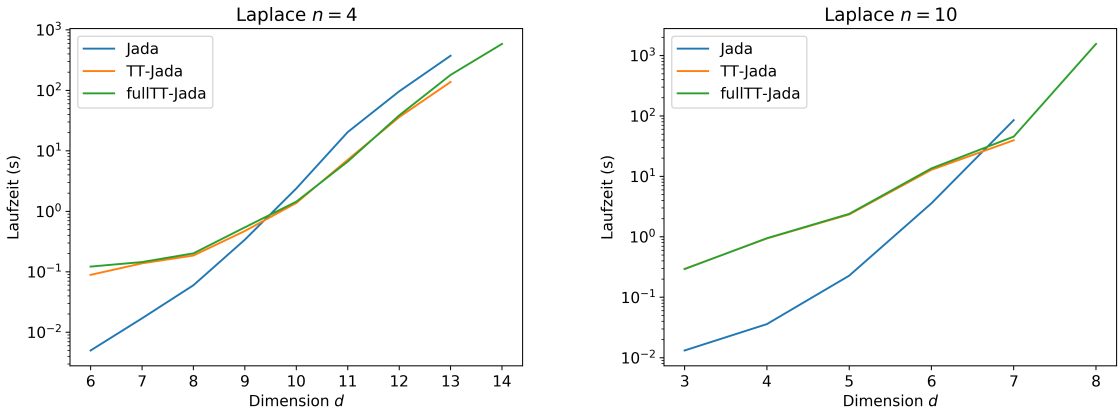


(a) maximale Bonddimension abhängig von der Toleranz ϵ der Truncation-Genauigkeit für $d = 7$



(b) maximale Bonddimension abhängig von der Dimension d mit $\epsilon = 1e - 12$

Abbildung 6.6: maximale Bonddimension der Eigenvektoren der 10 kleinsten Eigenwerte für den d-dimensionalen Laplace-Operator mit einer Diskretisierung von $n = 10$ Elementen in jeder Dimension



(a) $n = 4$ Elemente pro Dimension

(b) $n = 10$ Elemente pro Dimension

Abbildung 6.7: Laufzeitvergleich in Abhängigkeit der Dimension d für den d-dimensionalen Laplace-Operator

Die Tatsache, dass wir sehr niedrige, konstante Werte für die maximalen Bonddimensionen der Eigenvektoren erhalten, lässt uns auf Grund der Komplexitätsabschätzung in Kapitel 5 auf ein besseres Abschneiden der TT-Jada-Variante im Laufzeitvergleich mit der Jada-Variante hoffen. Deshalb führen wir im Folgenden den Vergleich der Jacobi-Davidson-Versionen für den d-dimensionalen Laplace-Operator durch, wobei wir als Abbruchkriterium $\|res\|_2 = 1e - 11$ wählen.

Abbildung 6.7 zeigt die Laufzeitergebnisse für den d-dimensionalen Laplace-Operator in Abhängigkeit der Dimension d. Hierbei betrachten wir eine Diskretisierung mit $n = 4$ Elementen in jeder Dimension (siehe Abbildung 6.7a) und eine Diskretisierung mit $n = 10$ Elementen in jeder Dimension (siehe Abbildung 6.7b). Verglichen werden die Varianten Jada, TT-Jada und fullTT-Jada.

Wir können beobachten, dass die Kurven der Jada- und TT-Jada-Variante nach Dimension $d_{sparse}(n = 4) = 13$ beziehungsweise $d_{sparse}(n = 10) = 7$ abbrechen, während die Kurve der fullTT-Jada-Variante bis $d_{fullTT}(n = 4) = 14$ beziehungsweise $d_{fullTT}(n = 10) = 8$ weiterläuft.

Dies hat den Grund, dass der Speicherbedarf der dünnbesetzten Matrix ab $d_{sparse} + 1$ (etwa $2 \cdot mn^d \cdot 8$ Byte $= 2 \cdot 29 \cdot 4^{14} \cdot 8$ Byte ≈ 125 GByte in doppelter Genauigkeit für den Laplace-Operator mit $n = 4$ und $d = 14$) in der Größenordnung des verfügbaren Arbeitsspeichers, etwa 300 GByte, liegt. Da wir für den Unterraum auch noch einige Vektoren mit jeweils etwa 2 GByte speichern, bricht unsere Rechnung wegen Speichermangels ab. Deshalb können wir mit der Jada und der TT-Jada-Variante nur bis zur Dimension d_{sparse} rechnen.

Die fullTT-Jada-Variante benötigt die dünnbesetzte Matrix nicht, da sie für alle Matrix-Operationen den TT-Operator nutzt. Dieser benötigt viel weniger Speicherplatz (etwa $dn^2r^2 \cdot 8$ Byte $= 14 \cdot 4^2 \cdot 2^2 \cdot 8$ Byte ≈ 7 kByte in doppelter Genauigkeit für den Laplace-Operator mit $n = 4$ und $d = 14$), weshalb diese Variante auf der gleichen Hardware noch eine Dimension größer rechnen kann (vergleiche auch Kapitel 5). Ab $d_{fullTT}+1$

benötigen die Vektoren jedoch so viel Speicherplatz, etwa 9 GByte pro Vektor für $n = 4$ und $d = 15$, dass der Arbeitsspeicher nun durch die Vektoren überläuft und auch diese Variante nicht weiter rechnen kann, obwohl der TT-Operator kaum Speicher benötigt, etwa 8 kByte für $n = 4$ und $d = 15$.

Dies zeigt, dass wir durch das TT-Format Größenordnungen an Speicherplatz einsparen und somit größere Probleme auf der gleichen Hardware lösen können. Eine interessante Überlegung wäre, auch die Vektoren in der äußeren Iteration durch TT-Tensoren zu ersetzen. Dies wird in dieser Arbeit jedoch nicht betrachtet (siehe Kapitel 4).

Bezüglich der Laufzeiten können wir in Abbildung 6.7 erkennen, dass die Jada-Variante einen steileren Anstieg als die TT-Jada-Variante hat. Hierdurch kommt die TT-Jada-Variante bei den größeren Dimension schneller zum Ergebnis als die Jada-Variante. Die fullTT-Jada-Variante ist ein wenig langsamer als die TT-Jada-Variante, aber ansonsten vom Laufzeitverhalten gleich.

Tabelle 6.1 zeigt die Laufzeitergebnisse für den 13-dimensionalen Laplace-Operator mit $n = 4$ Elementen in jeder Dimension aufgeteilt in die Zeit, welche im (TT-)GMRES-Verfahren, das heißt in der inneren Iteration, verbracht wird und der restlichen Zeit des Algorithmus (der Zeit der äußeren Iteration). Hier können wir beobachten, dass bei der Jada-Variante etwa 70% der Laufzeit von der inneren Iteration stammen und etwa 30% vom Rest des Algorithmus.

Bei der TT-Jada-Variante sieht dies anders aus. Hier ist die äußere Iteration, in welcher wir noch die dünnbesetzte Matrix und Vektoren verwenden, wir also eine exponentielle Abhängigkeit der Rechenkomplexität von d haben (siehe Kapitel 5), der aufwendige Teil der Rechnung. Die verbrachte Zeit im TT-GMRES-Verfahren, in welchem wir das TT-Format verwenden, fällt hier kaum noch ins Gewicht (etwa 4% der gesamten Laufzeit). Die TT-Jada-Variante verbraucht zwar in der äußeren Iteration aufgrund der zusätzlichen Formattransformationen etwas mehr Zeit als die Jada-Variante, ist aber insgesamt aufgrund der wesentlich kürzeren Zeit der inneren Iteration schneller.

	Laufzeit	(TT-)GMRES-Zeit	Laufzeit - (TT-)GMRES-Zeit
Jada-Variante	374,231s	266,495s	107,736s
TT-Jada-Variante	137,961s	5.365s	132,596s

Tabelle 6.1: Rechenzeiten für den 13-dimensionalen Laplace-Operator mit $n = 4$ Elementen in jeder Dimension

Zusammengefasst sehen wir, dass die TT-Jada-Variante für den d-dimensionalen Laplace-Operator, welcher low-rank-approximierbare Eigenvektoren hat, bei großen Dimensionen schneller ist als die Variante mit der dünnbesetzten Matrix, da wir einen flacheren Verlauf der Laufzeit aufgrund von einer Zeitersparnis in der inneren Iteration haben. Verwenden wir die fullTT-Jada-Variante, so können wir noch eine Dimension größer lösen, als wenn wir eine dünnbesetzte Matrix verwenden.

6.3 Vergleich mit der ALS-Variante eigb

In unseren bisherigen Experimenten haben wir verschiedene Varianten des selben Algorithmus miteinander verglichen. Nun wollen wir die Tensor-Train-Jacobi-Davidson-Methode mit einer komplett anderen Methode vergleichen. Hierfür verwenden wir die im ttpy Python-Paket implementierte ALS-Variante eigb. Als Testproblem nutzen wir das Spinketten-Problem aus Kapitel 6.1.

Beim Vergleich der verschiedenen Jacobi-Davidson-Varianten untereinander betrachten wir den selben Code mit kleinen Anpassungen für das jeweilige Format. Hierbei scheint ein Laufzeitvergleich sinnvoll zu sein.

Für den folgenden Vergleich ist dies nicht der Fall. Betrachten wir den eigb-Code in ttpy, so sehen wir, dass dieser tatsächlich nur ein Python-Interface ist. Der eigentliche Algorithmus ist optimiert in Fortran implementiert und arbeitet lokal im TT-Format. Für unseren nicht-optimierten Tensor-Train-Jacobi-Davidson-Code nutzen wir Python und rechnen global in zwei verschiedenen Formaten, was Truncations und Hin- und Rücktransformieren mit hohen Kosten erfordert.

Auf Grund dieser Unterschiede wollen wir hier statt der Laufzeiten die Anzahl an Matrix-Vektor-Multiplikationen vergleichen. Hierzu verwenden wir die fullTT-Jada-Variante, um ausschließlich TT-Matrix-Vektor-Multiplikationen zu berechnen. Würden wir die TT-Jada-Variante verwenden, hätten wir ein Gemisch aus TT-Operator-Anwendungen und dünnbesetzten Matrix-Vektor-Multiplikationen, was den Vergleich wieder erschweren würde. Eine globale TT-Matrix-Vektor-Multiplikation in der fullTT-Jada-Variante schätzen wir hier durch d lokale Matrix-Vektor-Multiplikationen des eigb-Algorithmus ab (vergleiche Kapitel 5). Eine lokale Matrix-Vektor-Multiplikation des eigb-Algorithmus ist hierbei eine Matrix-Vektor-Multiplikation auf einem einzelnen TT-Kern und d die Dimension unseres Problems.

Weiterhin scheint die Ausgabe im eigb-Algorithmus die Anzahl der durchgeführten Multiplikationen leicht zu unterschätzen. Bei sehr einfachen Problemen, wie beispielsweise dem Laplace-Problem, gibt er 0 als Anzahl lokaler Matrix-Vektor-Multiplikationen zurück. Aus diesem Grund haben wir dieses Problem nicht als Testproblem verwendet, sondern das Spinketten-Problem.

Zusätzlich ist bei unseren Experimenten noch zu beachten, dass die Bonddimension unserer Startlösung $\text{bond}(\boldsymbol{v}_{start})$ im eigb-Algorithmus in ttpy nur verkleinert werden kann. Damit wir hier sinnvolle Ergebnisse erhalten, müssen wir $\text{bond}(\boldsymbol{v}_{start})$ so groß wählen, dass unser Eigenvektor mit einer maximalen Bonddimension von $\text{bond}(\boldsymbol{v}_{start})$ mit der gewünschten Genauigkeit darstellbar ist.

Abbildung 3.1 zeigt die Anzahl an lokalen Matrix-Vektor-Produkten in logarithmischer Darstellung in Abhängigkeit der Dimension L. Wir vergleichen hier den eigb-Algorithmus mit der fullTT-Jada-Variante und der fullTT-Jada-Variante mit 20 Arnoldi-Startschritten. Hierbei wollen wir bis zu einer Genauigkeit von $\|res\|_2 = \|AQ - \sigma Q\|_2 = 1e-10 = tol$ rechnen. In den fullTT-Jada-Varianten ist dies als Abbruchkriterium eingebaut, sodass wir hier nur $tol_{Jada} = 1e-10$ an der Algorithmus übergeben müssen. Der eigb-Algorithmus hat ein lokales Abbruchkriterium, welches erfüllt sein kann, bevor $\|res\|_2 \leq tol$ gilt. In diesen Fällen haben wir die Kurve gestrichelt gezeichnet. Durch

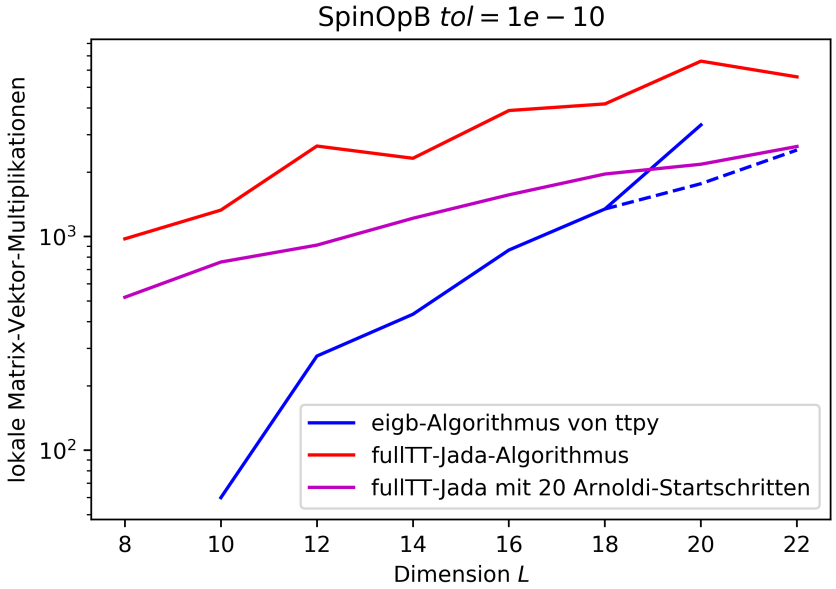


Abbildung 6.8: Anzahl lokaler Matrix-Vektor-Produkte bei Rechnungen bis zur Genauigkeit $\|AQ - Q\sigma\|_2 = 1e - 10$. Dort wo die Linie gestrichelt ist, erreichen wir die gewünschte Genauigkeit nicht. Bei eigb führt ein kleineres Abbruchkriterium dazu, dass die Genauigkeit erreicht werden kann. Hiermit führen wir die durchgezogene Linie fort.

die Wahl einer kleineren Toleranz *tol* für das lokale Kriterium erreichen wir bei $L = 20$ unser globales Abbruchkriterium (durchgezogene blaue Linie). Bei $L = 22$ erhalten wir Speicherprobleme bei kleinerer Toleranz *tol*.

Durch die Arnoldi-Startschritte können wir im fullTT-Jada einiges an Matrix-Vektor-Produkten sparen und die Kurve etwas glätten. Dies hat folgenden Hintergrund: Der Jacobi-Davidson-Algorithmus konvergiert erst, wenn eine gute Startlösung gefunden wurde. Geschieht dies durch die ersten Jada-Iterationen, so kostet dies viele Matrix-Vektor-Produkte in der inneren Iteration und es kann unterschiedlich viele Iterationen dauern, bis eine gute Startlösung gefunden wurde. Bei einer Arnoldi-Iteration wird die innere Iteration weggelassen. Setzen wir also nun eine festgelegte Anzahl an Arnoldi-Iterationen vorweg, so sparen wir die Matrix-Vektor-Multiplikationen der inneren Schleife und glätten die Kurve, da wir für die Suche der Startlösung eine vorgegebene Anzahl an Iterationen haben.

Beim Vergleich der fullTT-Jada-Varianten mit dem eigb-Algorithmus können wir beobachten, dass die Kurven mit zunehmender Dimensionen L aufeinander zu laufen. Die Anzahl an Matrix-Vektor-Multiplikationen steigt in den fullTT-Jada-Varianten also langsamer an als im eigb-Algorithmus. Bei Dimension $L = 20$ benötigt die fullTT-Jada-Variante mit 20 Arnoldi-Startschritten tatsächlich weniger Matrix-Vektor-Multiplikationen als der eigb-Algorithmus, wenn das gewünschte Abbruchkriterium eingehalten wird.

Für große Dimensionen lohnt es sich also möglicherweise, unseren Algorithmus zu betrachten.

6.4 Konvektions-Diffusions-Operator

Im Gegensatz zum eigb-Algorithmus (vergleiche Kapitel 3.2) kann das Jacobi-Davidson-Verfahren auch unsymmetrische Probleme lösen. Deshalb wollen wir uns hier mit einem solchen Problem beschäftigen und wählen als Testproblem das Eigenwertproblem für den Konvektions-Diffusions-Operator. Hierbei betrachten wir zunächst nur einen Fluss in x_1-Richtung: $-\Delta_d u + \alpha\delta_{x_1} u = \lambda u$.

Den Konvektions-Diffusions-Operator diskretisieren wir wie folgt:

$$\begin{aligned}\boldsymbol{Op}_{\text{ConvDiffx1}} &= -\Delta^h \otimes I \otimes \cdots \otimes I + I \otimes \Delta^h \otimes I \otimes \cdots \otimes I + \cdots + I \otimes \cdots \otimes I \otimes \Delta^h \\ &\quad + \alpha \cdot \nabla^h \otimes I \otimes \cdots \otimes I. \end{aligned} \tag{6.4.1}$$

Hierbei ist $\Delta^h = \frac{1}{h^2} \cdot \text{tridiag}[-1, 2, -1]$ der mit dem zentralen Differenzenquotienten diskretisierte Laplace-Operator in einer Dimension und $\nabla^h = \frac{1}{h} \cdot \text{tridiag}[-1, 1, 0]$ die mittels dem Vorwätsdifferenzenquotienten diskretisierte Ableitung in einer Dimension. Diese Tensorzerlegung ist durch die besondere Struktur durch einen TT-Operator mit maximaler Bonddimension 2 darstellbar.

Abbildung 6.9 zeigt die maximalen Bonddimensionen der Eigenvektoren der 10 kleinsten Eigenwerte für den Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung und $n = 4$ Elementen in jeder Dimension. Analog zum d-dimensionalen Laplace-Operator (vergleiche Abbildung 6.5) werden hier wieder identische Eigenwerte, welche die selben Kurven erzeugen, gruppiert. Die maximalen Bonddimensionen dieser Eigenvektoren werden in Abbildung 6.9a in Abhängigkeit der Toleranz ϵ der Truncation-Genauigkeit für Dimension $d = 10$ und in Abbildung 6.9b in Abhängigkeit der Dimension d für die Toleranz $\epsilon = 1e - 10$ der Truncation-Genauigkeit gezeigt.

Wir sehen hier analog zum d-dimensionalen Laplace-Operator, dass die Eigenvektoren sehr kleine maximale Bonddimension bei großer Truncation-Genauigkeit erhalten. Diese sind zum Teil wieder unabhängig von der Dimension d beziehungsweise wachsen nur sehr leicht mit wachsender Dimension. Wir haben also auch beim Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung low-rank-approximierbare Eigenvektoren, unabhängig von der Problemgröße.

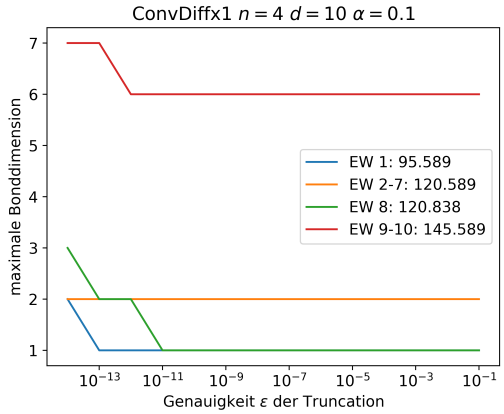
Abbildung 6.10 zeigt die Laufzeitergebnisse für den Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung, $n = 4$ Elementen in jeder Dimension und $\alpha = 0.1$. Hierbei wurden wieder die Jada-Variante, die TT-Jada-Variante und die fullTT-Jada-Variante betrachtet. Als Abbruchkriterium dient hier $\|res\|_2 = \|AQ - \sigma Q\|_2 = 1e - 9$.

Wir erhalten hier ähnliche Ergebnisse wie beim d-dimensionalen Laplace-Operator (vergleiche Abbildung 6.7).

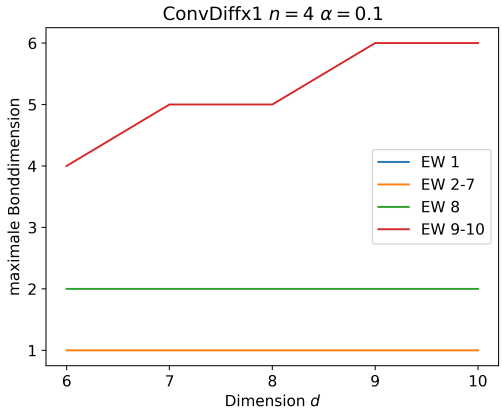
Ab Dimension $d = 13$ ist die dünnbesetzte Matrix wieder zu groß für den Arbeitsspeicher. Auch hier können wir beobachten, dass wir durch die fullTT-Jada-Variante noch eine Dimensionen größer auf der selben Hardware rechnen können, bevor auch die Vektoren zu viel Speicherplatz benötigen.

Im Vergleich der Varianten untereinander sehen wir, dass diese bei $d = 9$ etwa gleiche Laufzeiten haben und die TT-Jada-Variante ab $d = 10$ schneller ist als die Jada-Variante.

Analog zum d-dimensionalen Laplace-Operator können wir hier also die low-rank-approximierbaren Eigenvektoren des Konvektions-Diffusions-Operators mit Fluss in x_1-



(a) maximale Bonddimension abhängig von der Toleranz ϵ der Truncation-Genauigkeit für $d = 10$



(b) maximale Bonddimension abhängig von der Dimension d mit $\epsilon = 1e - 10$

Abbildung 6.9: maximale Bonddimension der Eigenvektoren der 10 kleinsten Eigenwerte für den Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung mit einer Diskretisierung von $n = 4$ Elementen in jeder Dimension

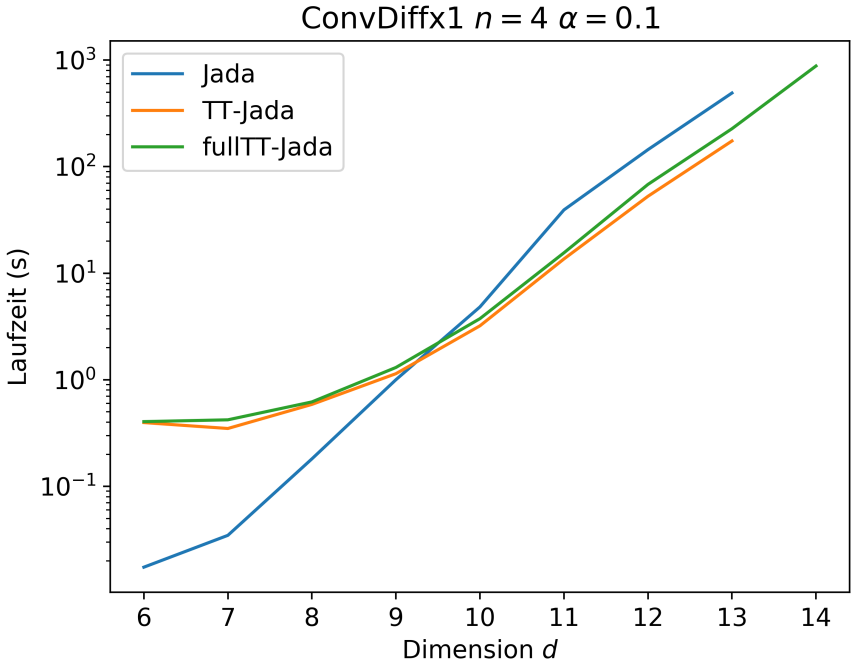


Abbildung 6.10: Laufzeitvergleich für ConvDiffx1 mit $n = 4$, $\alpha = 0.1$

Richtung bei großen Dimensionen mit der TT-Jada-Variante schneller berechnen als mit der üblichen Jada-Variante mit dünnbesetzter Matrix.

Zusätzlich betrachten wir noch den Konvektions-Diffusions-Operator mit einem Fluss quer durch alle Dimensionen: $-\Delta_d u + \alpha \nabla_d u = \lambda u$.

Analog zu oben können wir dieses Problem diskretisieren durch

$$\begin{aligned}\boldsymbol{Op}_{\text{ConvDiff}} = &-\Delta^h \otimes I \otimes \cdots \otimes I + I \otimes \Delta^h \otimes I \otimes \cdots \otimes I + \cdots + I \otimes \cdots \otimes I \otimes \Delta^h \\ &+ \alpha \cdot (\nabla^h \otimes I \otimes \cdots \otimes I + I \otimes \nabla^h \otimes I \otimes \cdots \otimes I + \cdots + I \otimes \cdots \otimes I \otimes \nabla^h).\end{aligned} \tag{6.4.2}$$

Auch diesen Operator können wir durch einen TT-Operator mit maximaler Bonddimension 2 darstellen.

Im Folgenden wollen wir den Konvektions-Diffusions-Operator wieder mit $n = 4$ Elementen in jeder Dimension betrachten. Um einen Fluss mit der selben Geschwindigkeit $\alpha_{quer} = 0.1$ wie beim Fluss in eine Dimension zu erhalten, müssen wir die Fließgeschwindigkeit in die einzelnen Richtungen α wie folgt anpassen:

$$\alpha = \sqrt{\alpha_{quer}^2 / d}. \tag{6.4.3}$$

In Abbildung 6.11 betrachten wir für den Konvektions-Diffusions-Operator wieder die gleichen Plots, welche wir in Abbildung 6.9 für den Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung betrachtet haben.

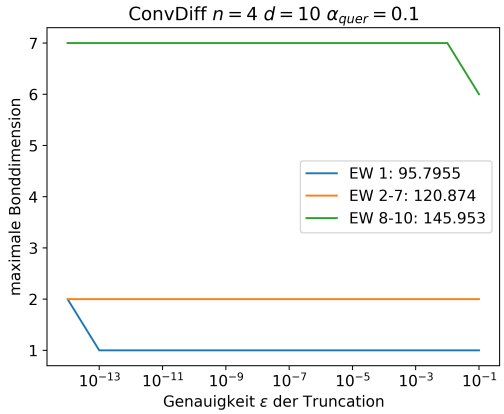
Auch für diesen Operator erhalten wir kleine, konstante Bonddimensionen für die Eigenvektoren der 10 kleinsten Eigenwerte. Wir erwarten also ein ähnliches Verhalten wie bei dem Operator mit Fluss in einer Dimension.

Betrachten wir nun die Laufzeiten der Varianten Jada- und TT-Jada bis zu einer Genauigkeit von $\|res\|_2 = 1e - 10$ in Abbildung 6.12, so hat die TT-Jada-Variante schlechtere Laufzeit als die Jada-Variante. Da beide Kurven etwa gleiche Steigung haben, ist davon auszugehen, dass die TT-Jada-Variante auch bei größeren Dimensionen nicht schneller sein wird als die Jada-Variante.

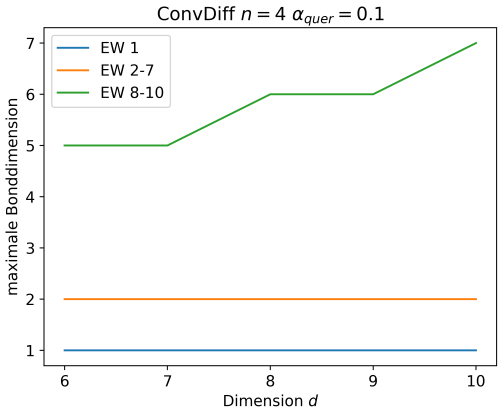
Bisher hatten wir bei kleinen Bonddimensionen der Eigenvektoren stets gutes Verhalten der Laufzeit im TT-Format. Die Frage ist nun, warum dies hier nicht der Fall ist.

Hierzu schauen wir uns in Abbildung 6.13 die maximalen Bonddimensionen der Zwischenergebnisse $\boldsymbol{Q}$ und des Residuums $\boldsymbol{res}$ im Verlauf der Iterationen an. Diese spielen bei unserer Laufzeit eine Rolle, da wir mit diesen TT-Tensoren im TT-GMRES-Verfahren weiter rechnen. Haben diese Tensoren der äußeren Iteration also hohe Bonddimensionen im TT-Format, so wird in der inneren Iteration mit hohen Bonddimensionen gerechnet und somit nach Kapitel 5 mit dem TT-Format eine schlechtere Laufzeit erwartet als mit der Jada-Variante.

Um unsere Ergebnisse vergleichen zu können, betrachten wir zu den Ergebnissen des Konvektions-Diffusions-Operators mit einem Fluss quer durch alle Dimensionen (Abbildung 6.13b) auch die Ergebnisse für den Konvektions-Diffusions-Operator mit Fluss in



(a) maximale Bonddimension abhängig von der Toleranz ϵ der Truncation-Genauigkeit für $d = 10$



(b) maximale Bonddimension abhängig von der Dimension d mit $\epsilon = 1e - 10$

Abbildung 6.11: maximale Bonddimension der Eigenvektoren der 10 kleinsten Eigenwerte für den Konvektions-Diffusions-Operator mit einer Diskretisierung von $n = 4$ Elementen in jeder Dimension

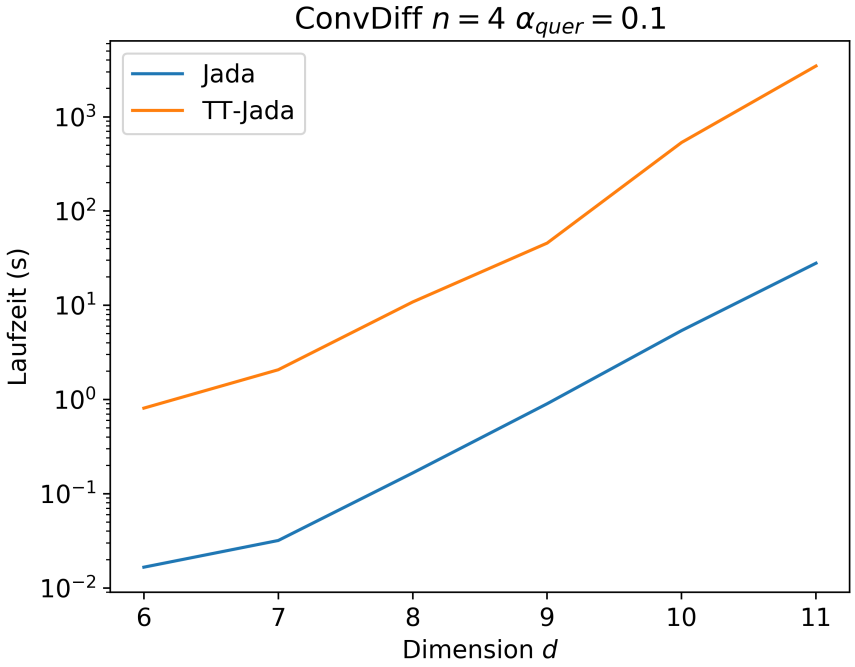
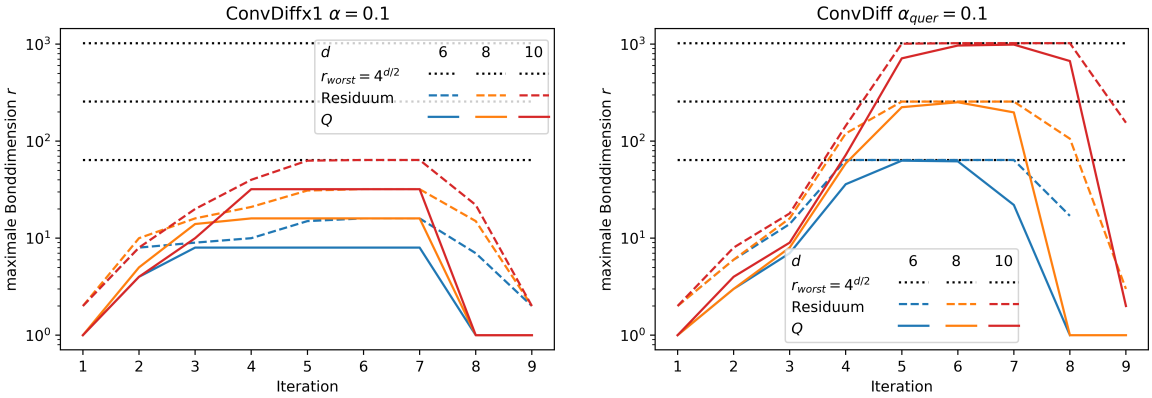


Abbildung 6.12: Laufzeitvergleich für ConvDiff mit $n = 4$, $\alpha_{quer} = 0.1$



(a) Konvektion in x_1-Richtung mit $\alpha = 0.1$

(b) Konvektion quer durch alle Dimensionen mit $\alpha_{quer} = 0.1$

Abbildung 6.13: maximale Bonddimension der Zwischenergebnisse $\boldsymbol{Q}$ und des Residuums $\boldsymbol{res}$ im Verlauf der Iterationen für das Konvektions-Diffusions-Problem mit $n = 4$ Elementen in jeder Dimension

x_1-Richtung (Abbildung 6.13a) mit den selben Parametern. Die schwarz-gepunkteten Linien zeigen die maximal mögliche Bonddimension $r_{worst} = 4^{d/2}$ für die drei betrachteten Dimension d an.

In beiden Abbildungen können wir beobachten, dass wir zu Beginn sehr kleine Bonddimensionen haben. Diese wachsen in den ersten Iterationen an, bis sie einen Wert erreichen, auf dem sie dann für einige Iterationen bleiben. In den letzten Iterationen sinken die Bonddimensionen wieder ab.

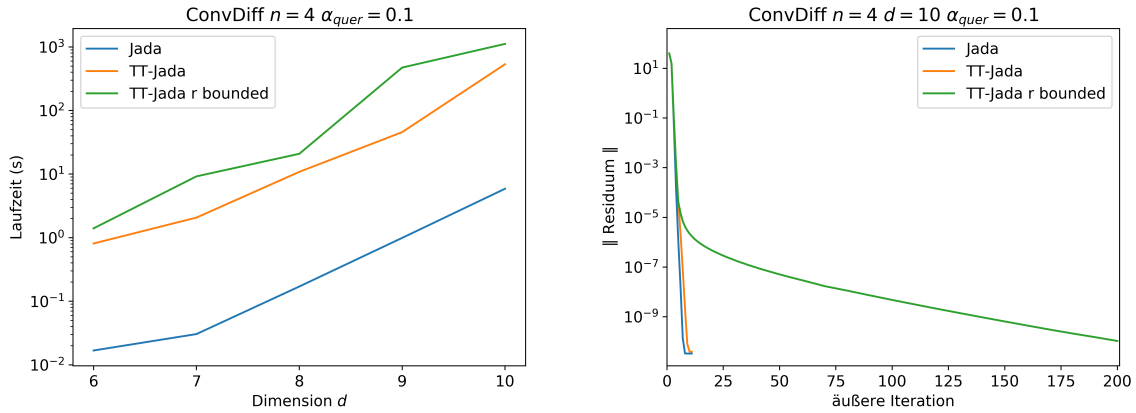
Worin sich die beiden Probleme unterscheiden, ist der maximale Wert, den die maximalen Bonddimensionen im Verlauf der Kurve annehmen.

Beim Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung bleiben die Bonddimensionen weit unter r_{worst}. Sie nehmen maximal den Wert $r = 2 \cdot 4^{d/4}$ an. In Kapitel 5 haben wir gesehen, dass für eine solche maximale Bonddimension die Komplexität der Matrix-Vektor-Operationen im TT-Format einen leicht besseren Verlauf hat als bei der Verwendung einer dünnbesetzten Matrix und Vektoren. Zusätzlich haben wir in den ersten und letzten Iterationen geringere Bonddimensionen, sodass wir bei großen Dimensionen zu der guten Laufzeit in Abbildung 6.10 kommen.

Beim Konvektions-Diffusions-Operator mit dem Fluss quer durch alle Dimensionen nehmen die Bonddimensionen den maximalen Wert r_{worst} an. Hierdurch werden diese mittleren Iterationen sehr teuer. Da wir viele solche Iterationen haben, erhalten wir insgesamt eine hohe Laufzeit, wie in Abbildung 6.12 zu sehen ist, und bleiben langsamer als mit der Jada-Variante ohne TT-Format.

Da unsere Eigenvektoren low-rank-approximierbar sind, sehen wir keine Notwendigkeit für die großen Bonddimensionen der Zwischenergebnisse im Problem mit dem Konvektions-Diffusions-Operator mit dem Fluss quer durch alle Dimensionen. Eine interessante Frage ist, ob wir die großen Bonddimensionen der Zwischenergebnisse vermeiden können.

Die einfachste Lösung wäre, eine Schranke R für die maximale Bonddimension der TT-



(a) Laufzeitvergleich

(b) Residuenverlauf für $d = 10$

Abbildung 6.14: beschränkte maximale Bonddimension $r \leq R = 2 \cdot 4^{d/4}$ beim Lösen des Eigenwertproblems für ConvDiff mit $\alpha_{quer} = 0.1$ und $n = 4$

Tensoren vorzugeben. Dies würde allerdings zu einem Verlust an Genauigkeit führen, da eventuell nicht alle Tensoren mit einer solchen maximalen Bonddimension in der gewünschten Genauigkeit im TT-Format darstellbar sind.

Um herauszufinden, wie sich dies auf unseren Tensor-Train-Jacobi-Davidson-Algorithmus auswirkt, haben wir genau diesen Lösungsansatz ausprobiert. Hierbei haben wir als maximale Bonddimension den Wert $R = 2 \cdot 4^{d/4}$ gewählt. Dies ist der maximale Wert, den die Bonddimensionen der Zwischenergebnisse beim Problem mit dem Konvektions-Diffusions-Operator mit Fluss in x_1-Richtung annehmen (vergleiche Abbildung 6.13a).

Abbildung 6.14a zeigt die Laufzeit der TT-Jada-Variante mit der beschränkten Bonddimension im Vergleich mit der Laufzeit der Jada-Variante und der TT-Jada-Variante ohne Beschränkung der Bonddimension, wobei bis zu einer Genauigkeit von $\|res\|_2 = 1e - 10$ gerechnet wurde. Hier ist zu sehen, dass wir durch das Beschränken der Bonddimension schlechtere Laufzeit erhalten als bei der unbeschränkten Variante.

Den Grund hierfür zeigt uns der Residuenverlauf in Abbildung 6.14b. Solange wir uns in den ersten Iterationen befinden, in welchen die Bonddimensionen der Zwischenergebnisse noch nicht so hoch sind, wir also nicht so viel Genauigkeit verlieren, erhalten wir einen steil abfallenden Residuenverlauf, welchen wir auch bei der Jada und der TT-Jada-Variante erhalten.

Ab Iteration 5 sind die Bonddimensionen der Zwischenergebnisse so hoch, dass wir sie durch unsere Beschränkung wesentlich reduzieren (vergleiche Abbildung 6.13) und die TT-Darstellung ungenauer wird. Hier geht der Residuenverlauf in eine flachere Kurve über. Wir beobachten auch für diesen Fall konvergentes Verhalten, brauchen hierdurch allerdings um die 200 Iterationen statt wie zuvor 9.

Da der Algorithmus nun etwa 20-mal so viele Iterationen benötigt, bringt es nichts, dass die einzelnen Iterationen etwas weniger Zeit kosten. Deshalb erhalten wir die höhere Laufzeit in Abbildung 6.14a.

Durch das einfache Beschränken der maximalen Bonddimension umgehen wir demnach

zwar die hohen Bonddimensionen, verlieren aber das quadratische Konvergenzverhalten der Jacobi-Davidson-Methode. Die Frage bleibt also offen, ob wir die hohen Bonddimensionen der Zwischenergebnisse mit anderen Mitteln vermeiden können, um auch hier mit dem Tensor-Train-Format Laufzeit zu sparen.

7 Fazit

Wir konnten zeigen, dass wir das Tensor-Train-Format in der inneren Iteration des Jacobi-Davidson-Algorithmus verwenden können und dabei das quadratische Konvergenzverhalten beibehalten. Verwenden wir auch in der äußeren Iteration den TT-Operator statt der dünnbesetzten Matrix, so können wir durch den geringeren Speicherverbrauch des Operators im TT-Format größere Problemgrößen auf der selben Hardware betrachten.

Der Vergleich der Komplexitätsabschätzungen verschiedener Operationen im Tensor-Train-Format und bei der Verwendung einer dünnbesetzten Matrix und Vektoren hat gezeigt, dass die maximale Bonddimension der TT-Tensoren im Verhältnis zur Problemgröße klein sein muss, damit wir den Rechenaufwand durch diesen Ansatz verringern können.

Dies konnten wir auch in unseren numerischen Experimenten beobachten. Haben die zu berechnenden Eigenvektoren eine hohe maximale Bonddimension, so treten auch während der Berechnung hohe Bonddimensionen auf, wodurch wir im Vergleich zur Verwendung der dünnbesetzten Matrix und Vektoren hohe Laufzeiten bei der Verwendung des TT-Formats erhalten. Sind die Eigenvektoren hingegen low-rank-approximierbar und die Zwischenergebnisse während der Rechnung haben ebenfalls niedrige Bonddimensionen, so kommen wir bei großen Dimensionen mit dem TT-Format schneller zum Ergebnis.

Es reicht hierbei nicht, dass die Eigenvektoren low-rank-approximierbar sind. Für manche Probleme mit low-rank-approximierbaren Eigenvektoren nehmen die Bonddimensionen der Zwischenergebnisse während der Rechnung hohe Werte an. Da die Rechnung mit hohen Bonddimensionen durchgeführt wird, kommt es auch hier im TT-Format zu höheren Laufzeiten als bei der Verwendung einer dünnbesetzten Matrix und Vektoren.

Hohe Bonddimensionen der Zwischenergebnisse können wir vermeiden, indem wir eine kleinere maximale Bonddimension bei der Berechnung vorgeben. Dadurch reduziert sich jedoch die Genauigkeit der TT-Operationen in der Rechnung. Dies bewirkte bei unseren Experimenten, dass das quadratische Konvergenzverhalten des Jacobi-Davidson-Algorithmus verloren ging. Durch die stark erhöhte Anzahl an benötigten Iterationen ist die Laufzeit gegenüber der TT-Version mit unbeschränkter Bonddimension noch weiter angestiegen. Eine kleine maximale Bonddimension fest vorzugeben, hilft in einem solchen Fall demnach nicht.

Beim Vergleich unseres Tensor-Train-Jacobi-Davidson-Verfahrens mit dem auf dem ALS-Verfahren basierenden eigb-Algorithmus aus dem Python-Paket ttpy haben wir festgestellt, dass wir bei großen Dimensionen weniger Matrix-Vektor-Operationen benötigen als der eigb-Algorithmus. Es könnte sich also lohnen, in einem solchen Fall einen Jacobi-Davidson-Algorithmus zu verwenden.

Ein weiterer Vorteil gegenüber dem eigb-Algorithmus ist, dass wir mit dem Tensor-Train-Jacobi-Davidson-Verfahren auch unsymmetrische Probleme lösen können, was mit

dem eigb-Algorithmus nicht möglich ist.

In der eigb-Implementation in ttpy werden die Bonddimensionen in jeder Iteration verringert oder bleiben gleich. Hierdurch besteht das Problem mit den wachsenden Bonddimensionen der Zwischenergebnisse für dieses Verfahren nicht. Ein Nebeneffekt ist allerdings, dass man durch die Startlösung die maximale Bonddimension der Lösung vorgibt. Ist die gesuchte Lösung nur mit einer größeren Bonddimension darstellbar, so kann diese Implementierung die Lösung nicht finden.

Wir haben in dieser Arbeit nur die Berechnung eines Eigenpaares betrachtet. Mit dem Jacobi-Davidson-QR-Algorithmus kann man sukzessiv mehrere Eigenpaare berechnen. Dazu verwendet man die Projektion des Problems auf das orthogonale Komplement der bereits konvergierten Eigenvektoren. Hier wäre es interessant zu sehen, wie dies am besten im TT-Format umzusetzen ist. Wählt man zum Speichern der konvergierten Eigenvektoren einzelne TT-Tensoren oder fasst man diese besser zu einem TT-Operator zusammen? Und ergeben sich dabei vielleicht wieder Probleme mit der Genauigkeit der Approximation und dadurch den Konvergenzeigenschaften?

Des Weiteren könnte man sich mit der Verwendung des Tensor-Train-Formats in der äußeren Iteration des Jacobi-Davidson-Verfahrens beschäftigen. Hier stellt sich die Frage, ob durch die zusätzliche Ungenauigkeit in der äußeren Iteration die Konvergenz des Verfahrens beeinträchtigt wird. Allerdings könnten wir durch die TT-Darstellung eventuell Speicherplatz sparen und somit größere Probleme berechnen.

Außerdem bleibt die Frage ungeklärt, ob man bei Problemen mit low-rank-approximierbaren Eigenvektoren die möglicherweise hohen Bonddimensionen bei den Zwischenergebnissen vermeiden kann. Nur die maximale Bonddimension zu beschränken funktioniert hier zwar nicht, aber vielleicht lassen sich die hohen Bonddimensionen auf andere Weise vermeiden.

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Ort, Datum Unterschrift

Literaturverzeichnis

[1] Wolfgang Nolting. *Quantentheorie des Magnetismus: Teil 2: Modelle*. Teubner Verlag, 1986.

[2] Patrick Gelß. *The Tensor-Train Format and Its Applications: Modeling and Analysis of Chemical Reaction Networks, Catalytic Processes, Fluid Flows, and Brownian Dynamics*. PhD thesis, 2017.

[3] Ivan V Oseledets. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[4] Lars Grasedyck, Daniel Kressner, and Christine Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[5] Gerard LG Sleijpen and Henk A Van der Vorst. A Jacobi–Davidson Iteration Method for Linear Eigenvalue Problems. *SIAM review*, 42(2):267–293, 2000.

[6] Sergey V Dolgov. TT-GMRES: solution to a linear system in the structured tensor format. *Russian Journal of Numerical Analysis and Mathematical Modelling*, 28(2):149–172, 2013.

[7] Ivan V Oseledets. ttpy. `https://github.com/oseledets/ttpy`, 2013. Zugriffsdatum: 28.11.2019.

[8] Sebastian Holtz, Thorsten Rohwedder, and Reinhold Schneider. The Alternating Linear Scheme for Tensor Optimization in the Tensor Train Format. *SIAM Journal on Scientific Computing*, 34(2):A683–A713, 2012.

[9] Sergey V Dolgov, Boris N Khoromskij, Ivan V Oseledets, and Dmitry V Savostyanov. Computation of extreme eigenvalues in higher dimensions using block tensor train format. *Computer Physics Communications*, 185(4):1207–1216, 2014.

[10] Roger Penrose. Applications of Negative Dimensional Tensors. *Combinatorial Mathematics and its Applications*, 1:221–244, 1971.

[11] Vin De Silva and Lek-Heng Lim. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1084–1127, 2008.

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

[13] Lars Grasedyck and Wolfgang Hackbusch. An Introduction to Hierarchical (h-) Rank and TT-Rank of Tensors with Examples. *Computational Methods in Applied Mathematics*, 11(3):291–304, 2011.

[14] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[15] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. Manchester University Press, 1992.

[16] Charles F Van Loan and Gene H Golub. *Matrix Computations*. Johns Hopkins University Press Baltimore, 1983.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[18] Steffen Börm and Christian Mehl. *Numerical Methods for Eigenvalue Problems*. Walter de Gruyter, 2012.

[19] Gerard LG Sleijpen, Albert GL Booten, Diederik R Fokkema, and Henk A Van der Vorst. Jacobi-davidson type methods for generalized eigenproblems and polynomial eigenproblems. *BIT Numerical Mathematics*, 36(3):595–633, 1996.

[20] Diederik R Fokkema, Gerard LG Sleijpen, and Henk A Van der Vorst. Jacobi–Davidson Style QR and QZ Algorithms for the Reduction of Matrix Pencils. *SIAM Journal on Scientific Computing*, 20(1):94–125, 1998.

[21] Assa Auerbach. Interacting Electrons and Quantum Magnetism. 1994.

[22] Ralph Skomski et al. *Simple Models of Magnetism*. Oxford University Press on Demand, 2008.

[23] Wolfgang Nolting. *Quantentheorie des Magnetismus: Teil 1 Grundlagen*. Teubner Verlag, 1986.

[24] Steven R White. Density-matrix algorithms for quantum renormalization groups. *Physical Review B*, 48(14):10345, 1993.

[25] Melven Röhrig-Zöllner. Parallel solution of large sparse eigenproblems using a Block-Jacobi-Davidson method. Master's thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen, Germany, 2014.