Eigenwertproblem von hierarchischen Matrizen mit lokalem Rang 1

Diplomarbeit

von
Jessica Gördes

vorgelegt an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Christian-Albrechts-Universität zu Kiel

betreut durch Dr. Dr. h.c. Wolfgang Hackbusch

Mai 2009

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bei der Diskretisierung von elliptischen Differentialgleichungen und Integralgleichungen entstehen oft Gleichungssysteme von sehr hoher Dimension. Matrixoperationen von quadratischem oder kubischem Aufwand sind bei solchen Dimensionen nicht mehr praktikabel. Die Approximation dieser vollbesetzten Matrizen durch hierarchische Matrizen erlaubt neben einer speichergünstigen Darstellung auch die Matrixoperationen, wie die Matrix-Vektor-Multiplikation, die Matrix-Multiplikation und die Inversion, mit fast linearem, also nahezu optimalem Aufwand durchzuführen.

Bei Problemstellungen aus der Physik stellt sich oft die Frage nach den Eigenwerten einer Matrix, bei einigen Problemen sogar nach allen Eigenwerten.

Eigenwertprobleme für hierarchische Matrizen sind bisher wenig untersucht worden. Um sich dieser Aufgabe anzunähern, beschäftigt sich diese Arbeit mit der Suche nach allen Eigenwerten (und allen Eigenvektoren) von einem einfachen Modellformat für hierarchische Matrizen. Dabei werden hier zunächst nur symmetrische Modellmatrizen mit lokalem Rang 1 betrachtet.

Für diese speziellen hierarchischen Matrizen wird ein Algorithmus konstruiert, angelehnt an den Divide-and-Conquer-Algorithmus, den Cuppen 1981 für Tridiagonalmatrizen vorgestellt hat und der 1987 von Dongarra und Sorensen noch weiterentwickelt wurde.

Mit einem solchen Algorithmus lassen sich Rückschlüsse auf die weniger speziellen Fälle wie das Eigenwertproblem für das Modellformat mit einem allgemeinen lokalen Rang k oder auch für allgemeinere hierarchische Matrixstrukturen ziehen.

In Kapitel 2 werden zunächst ein paar grundlegende Definitionen aufgeführt, die im weiteren Verlauf der Arbeit benutzt werden und die Nullstellenverfahren der Bisektion und des Newton-Verfahrens kurz erläutert.

Kapitel 3 führt den Begriff der Rang-k-Matrizen ein und gibt eine Definition für das Modell der hierarchischen Matrizen an.

Der Algorithmus von Cuppen für Tridiagonalmatrizen wird in der er-

sten Hälfte von Kapitel 4 vorgestellt, einschließlich der Lösungsmethode für die im Conquer-Teil entstehende rationale Gleichung, der sogenannten Säkulärgleichung. In der zweiten Hälfte des Kapitels wird der Algorithmus auf das hierarchische Matrix-Modell mit lokalem Rang 1 übertragen und die Unterschiede der dort entstehenden Gleichung zur Säkulärgleichung betrachtet sowie eine Lösungsmöglichkeit dazu vorgestellt. Am Ende des Kapitels wird die Eigenvektorberechnung mittels inverser Vektoriteration, speziell zugeschnitten auf dieses Problem, betrachtet.

Kapitel 5 gibt einen genauen Plan des Algorithmus in Form von Pseudocode an, sowie Aufwandsbetrachtungen dazu und ein Beispiel einer praktischen Implementierung.

In Kapitel 6 sind schließlich die Ergebnisse und noch offenen Fragen dieser Arbeit zusammengestellt.

Danksagung:

Für die Vergabe des Themas und die hilfreichen Anregungen danke ich Herrn Prof. Dr. Dr. Wolfgang Hackbusch.

Mein besonderer Dank gilt Herrn Prof. Dr. Steffen Börm, der mir stets mit spontaner Hilfe bei der Suche nach Lösungswegen bei den unterschiedlichen fachlichen (und nicht fachlichen) Problemen zur Seite stand.

Ebenfalls danken möchte ich Herrn Jens Burmeister für seine engagierte Unterstützung.

Außerdem bedanke ich mich ganz herzlich bei meinem Vater.

Kapitel 2

Grundlagen

Dieses Kapitel gibt eine kurze Übersicht der mathematischen Grundlagen, die in dieser Arbeit benutzt werden.

2.1 Eigenwertprobleme

Sei $\mathbb{K}$ ein Körper und $A \in \mathbb{K}^{n\times n}$ eine Matrix.

Definition 1 (Eigenwertproblem, Eigenwert und Eigenvektor) *Die folgende Aufgabe wird als* ***Eigenwertproblem*** *bezeichnet: Suche $\lambda \in \mathbb{K}$ und $x \in \mathbb{K}^n \setminus \{0\}$ mit*

$$A\,x \;=\; \lambda\,x\,.$$

Dabei heißt λ ***Eigenwert*** *und x der* ***Eigenvektor zu*** *λ. Das Tupel (λ, x) wird dann als* ***Eigenpaar*** *bezeichnet.*

Definition 2 (Spektrum) *Das* ***Spektrum*** *einer Matrix A ist die Menge aller Eigenwerte von A und wird mit $\sigma(A)$ bezeichnet:*

$$\sigma(A) \;=\; \{\lambda \in \mathbb{K} \mid \exists\, x \in \mathbb{K}^n \setminus \{0\} : (\lambda, x) \text{ Eigenpaar zu } A\}$$

Definition 3 (charakteristisches Polynom) *Das* ***charakteristische Polynom*** *von A ist für $\lambda \in \mathbb{K}$ gegeben durch*

$$p(\lambda) \;=\; \det\left(A - \lambda I\right).$$

Die Nullstellen des charakteristischen Polynoms von A sind genau die Eigenwerte.

Lemma 1 *λ ist genau dann ein Eigenwert von A, wenn $p(\lambda) = 0$ gilt.*

Beweis: Siehe [5, Bemerkung 4.2.1.].

2.2 Ähnlichkeitstransformationen

Definition 4 (ähnliche Matrizen) *Zwei Matrizen A und B aus $\mathbb{K}^{n\times n}$ heißen **ähnlich**, falls es eine invertierbare Transformationsmatrix $T \in \mathbb{K}^{n\times n}$ gibt, so dass*

$$A \;=\; T\,B\,T^{-1} \tag{2.1}$$

*gilt. (2.1) heißt dabei **Ähnlichkeitstransformation**.*

Ähnliche Matrizen haben die gleichen Eigenwerte:

Lemma 2 *Seien $A, B \in \mathbb{K}^{n\times n}$ ähnlich. Dann gilt*

$$p_A(\lambda) \;=\; p_B(\lambda) \quad \text{und} \quad \sigma(A) \;=\; \sigma(B)\,.$$

Beweis: Siehe [5, Lemma 4.2.2.].

Definition 5 (Diagonalisierbarkeit) *$A \in \mathbb{K}^{n\times n}$ heißt **diagonalisierbar**, falls A ähnlich zu einer Diagonalmatrix ist.*

Definition 6 (unitäre Matrizen) *Eine Matrix $U \in \mathbb{K}^{n\times n}$ heißt **unitär**, falls $U^H = U^{-1}$ gilt.*

Lemma 3 *Das Produkt zweier unitärer Matrizen ist wieder unitär.*

Beweis: Seien U, V unitär, dann gilt:

$$(UV)^{-1} = V^{-1}U^{-1} = V^H U^H = (UV)^H\,.$$

□

Definition 7 (unitär ähnliche Matrizen) *Zwei ähnliche Matrizen A, B heißen **unitär ähnlich**, wenn es eine unitäre Transformationsmatrix U gibt, so dass*

$$A \;=\; U\,B\,U^H \tag{2.2}$$

*gilt. (2.2) heißt dann **unitäre Ähnlichkeitstransformation**.*

Definition 8 (normale Matrizen) *Eine Matrix $A \in \mathbb{K}^{n\times n}$ heißt **normal**, falls $A^H A = A\,A^H$ gilt.*

Bemerkung 1 *Hermitesche und unitäre Matrizen sind normal. (Also ist auch eine symmetrische Matrix aus $\mathbb{R}^{n\times n}$ normal.)*

2.3 Schur-Zerlegung

Satz 1 (Schur-Zerlegung) *Zu jeder Matrix $A \in \mathbb{C}^{n\times n}$ gibt es eine unitäre Matrix U, so dass*

$$U^H A U = \begin{bmatrix} \lambda_1 & * & \dots & * \\ & \lambda_2 & \ddots & \vdots \\ & & \ddots & * \\ 0 & & & \lambda_n \end{bmatrix}$$

mit $\lambda_i \in \sigma(A)$. Die λ_i sind dann genau die Eigenwerte von A.

Beweis: Siehe [7, Satz 1.4.1.].

Falls die Matrix normal ist, gilt die Aussage sogar mit einer Diagonalmatrix.

Lemma 4 *Genau dann wenn A normal ist, existiert eine unitäre Ähnlichkeitstransformation, so dass*

$$U^H A U = \operatorname{diag}\{\lambda_1, \dots, \lambda_n\}.$$

Beweis: Siehe [7, Satz 1.4.2.].

2.4 Nullstellenverfahren

Um die Nullstellen von Funktionen (wie zum Beispiel vom charakteristischen Polynom) zu bestimmen, gibt es Verfahren wie das Newton-Verfahren, die Bisektion (Intervallhalbierung), das Sekantenverfahren oder das Regula-Falsi-Verfahren (eine Mischung aus Bisektion und Sekantenverfahren).

In dieser Arbeit sind davon jedoch nur das Newton-Verfahren und die Bisektion relevant und werden im Folgenden kurz vorgestellt.

2.4.1 Bisektion

Die Bisektion ist ein einfaches Verfahren zur Bestimmung einer Nullstelle einer reellwertigen Funktion auf einem Intervall mittels Intervallschachtelungen und Vorzeichenbetrachtungen.

Voraussetzung für diese Methode ist die Stetigkeit der Funktion auf dem betrachteten Intervall. Falls dann genau eine Nullstelle in dem Intervall existiert, ist die Konvergenz der Bisektion sichergestellt.

Algorithmus:

Bisektion für eine Funktion f auf einem Intervall $[l, r]$ mit $l < r$:

- Berechne den Mittelpunkt des Intervalls $t := \frac{l+r}{2}$.
- Falls $r - l$ klein genug ist, ist das Lösungsintervall gefunden und t kann als Lösungswert zurückgegeben werden, sonst:
  - Falls $f(l)f(t) < 0$ ist, wende die Bisektion auf $[l, t]$ an,
  - sonst auf $[t, r]$.

Je Iterationsschritt muss für die Bisektion nur eine neue Funktionsauswertung berechnet werden.

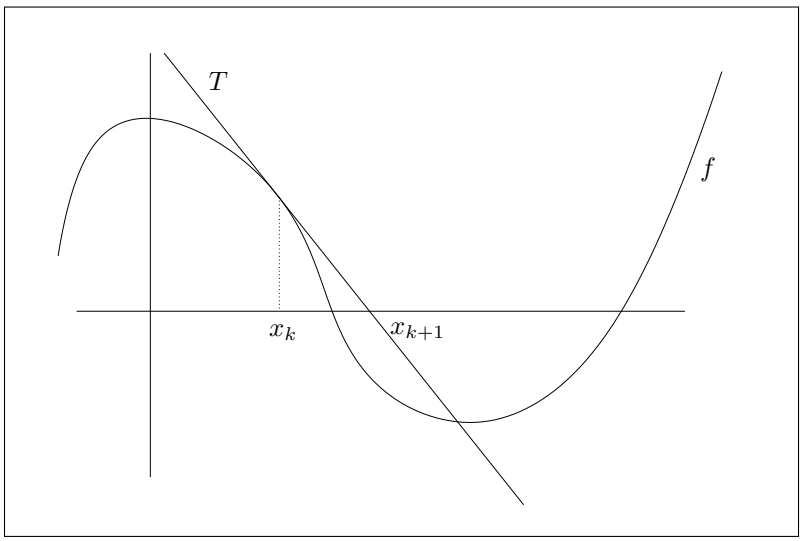
Sei mit ε die gewünschte Endtoleranz bezeichnet. Da das Anfangsintervall $[l, r]$ in jedem Schritt halbiert wird, ergibt sich für die benötigten Iterationsschritte n folgende Abschätzung:

$$\frac{l-r}{2^n} \leq \varepsilon \qquad \Longleftrightarrow \qquad \log_2 \frac{l-r}{\varepsilon} \leq n\,.$$

2.4.2 Newton-Verfahren

Die Idee hinter dem Newton-Verfahren ist, die (reelle, nichtlineare) Funktion f in einem Punkt x_k lokal durch eine Tangente T zu approximieren (zu linearisieren), so dass die Nullstelle der Tangente x_{k+1} die neue Näherung für die gesuchte Nullstelle der Funktion ist (siehe Abbildung).

Die Funktion muss dazu stetig differenzierbar sein, da nicht nur Funktionswerte, sondern auch deren erste Ableitung gebraucht werden.



Algorithmus:

Newton-Verfahren für eine Funktion $f : \mathbb{R} \to \mathbb{R}$:

(1) Wähle einen Startwert x_0.

(2) Berechne

$$x_{k+1} \;=\; x_k - \frac{f(x_k)}{f'(x_k)} \,. \tag{2.3}$$

(3) Falls $|x_{k+1} - x_k| > \varepsilon$ und $f(x_k) > \delta$ für ein $\varepsilon > 0$ und ein $\delta > 0$: weiter mit (2).
Sonst: Abbruch.

Je Iterationsschritt ist hierbei eine Funktionsauswertung und eine Ableitungsauswertung zu berechnen.

Wenn der Startwert nahe genug an der gesuchten Nullstelle liegt, konvergiert das Newton-Verfahren quadratisch.

Satz 2 *Sei $f : \mathbb{R} \to \mathbb{R}$ eine hinreichend oft stetig differenzierbare Funktion und $x_* \in \mathbb{R}$ eine Nullstelle von f.*

Falls $f'(x_*) \neq 0$ ist, konvergiert das Newton-Verfahren (2.3) für Startwerte hinreichend nahe der gesuchten Nullstelle quadratisch.

Beweis: Siehe [10, Theorem 5.6.(a)].

Kapitel 3

$\mathcal{H}$-Matrizen

Das Format der hierarchischen Matrizen beruht auf der Idee, sehr große Matrizen, welche zum Beispiel aus der Diskretisierung partieller Differentialgleichungen oder Integralgleichungen resultieren, speichergünstig aber möglichst gut zu approximieren und damit Rechenoperationen mit akzeptablem Aufwand ausführen zu können.

Dazu werden bestimmte Blöcke einer Matrix durch Niedrigrang-Matrizen approximiert. Eine speicherarme Darstellung dieser Niedrigrang-Matrizen ist das Rang-k-Matrix-Format, das in Kapitel 3.1 kurz vorgestellt wird. In Kapitel 3.2 wird dann darauf aufbauend ein einfaches Modell für hierarchische Matrizen eingeführt, bei dem das Matrixformat fest vorgegeben ist.

Eine ausführliche Einführung in das Thema der hierarchischen Matrizen ist nachzulesen in [8] und [1].

3.1 Rang-k-Matrizen

Rang-k-Matrizen sind ein elementarer Bestandteil hierarchischer Matrizen, da die Rechenoperationen von $\mathcal{H}$-Matrizen auf Additionen und Multiplikationen von Rang-k-Matrizen zurückgeführt werden können.

Für dieses Rang-k-Matrix-Format kann eine Arithmetik mit fast linearem Aufwand definiert werden.

Definition 9 (Rang-k-Darstellung) *Seien Indexmengen $I = \{1, \ldots, n\}$ und $J = \{1, \ldots, m\}$ gegeben. Eine Matrix $M \in \mathbb{R}^{I \times J}$ mit* $\mathrm{Rang}(M) = k$*, wobei $k \leq \min\{n, m\}$ ist, lässt sich in der* Rang-k-Darstellung *schreiben als:*

$$M \;=\; A\,B^T \qquad \text{mit } A \in \mathbb{R}^{I \times \{1,\ldots,k\}} \text{ und } B \in \mathbb{R}^{J \times \{1,\ldots,k\}}. \tag{3.1}$$

Die Menge aller Matrizen in der Darstellung (3.1) wird mit $\mathcal{R}(k, I, J)$ oder auch kurz mit $\mathcal{R}(k)$ bezeichnet.

Jede Matrix in der Rang-k-Matrix-Darstellung hat höchstens Rang k und jede Matrix mit Rang k besitzt eine Rang-k-Matrix-Darstellung, genauer:

Bemerkung 2 *Es gilt:*

1. *Falls $M \in \mathcal{R}(k, I, J)$, so ist Rang$(M) \leq k$.*
2. *Wenn Rang$(M) = k$, dann existiert eine Darstellung (3.1), so dass $M \in \mathcal{R}(k, I, J)$.*

Beweis: Siehe [8, Bemerkung 2.2.1 und 2.2.2.]

Wenn man den Speicherbedarf einer Matrix in klassischer Darstellung $M = (a_{ij})_{i,j}$ mit $i = 1, \ldots, n$ und $j = 1, \ldots, m$ von $\mathcal{O}(nm)$ betrachtet, wird die Darstellung im Rang-k-Matrix-Format dann sinnvoll, wenn $k \ll \min\{n, m\}$ ist, da sich der Speicheraufwand in dem Fall deutlich reduziert.

Lemma 5 (Speicheraufwand von Rang-k-Matrizen) *Der Speicherbedarf für eine Matrix $M \in \mathcal{R}(k, I, J)$ beträgt $\mathcal{O}(k(n + m))$.*

Beweis: Siehe [8, Bemerkung 2.2.5.].

3.2 Einfaches Modell für $\mathcal{H}$-Matrizen

Die Definition des einfachen Modellformats für hierarchische Matrizen führt nur zu quadratischen Matrizen mit Zweierpotenz-Dimensionen und auf eine fest vorgegebene Block-Unterteilung.

Für den Algorithmus, der in Kapitel 4 vorgestellt wird, genügt die Betrachtung dieses Modellformates. Eine Idee zur Erweiterung des Algorithmus auf eine allgemeinere $\mathcal{H}$-Matrix-Struktur ist in Abschnitt 6.2.3 zu finden.

Definition 10 ($\mathcal{H}_l$-Matrizen) *Sei $I = \{1, \ldots, n\}$ eine Indexmenge und $n = 2^l$ mit $l \in \mathbb{N}$. Sei $M \in \mathbb{R}^{I \times I}$ und $k \in \mathbb{N}$ mit $k < n$. Die Menge der $\mathcal{H}_l$-Matrizen (mit lokalem Rang k) sei induktiv wie folgt definiert:*

$M \in \mathcal{H}_l(k)$, falls

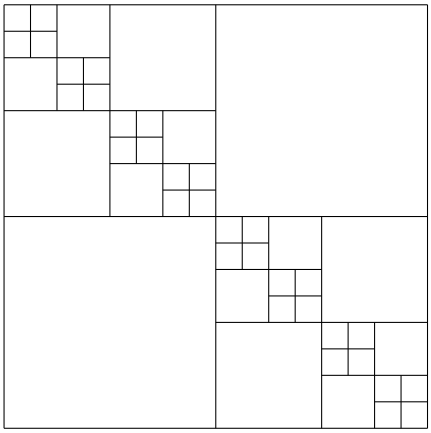
- *$n = 1$ (also $l = 0$) oder*
- *die Partitionierung in 2 mal 2 Blöcke der Größe $\frac{n}{2} \times \frac{n}{2}$ ergibt*

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$$

wobei $M_{11}, M_{22} \in \mathcal{H}_{l-1}(k)$ und $M_{12}, M_{21} \in \mathcal{R}(k)$.

Bemerkung 3 *Allgemeiner kann man $\mathcal{H}_0$ aus Definition 10 als Klasse aller $q \times q$-Matrizen für $1 \leq q \leq n_{min}$ erklären.*

Die Matrixstruktur für $\mathcal{H}_4$ sieht mit Definition 10 wie folgt aus:



Ein Beispiel für Matrizen aus $\mathcal{H}_l(k)$ sind Bandmatrizen. Dabei ist dann k das Maximum der Anzahl der Nebendiagonalen auf beiden Seiten der Hauptdiagonalen. Im Falle einer Bandmatrix aus $\mathcal{H}_l(1)$ handelt es sich um eine Tridigonalmatrix.

Im Folgenden wird das Modellformat $\mathcal{H}_l(1)$ betrachtet, so wie es später in Kapitel 4.2 für den Eigenwertlöser benutzt wird.

Der Speicheraufwand dieser Matrizen ist auf den Speicherbedarf von Rang-k-Matrizen zurückzuführen.

Lemma 6 *Sei $n = 2^l$ und $k = 1$. Der Speicherbedarf einer Matrix aus $\mathcal{H}_l(1)$ beträgt $n + 2n \log_2 n$.*

Beweis: Siehe [8, Lemma 3.3.1.].

Kapitel 4

Eigenwertlöser für $\mathcal{H}$-Matrizen mit lokalem Rang 1

In diesem Kapitel wird zunächst als Motivation in Kapitel 4.1 ein Algorithmus für das Eigenwertproblem von Tridiagonalmatrizen behandelt, auf dessen Grundideen das Verfahren für das Modellformat von $\mathcal{H}$-Matrizen mit lokalem Rang 1 aufgebaut ist, das dann in Kapitel 4.2 vorgestellt wird.

4.1 Motivation: Algorithmus für Tridiagonalmatrizen

Klassische Eigenwertlöser für symmetrische Matrizen gehen in der Regel so vor, dass die Matrix zunächst mittels Householder-Transformationen auf Tridiagonalgestalt gebracht wird, um dann anschließend mit einem QR-Algorithmus die Eigenwerte und Eigenvektoren der Tridiagonalmatrix zu bestimmen.

Zur Lösung des Eigenwertproblems der Tridiagonalmatrix schlug Cuppen 1981 einen Divide-and-Conquer-Algorithmus vor. Ausgehend von diesem Algorithmus entwickelten Dongorra und Sorensen 1987 ein vollständig parallelisiertes Verfahren für dieses Problem.

Dongarra und Sorensen stellten auch Ideen vor, wie man die Reduktion auf Tridiagonalgestalt mit dem von ihnen entwickelten Algorithmus verbinden könnte, um dann einen vollständig parallelen Algorithmus zur Berechnung eines symmetrischen Eigenwertproblems zu erhalten. Näheres dazu ist zu finden in [4].

Der hier vorgestellte Algorithmus ist eine einfache Variante basierend auf dem ursprünglichen Divide-and-Conquer-Algorithmus von Cuppen [2].

4.1.1 Divide and Conquer

Die allgemeine Idee hinter einem Divide-and-Conquer-Algorithmus besteht darin, das Problem in kleinere Teilprobleme zu zerlegen und diese unabhängig voneinander zu lösen. Anschließend werden die Teillösungen zur Lösung des Gesamtproblems zusammengefügt.

Divide

Bei einer Block-Diagonalmatrix der Form

$$T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix}$$

zerfällt das Eigenwertproblem für T vollkommen in zwei Teilprobleme: Die Eigenwerte der Teilmatrizen T_1 und T_2 zusammengenommen sind bereits die Eigenwerte der Gesamtmatrix T.

Eine Tridiagonalmatrix ist *fast von Block-Diagonalgestalt*; und zwar bis auf die zwei Einträge auf den Nebendiagonalen *in der Mitte*.

Um auch für eine Tridiagonalmatrix diese Teilung vornehmen zu können, wird diese als Block-Diagonalmatrix mit einer Rang-1-Störung geschrieben.

Sei $T = (t_{ij})_{i,j=1,\dots,n} \in \mathbb{R}^{n\times n}$ eine symmetrische strikte Tridiagonalmatrix. (Die Eigenschaft, dass die Nebendiagonaleinträge nicht Null sind, also die Tridiagonalmatrix strikt ist, wird gefordert, da das Problem sonst wie im einfachen Fall oben zerfallen würde.) Dann gilt

$$\mathrm{T} = \left[\begin{array}{cc|cc} & & & \\ & \mathrm{T}_1 & & 0 \\ & & \beta & \\ \hline & \beta & & \\ & 0 & & \mathrm{T}_2 \\ & & & \end{array}\right] = \begin{bmatrix} \mathrm{T}_1 & \beta e_m e_1^T \\ \\ \beta e_1 e_m^T & \mathrm{T}_2 \end{bmatrix}$$

wobei $T_1 \in \mathbb{R}^{m\times m}$, $T_2 \in \mathbb{R}^{(n-m)\times(n-m)}$, $\beta \in \mathbb{R}$, $e_1 \in \mathbb{R}^{n-m}$ erster Einheitsvektor und $e_m \in \mathbb{R}^m$ m-ter Einheitsvektor.

Aus Effizienzgründen sollte hier $m \approx \frac{n}{2}$ gewählt werden.

Die Matrix lässt sich als Summe einer Blockdiagonalmatrix und einer Rang-1-Matrix schreiben:

$$\mathrm{T} = \left[\begin{array}{cc|cc} & & & \\ & \hat{\mathrm{T}}_1 & & 0 \\ & & & \\ \hline & & & \\ & 0 & & \hat{\mathrm{T}}_2 \\ & & & \end{array}\right] + \left[\begin{array}{cc|cc} & & & \\ & 0 & & 0 \\ & & \beta & \beta \\ \hline & & \beta & \beta \\ & 0 & & 0 \\ & & & \end{array}\right]$$

oder genauer

$$\begin{aligned} T &= \begin{bmatrix} \hat{T}_1 & 0 \\ 0 & \hat{T}_2 \end{bmatrix} + \beta \begin{bmatrix} e_m e_m^T & e_m e_1^T \\ e_1 e_m^T & e_1 e_1^T \end{bmatrix} \\ &= \begin{bmatrix} \hat{T}_1 & 0 \\ 0 & \hat{T}_2 \end{bmatrix} + \beta \begin{bmatrix} e_m \\ e_1 \end{bmatrix} \begin{bmatrix} e_m^T & e_1^T \end{bmatrix} \end{aligned} \tag{4.1}$$

mit

$$\hat{T}_1 = \begin{bmatrix} t_{1,1} & \dots & t_{1,m} \\ \vdots & & \vdots \\ t_{m,1} & \dots & t_{m,m} - \beta \end{bmatrix}, \quad \hat{T}_2 = \begin{bmatrix} t_{m+1,m+1} - \beta & \dots & t_{m+1,n} \\ \vdots & & \vdots \\ t_{n,m+1} & \dots & t_{n,n} \end{bmatrix}.$$

Seien $\hat{T}_1 = U_1 D_1 U_1^T$ und $\hat{T}_2 = U_2 D_2 U_2^T$ die Diagonalisierungen der Matrizen $\hat{T}_1$ und $\hat{T}_2$ im Sinne von Lemma 4, wobei $U_1 \in \mathbb{R}^{m \times m}$, $U_2 \in \mathbb{R}^{(n-m)\times(n-m)}$ unitär und die Diagonalmatrizen $D_1 = \text{diag}\,(\lambda_1^{(\hat{T}_1)}, \dots, \lambda_m^{(\hat{T}_1)})$, $D_2 = \text{diag}\,(\lambda_1^{(\hat{T}_2)}, \dots, \lambda_{n-m}^{(\hat{T}_2)})$ die Eigenwerte von $\hat{T}_1$ und $\hat{T}_2$ enthalten.

Diese können durch rekursive Aufrufe des Algorithmus berechnet werden, wobei in der Praxis oft bei genügend kleinen Teilproblemen der QR-Algorithmus benutzt wird.

Conquer

Aus den Diagonalisierungen der Teilmatrizen $\hat{T}_1$ und $\hat{T}_2$ soll nun die Diagonalisierung von T berechnet werden.

$\hat{T}_1 = U_1 D_1 U_1^T$ und $\hat{T}_2 = U_2 D_2 U_2^T$ in (4.1) eingesetzt, ergibt

$$T = \begin{bmatrix} U_1 D_1 U_1^T & 0 \\ 0 & U_2 D_2 U_2^T \end{bmatrix} + \beta \begin{bmatrix} e_m \\ e_1 \end{bmatrix} \begin{bmatrix} e_m^T & e_1^T \end{bmatrix}$$

und mit

$$U := \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} \in \mathbb{R}^{n \times n}, \quad D := \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \in \mathbb{R}^{n \times n}$$

folgt

$$T = U \left(D + \beta \begin{bmatrix} U_1^T e_m \\ U_2^T e_1 \end{bmatrix} \begin{bmatrix} e_m^T U_1 & e_1^T U_2 \end{bmatrix} \right) U^T.$$

Setze

$$z := \sqrt{\beta} \begin{bmatrix} U_1^T e_m \\ U_2^T e_1 \end{bmatrix}$$

und es folgt schließlich

$$T = U(D + zz^T)U^T. \tag{4.2}$$

Damit sind nun die Eigenwerte der Matrix $D + zz^T$ gesucht.

Falls z dabei Nulleinträge enthält, können entsprechend viele Eigenwerte an $D + zz^T$ direkt abgelesen werden.

Lemma 7 *Sei $D = diag\,(d_1, \ldots, d_n) \in \mathbb{R}^{n\times n}$ und $z \in \mathbb{R}^n$. Falls $z_i = 0$ für ein $i \in \{1, \ldots, n\}$, ist d_i ein Eigenwert von $D + zz^T$.*

Beweis: Sei $z_i = 0$ für ein $i \in \{1, \ldots, n\}$. Dann sind die i-te Zeile und die i-te Spalte von zz^T gleich 0, also gilt

$$D + zz^T = \left[\begin{array}{c|c|c} * & 0 & * \\ \hline 0 & d_i & 0 \\ \hline * & 0 & * \end{array}\right]$$

und damit folgt sofort, dass d_i ein Eigenwert von $D + zz^T$ ist. □

Im Folgenden wird daher angenommen, dass in z keine Nulleinträge existieren.

Falls nun die Diagonalmatrix D mehrfache Einträge hat, sind ebenso direkt Eigenwerte abzulesen.

Lemma 8 *Sei $D = diag\,(d_1, \ldots, d_n) \in \mathbb{R}^{n\times n}$ mit $d_1 \leq \cdots \leq d_n$ und $z \in \mathbb{R}^n$ mit $z_i \neq 0$ für alle $i \in \{1, \ldots, n\}$. Falls $d_k = d_{k+1}$ für ein $k \in \{1, \ldots, n-1\}$, dann ist d_k ein Eigenwert von $D + zz^T$.*

Beweis: Gelte $d_k = d_{k+1}$ für ein $k \in \{1, \ldots, n-1\}$. Sei $v \in \mathbb{R}^n$ mit

$$v^T = (0, \ldots, 0, \underbrace{z_{k+1}}_{\substack{k\text{-te}\\ \text{Stelle}}}, \underbrace{-z_k}_{\substack{(k+1)\text{-te}\\ \text{Stelle}}}, 0, \ldots, 0)^T.$$

Da nach Voraussetzung $z_k, z_{k+1} \neq 0$, ist $v \neq 0$ und es gilt weiter

$$((D + zz^T) - d_k I)v = \begin{pmatrix} z_1 z_k \cdot z_{k+1} - z_1 z_{k+1} \cdot z_k \\ \vdots \\ z_{k-1} z_k \cdot z_{k+1} - z_{k-1} z_{k+1} \cdot z_k \\ (d_k - d_k + z_k^2) z_{k+1} - z_k z_{k+1} \cdot z_k \\ z_{k+1} z_k \cdot z_{k+1} - (d_{k+1} - d_k + z_{k+1}^2) z_k \\ z_{k+2} z_k \cdot z_{k+1} - z_{k+2} z_{k+1} \cdot z_k \\ \vdots \\ z_n z_k \cdot z_{k+1} - z_n z_{k+1} \cdot z_k \end{pmatrix} = 0\,.$$

Also ist $v \in \text{Kern}((D+zz^T)-d_kI)$ und $v \neq 0$ und damit ist $(D+zz^T)-d_kI$ nicht injektiv, also nicht regulär und daraus folgt, dass d_k ein Eigenwert von $D+zz^T$ ist. □

Auch dieser Fall wird im Folgenden ausgeschlossen, indem verlangt wird, dass die Diagonaleinträge von D streng monoton steigend sortiert sind.

Für weitere Umformungen ist das folgende Lemma hilfreich.

Lemma 9 *Sei $D = diag\,(d_1, \ldots, d_n) \in \mathbb{R}^{n\times n}$ mit $d_1 < \cdots < d_n$ und $z \in \mathbb{R}^n$ mit $z_i \neq 0$ für alle $i \in \{1, \ldots, n\}$ und sei (λ, v) ein Eigenpaar der Matrix $D + zz^T$. Dann ist*

$$z^T v \neq 0 \quad \textit{und} \quad D - \lambda I \quad \textit{regulär.}$$

Beweis: Da (λ, v) ein Eigenpaar von $D + zz^T$ ist, gilt

$$(D + zz^T)v = \lambda v \quad \Leftrightarrow \quad (D - \lambda I)v = -z(z^T v) \tag{4.3}$$

Annahme: λ ist auch ein Eigenwert von D.

Dann ist $\lambda = d_i$ für ein $i \in \{1, \ldots, n\}$ und mit (4.3) gilt

$$0 = (d_i - \lambda)v_i = -(z^T v)z_i$$

Da nach Voraussetzung $z_i \neq 0$ ist, folgt daraus $z^T v = 0$ und mit (4.3) dann

$$(D - \lambda I)v = 0 \quad \Leftrightarrow \quad Dv = \lambda v$$

Damit ist v auch ein Eigenvektor von D und D ist eine Diagonalmatrix mit nur einfachen Eigenwerten, also folgt daraus $v \in \text{span}\,\{e_i\}$, und insbesondere $v_i \neq 0$. Damit ist aber $0 = z^T v = z_i v_i$ im Widerspruch zu $z_i \neq 0$.

Also ist $z^T v \neq 0$; D und $D + zz^T$ haben keinen gemeinsamen Eigenwert und demnach ist $D - \lambda I$ regulär. □

Seien nun D, z, λ und v wie in Lemma 9. Dann gilt

$$\begin{aligned} (D + zz^T)v = \lambda v \quad &\Longleftrightarrow \quad Dv + z(z^T v) = \lambda v \\ &\Longleftrightarrow \quad (D - \lambda I)v + z(z^T v) = 0 \end{aligned}$$

In Lemma 9 wurde gezeigt, dass $D-\lambda I$ regulär ist, also existiert $(D-\lambda I)^{-1}$ und es folgt weiter

$$\begin{aligned} & (D - \lambda I)^{-1}(D - \lambda I)v + (D - \lambda I)^{-1} z(z^T v) = 0 \\ \Longleftrightarrow \quad & v + (D - \lambda I)^{-1} z(z^T v) = 0 \\ \Longleftrightarrow \quad & (z^T v) + z^T (D - \lambda I)^{-1} z(z^T v) = 0 \end{aligned}$$

und da $z^T v \neq 0$ nach Lemma 9, kann man beide Seiten durch $z^T v$ teilen und erhält

$$1 + z^T (D - \lambda I)^{-1} z = 0 \quad \Longleftrightarrow \quad 1 + (z\ ,\ (D - \lambda I)^{-1} z) = 0\,,$$

wobei $(\,\cdot\,,\,\cdot\,)$ das Euklidische Skalarprodukt ist, oder komponentenweise

$$1 + \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda} \ = \ 0. \tag{4.4}$$

Bemerkung 4 *Lemma 9 gilt auch für $D - zz^T$ und dazu lässt sich die folgende Gleichung herleiten:*

$$1 \ - \sum_{i=0}^{n} \frac{z_i^2}{d_i - \lambda} \ = \ 0$$

Beweis: Lemma 9 kann mit $D - zz^T$ genauso bewiesen werden. Die Gleichung kann dann analog zur Herleitung von (4.4) aufgestellt werden. □

Hiermit wird das Eigenwertproblem auf das Lösen einer rationalen Gleichung reduziert. Die Nullstellen der Funktion

$$f(x) \ := \ 1 + \sum_{i=1}^{n} \frac{z_i^2}{d_i - x}$$

sind die Eigenwerte der Matrix T aus (4.2) und $f(x) = 0$ heißt **Säkulärgleichung**.

Um die Eigenwerte zu berechnen, genügt es, mit Hilfe eines Verfahrens ähnlich dem Newton-Verfahren die Nullstellen von f zu bestimmen. Unter anderem wird im folgenden Unterkapitel 4.1.2 ein solches Verfahren beschrieben.

Diese Vorgehensweise ist auch dann möglich, wenn z Nulleinträge haben sollte. Dazu wird dann folgendes Lemma benötigt.

Lemma 10 *Sei $D = diag(d_1, \ldots, d_n)$ mit $d_1 < \cdots < d_n$ und $z \in \mathbb{R}^n$. Dann existiert eine Permutationsmatrix P so, dass*

$$P\,D\,P^T \ = \ diag(\widetilde{d}_1, \ldots, \widetilde{d}_n) \quad \ mit \quad \ \widetilde{d}_1 < \widetilde{d}_2 < \cdots < \widetilde{d}_k$$

und für $w := Pz$ gilt dann $w_i \neq 0$ für $i \in \{1, \ldots, k\}$ und $w_i = 0$ für $i \in \{k+1, \ldots, n\}$.

Beweis: Die gesuchte Permutationsmatrix P ist wie folgt konstruierbar: Sei $z_i = 0$ und $z_{i+1} \neq 0$ für ein $i \in \{1, \ldots, n\}$. Sei P_1 eine Einheitsmatrix, in welcher die i-te und die $(i+1)$-te Zeile vertauscht sind. Dann ist $(P_1 z)_i \neq 0$ und $(P_1 z)_{i+1} = 0$ und es gilt

$$P_1 D P_1^T \;=\; \operatorname{diag}(d_1, \ldots, d_{i-1}, d_{i+1}, d_i, d_{i+2}, \ldots, d_n)\,.$$

Durch wiederholtes Anwenden von Matrizen P_j werden die Nulleinträge von z nach unten permutiert. Da es sich hierbei jeweils nur um Transpositionen handelt, werden die zu den $z_i = 0$ gehörenden d_i ebenso nach unten geschoben ohne die Reihenfolge der übrigen Diagonaleinträge zu verändern.

Die Permutationsmatrix P ist das Produkt aller m angewandten Transpositionen:

$$P_m \ldots P_2 P_1 \;=\; P\,.$$

□

Eine etwas allgemeinere Version dieses Lemmas, die auch mehrfache Einträge in der Diagonalmatrix D berücksichtigt, ist zu finden in [6, Theorem 8.5.4].

Nach Lemma 7 sind die d_i zu $z_i = 0$ bereits Eigenwerte von $D + zz^T$ und nach Lemma 10 kann nach den anderen Eigenwerten gesucht werden, indem die Säkulärgleichung für die durch geeignete Permutationen entstandene obere $k \times k$-Teilmatrix wie zuvor aufgestellt wird.

4.1.2 Die Säkulärgleichung: Eigenschaften und Lösung

Im ersten Abschnitt dieses Unterkapitels wird zunächst die Säkulärgleichung näher betrachtet, um dann im zweiten Abschnitt ein mögliches Verfahren zur Lösung dieser Gleichung vorzustellen.

Spezielle Eigenschaft der Säkulärgleichung

Die Funktion f aus der Säkulärgleichung $f(x) = 0$ hat die Eigenschaft, dass zwischen je zwei ihrer Polstellen genau eine ihrer Nullstellen liegt, so dass die Suche nach einer Nullstelle immer auf ein bestimmtes Intervall beschränkt werden kann.

Satz 3 *Seien $d_i \in \mathbb{R}$ für $i \in \{1, \ldots, n\}$ mit $d_1 < \cdots < d_n$ und $z \in \mathbb{R}^n$, wobei $z_i \neq 0$ für alle $i \in \{1, \ldots, n\}$. Die Funktion*

$$f(x) \;=\; 1 + \sum_{i=1}^{n} \frac{z_i^2}{d_i - x} \tag{4.5}$$

hat genau n (reelle) Nullstellen $\lambda_1 < \cdots < \lambda_n$ und diese trennen die d_i. Genauer gilt

$$d_1 < \lambda_1 < d_2 < \cdots < \lambda_{n-1} < d_n < \lambda_n < d_n + z^T z\,.$$

Beweis: f ist eine rationale Funktion vom Zählergrad n und hat daher höchstens n (reelle) Nullstellen.

Sei $j \in \{1, \ldots, n\}$ und $\varepsilon \in \mathbb{R}$. Dann gilt

$$f(d_j + \varepsilon) \;=\; 1 + \sum_{i=1}^{n} \frac{z_i^2}{d_i - d_j - \varepsilon} \quad \xrightarrow[\varepsilon \to 0]{} \begin{cases} +\infty & \text{für } \varepsilon < 0 \\ -\infty & \text{für } \varepsilon > 0 \end{cases} \tag{4.6}$$

da der Nenner des j-ten Summanden $d_j - d_j - \varepsilon = -\varepsilon$ ist. Also gibt es an jeder Polstelle einen Vorzeichenwechsel. Weiter kann man aus der Ableitung

$$f'(x) = \sum_{i=1}^{n} \frac{z_i^2}{(d_i - x)^2}$$

ablesen, dass f außerhalb der Polstellen monoton steigend verläuft. Mit dem Zwischenwertsatz folgt dann insgesamt, dass zwischen je zwei Polstellen von f eine Nullstelle zu finden ist, also eine in jedem Intervall (d_j, d_{j+1}) mit $j = 1, \ldots, n-1$.

Es ist $f(x) > 0$ für alle $x < d_1$ und ferner gilt

$$f(x) \;=\; 1 + \sum_{i=1}^{n} \frac{z_i^2}{d_i - x} \quad \longrightarrow \quad 1 \quad \text{für } x \to \pm\infty \,.$$

Also gibt es keine Nullstelle, die kleiner als d_1 ist.

Nach (4.6) gilt $f(d_n + \varepsilon) \to -\infty$ für $\varepsilon > 0$. Und wegen $d_n - d_i > 0$ für $i < n$ gilt weiter

$$\begin{aligned} f(d_n + z^T z) &= 1 + \sum_{i=1}^{n} \frac{z_i^2}{d_i - d_n - z^T z} \\ &= 1 - \sum_{i=1}^{n} \frac{z_i^2}{z^T z + d_n - d_i} \\ &> 1 - \sum_{i=1}^{n} \frac{z_i^2}{z^T z} \\ &= 0 \end{aligned}$$

Damit folgt auch die letzte Einschließung

$$d_n < \lambda_n < d_n + z^T z \,.$$

□

Bemerkung 5 *Ersetzt man in (4.5) das + durch ein −, also*

$$\widehat{f}(x) \;=\; 1 - \sum_{i=0}^{n} \frac{z_i^2}{d_i - x}$$

erhält man eine Anordnung der Nullstellen λ_i von $\widehat{f}$ der Form

$$d_1 - zz^T < \lambda_1 < d_1 < \lambda_2 < \cdots < d_{n-1} < \lambda_n < d_n \ .$$

Beweis: Der Beweis geht analog zum Beweis von Satz 3.

Das Verhalten der Nullstellen und Polstellen der Funktion f aus (4.5) ist an dem Graphen in Abbildung 4.1 gut zu sehen.

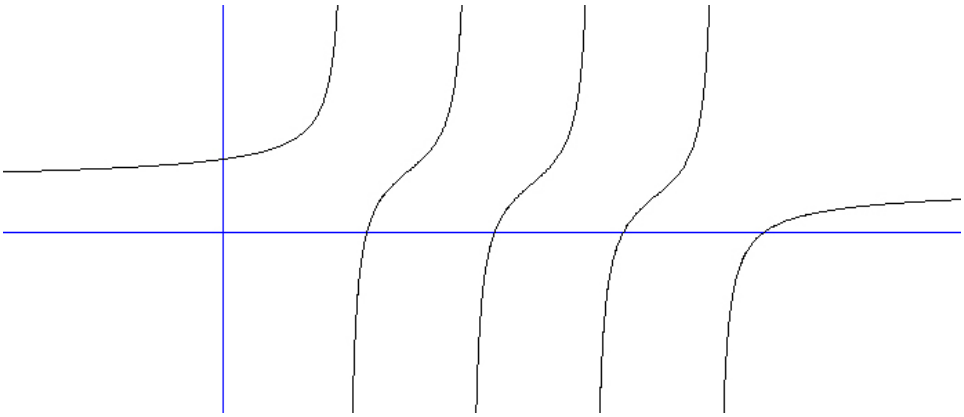


Abbildung 4.1: Die Funktion f aus (4.5)

Lösen der Säkulärgleichung mit Hebden-ähnlichen Verfahren

Um die (nach Satz 3 jeweils eindeutige) Nullstelle im Intervall (d_j, d_{j+1}) für $j \in \{1, \ldots, n-1\}$ zu bestimmen, wäre es naheliegend, auf das Newton-Verfahren zurückzugreifen. Allerdings ergibt sich dabei die Schwierigkeit, dass die Folge der Newton-Iterierten nicht unbedingt in dem Intervall zwischen den Polstellen bleibt, in dem der Startwert gewählt wurde.

Dieses Verhalten ist darin begründet, dass die Funktion f nahe ihrer Nullstellen nur sehr schlecht durch eine Gerade approximiert werden kann. Daher wird in dem folgenden Iterationsverfahren die Funktion, ähnlich wie bei dem Hebden-Verfahren, durch einfache rationale Funktionen approximiert werden, die zu dem Kurvenverlauf von f *besser passen* (siehe dazu auch [3, Kapitel 5.5.3]).

Dazu wird f aus (4.5) zunächst wie folgt zerlegt: Sei x aus dem aktuellen Intervall (d_j, d_{j+1}), in dem die Nullstelle gesucht werden soll. Definiere

$$f(x) \ = \ 1 + f_-(x) + f_+(x)$$

mit

$$f_-(x) \ := \ \sum_{i=1}^{j} \frac{z_i^2}{d_i - x} \quad \text{und} \quad f_+(x) \ := \ \sum_{i=j+1}^{n} \frac{z_i^2}{d_i - x}$$

wobei $f_-(x)$ dann die Summe aller negativen Terme und $f_+(x)$ die Summe aller positiven Terme von f ist. Auf diese Weise ist eine Auslöschung bei der Auswertung von f_- und f_+ ausgeschlossen.

Nun sollen die Teilfunktionen f_- und f_+ durch rationale Funktionen mit denselben Polstellen in $[d_j, d_{j+1}]$ approximiert werden. Diese seien mit h_- und h_+ bezeichnet und wie folgt definiert:

$$\begin{aligned} h_-(x) &:= \gamma_- + \frac{\alpha_-}{d_j - x} && \approx f_-(x), \\ h_+(x) &:= \gamma_+ + \frac{\alpha_+}{d_{j+1} - x} && \approx f_+(x) \end{aligned} \tag{4.7}$$

mit $\alpha_-, \alpha_+, \gamma_-, \gamma_+ \in \mathbb{R}$.

Die Approximationen h_- und h_+ sollen die Teilfunktionen f_- und f_+ in der aktuellen Iterierten x_k berühren, also denselben Funktionswert

$$h_-(x_k) = f_-(x_k) \quad \text{und} \quad h_+(x_k) = f_+(x_k)$$

und dieselbe Steigung

$$h'_-(x_k) = f'_-(x_k) \quad \text{und} \quad h'_+(x_k) = f'_+(x_k)$$

haben.

Daraus lassen sich nun α_-, α_+ und γ_-, γ_+ aus (4.7) berechnen: Aus den Ableitungen erhält man zunächst α_- und α_+

$$\begin{aligned} h'_-(x_k) &= \frac{\alpha_-}{(d_j - x_k)^2} \stackrel{!}{=} f'_-(x_k) \\ &\qquad \Longrightarrow \quad \alpha_- = f'_-(x_k)(d_j - x_k)^2 \\ h'_+(x_k) &= \frac{\alpha_+}{(d_{j+1} - x_k)^2} \stackrel{!}{=} f'_+(x_k) \\ &\qquad \Longrightarrow \quad \alpha_+ = f'_+(x_k)(d_{j+1} - x_k)^2 \end{aligned}$$

und durch Einsetzen der berechneten α_- und α_+ erhält man γ_- und γ_+

$$\begin{aligned} h_-(x_k) &= \gamma_- + \frac{\alpha_-}{d_j - x_k} \stackrel{!}{=} f_-(x_k) \\ &\Longrightarrow \quad \gamma_- = f_-(x_k) - \frac{\alpha_-}{d_j - x_k} = f_-(x_k) - f'_-(x_k)(d_j - x_k) \\ h_+(x_k) &= \gamma_+ + \frac{\alpha_+}{d_{j+1} - x_k} \stackrel{!}{=} f_+(x_k) \\ &\Longrightarrow \quad \gamma_+ = f_+(x_k) - \frac{\alpha_+}{d_{j+1} - x_k} = f_+(x_k) - f'_+(x_k)(d_{j+1} - x_k) \end{aligned}$$

Als Approximation von f wird schließlich h definiert:

$$h(x) := 1 + h_-(x) + h_+(x)$$

Damit erhält man mit

$$h(x_k) \;=\; 0$$

eine quadratische Gleichung, die im aktuellen Intervall (d_j, d_{j+1}) genau eine Lösung hat, welche als neue Iterierte x_{k+1} gewählt wird.

4.2 Algorithmus für $\mathcal{H}_l(1)$-Matrizen

Der Eigenwertlöser für das in Kapitel 3.2 vorgestellte einfache Modellformat der $\mathcal{H}$-Matrizen mit lokalem Rang 1 ist nach dem Vorbild des in Kapitel 4.1 vorgestellten Algorithmus für Tridiagonalmatrizen entstanden. Insbesondere sind Tridiagonalmatrizen ja auch als Matrizen aus $\mathcal{H}_l(1)$ darstellbar.

Der Divide-and-Conquer-Teil ist dem für Tridiagonalmatrizen sehr ähnlich. Auch hier wird das Eigenwertproblem auf das Lösen einer rationalen Gleichung reduziert, die dann ähnliche Eigenschaften wie die Säkulärgleichung besitzt, deren Lösung allerdings nicht so problemlos zu realisieren ist.

Die Eigenvektoren erhält man schließlich durch inverse Vektoriteration, wobei die zuvor berechneten Eigenwerte als Shiftparameter dazu dienen, die jeweils zugehörigen Eigenvektoren *anzusteuern*.

4.2.1 Divide and Conquer

Divide

Sei $l \in \mathbb{N}$ und $n = 2^l$. Sei $A \in \mathcal{H}_l(1)$ symmetrisch, mit der Matrixklasse aus Kapitel 3.2, wobei der lokale Rang hier auf 1 festgelegt sei. Ideen zur möglichen Erweiterung des Algorithmus auf den allgemeinen Fall $k \in \mathbb{N}$ folgen in Kapitel 6.2.2.

A lässt sich unter diesen Voraussetzungen nach Definition 10 schreiben als

$$A \;=\; \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix}$$

mit $A_1, A_2, R \in \mathbb{R}^{\frac{n}{2}\times\frac{n}{2}}$, wobei genauer $R \in \mathcal{R}(1)$, also $\mathrm{Rang}(R) \leq 1$.

Um den einfachen Fall auszuschließen, bei dem das Problem sofort komplett zerfällt, gelte $\mathrm{Rang}(R) = 1$.

Dann existieren Vektoren $\hat{a}, \hat{b} \in \mathbb{R}^{\frac{n}{2}}$ so, dass $R = \hat{a}\hat{b}^T$. Damit gilt

$$A \;=\; \begin{bmatrix} A_1 & \hat{a}\hat{b}^T \\ \hat{b}\hat{a}^T & A_2 \end{bmatrix}. \tag{4.8}$$

Seien $A_1 = U_1^T D_1 U_1$ und $A_2 = U_2^T D_2 U_2$ die Diagonalisierungen der Matrizen A_1 und A_2 mit U_1, $U_2 \in \mathbb{R}^{\frac{n}{2}\times\frac{n}{2}}$ unitär und $D_1 = \mathrm{diag}(d_{1_1}, \ldots, d_{1_{\frac{n}{2}}})$, $D_2 = \mathrm{diag}(d_{2_1}, \ldots, d_{2_{\frac{n}{2}}})$ die Diagonalmatrizen mit den Eigenwerten.

Diese können durch rekursive Aufrufe des Algorithmus oder bei genügend kleinen Teilproblemen zum Beispiel mit Hilfe des QR-Algorithmus berechnet werden.

Conquer

Aus den Diagonalisierungen der Matrizen A_1 und A_2 auf den Hauptdiagonalblöcken und aus den Informationen aus den Nebendiagonalblöcken soll nun die Diagonalisierung von A berechnet werden.

Sei

$$U := \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} \in \mathbb{R}^{n \times n}. \tag{4.9}$$

Die Ähnlichkeitstransformation $U^T A U$ von A aus (4.8) ergibt

$$\begin{bmatrix} U_1^T & 0 \\ 0 & U_2^T \end{bmatrix} \begin{bmatrix} A_1 & \hat{a}\hat{b}^T \\ \hat{b}\hat{a}^T & A_2 \end{bmatrix} \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} = \begin{bmatrix} U_1^T A_1 U_1 & U_1^T \hat{a}\hat{b}^T U_2 \\ U_2^T \hat{b}\hat{a}^T U_1 & U_2^T A_2 U_2 \end{bmatrix},$$

und mit

$$a := U_1^T \hat{a} \quad \text{und} \quad b := U_2^T \hat{b} \tag{4.10}$$

folgt

$$\begin{bmatrix} U_1^T A_1 U_1 & U_1^T \hat{a}\hat{b}^T U_2 \\ U_2^T \hat{b}\hat{a}^T U_1 & U_2^T A_2 U_2 \end{bmatrix} = \begin{bmatrix} D_1 & ab^T \\ ba^T & D_2 \end{bmatrix} =: A' \tag{4.11}$$

Da A' durch unitäre Ähnlichkeitstransformationen aus A hervorgegangen ist, haben A' und A dieselben Spektren. Daher reicht es, die Eigenwerte von A' zu bestimmen, um diejenigen von A zu erhalten.

Für zwei Spezialfälle lassen sich an A' direkt Eigenwerte ablesen. Der eine Spezialfall tritt ein, wenn die Vektoren a oder b aus (4.10) Nulleinträge enthalten. Dann zerfällt das Problem weiter und man erhält sofort bestimmte Eigenwerte. Genaueres sagt folgendes Lemma aus.

Lemma 11 *Seien $D_j = diag(d_{j_1}, \ldots, d_{j_{\frac{n}{2}}})$ für $j = 1, 2$ und seien $a, b \in \mathbb{R}^{\frac{n}{2}}$, sei $i \in \{1, \ldots, \frac{n}{2}\}$ und A' wie in (4.11). Falls $a_i = 0$ für ein i, dann ist d_{1_i} ein Eigenwert von A' und falls $b_i = 0$ für ein i, dann ist d_{2_i} ein Eigenwert von A'.*

Beweis: Sei $a_i = 0$ für ein $i \in \{1, \ldots, \frac{n}{2}\}$. Dann ist die i-te Zeile von ab^T gleich 0 (und damit auch die i-te Spalte von ba^T):

$$ab^T = \left[\begin{array}{ccc} a_1b_1 & \dots & a_1b_{\frac{n}{2}} \\ \vdots & & \vdots \\ a_{i-1}b_1 & \dots & a_{i-1}b_{\frac{n}{2}} \\ \hline 0 & \dots & 0 \\ \hline a_{i+1}b_1 & \dots & a_{i+1}b_{\frac{n}{2}} \\ \vdots & & \vdots \\ a_{\frac{n}{2}}b_1 & \dots & a_{\frac{n}{2}}b_{\frac{n}{2}} \end{array}\right]$$

und A' hat die Form

$$A' = \left[\begin{array}{cc} D_1 & ab^T \\ ba^T & D_2 \end{array}\right] = \left[\begin{array}{c|c|c} * & 0 & * \\ \hline 0 & d_{1_i} & 0 \\ \hline * & 0 & * \end{array}\right].$$

Damit folgt sofort, dass d_{1_i} ein Eigenwert von A' ist.

Analog folgt aus $b_i = 0$, dass dann die i-te Spalte von ab^T Null wird (und die i-te Zeile von ba^T) und damit d_{2_i} ein Eigenwert von A' ist. □

Im Folgenden wird daher angenommen, dass a und b keine Nulleinträge haben.

Ein weiterer Spezialfall tritt auf, wenn die Diagonalmatrizen mehrfache Eigenwerte haben. Das folgende Lemma zeigt, dass ein Eigenwert, der in D_1 oder D_2 mehrfach auftritt, auch ein Eigenwert von A' ist.

Lemma 12 *Seien $D_j = diag(d_{j_1}, \dots, d_{j_{\frac{n}{2}}})$ mit $d_{j_1} \leq \dots \leq d_{j_{\frac{n}{2}}}$ für $j = 1, 2$ und seien $a, b \in \mathbb{R}^{\frac{n}{2}}$ mit $a_i, b_i \neq 0$ für alle $i \in \{1, \dots, \frac{n}{2}\}$ und A' wie in (4.11). Falls $d_{j_k} = d_{j_{k+1}}$ für ein $j \in \{1, 2\}$ und ein $k \in \{1, \dots, \frac{n}{2} - 1\}$, dann ist d_{j_k} ein Eigenwert von A'.*

Beweis: Gelte $d_{1_k} = d_{1_{k+1}}$ für ein $k \in \{1, \dots, \frac{n}{2} - 1\}$. Betrachte

$$(A' - d_{1_k}I) = \left[\begin{array}{cc} D_1 - d_{1_k}I & ab^T \\ ba^T & D_2 - d_{1_k}I \end{array}\right].$$

Sei $v \in \mathbb{R}^n$ mit

$$v^T = (0, \dots, 0, \underbrace{a_{k+1}}_{\substack{k\text{-te} \\ \text{Stelle}}}, \underbrace{-a_k}_{\substack{(k+1)\text{-te} \\ \text{Stelle}}}, 0, \dots, 0)^T.$$

Da nach Voraussetzung $a_k, a_{k+1} \neq 0$, ist $v \neq 0$ und es gilt weiter

$$(A' - d_{1_k} I)v = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ (d_{1_k} - d_{1_k}) \cdot a_{k+1} \\ (d_{1_{k+1}} - d_{1_k}) \cdot (-a_k) \\ 0 \\ \vdots \\ 0 \\ b_1 a_k \cdot a_{k+1} - b_1 a_{k+1} \cdot a_k \\ \vdots \\ b_{\frac{n}{2}} a_k \cdot a_{k+1} - b_{\frac{n}{2}} a_{k+1} \cdot a_k \end{pmatrix} = 0\,.$$

Also ist $v \in \text{Kern}(A' - d_{1_k} I)$ und $v \neq 0$ und damit ist $A' - d_{1_k} I$ nicht injektiv, also nicht regulär und daraus folgt, dass d_{1_k} ein Eigenwert von A' ist.

Analog folgt aus $d_{2_k} = d_{2_{k+1}}$ für ein $k \in \{1, \ldots, \frac{n}{2} - 1\}$, dass d_{2_k} ein Eigenwert von A' ist. □

Dieser Fall wird im Folgenden ausgeschlossen werden, indem davon ausgegangen wird, dass die Einträge der Diagonalmatrizen D_1 und D_2 jeweils streng monoton steigend sortiert sind.

Wünschenswert wäre jetzt eine Aussage analog zum Lemma 9. Allerdings ist nicht gesichert, dass in den Spektren von D_1 und D_2 keine Eigenwerte auftreten, die auch Eigenwerte von A' sind. Damit sind also $D_1 - \lambda I$ und $D_2 - \lambda I$ für $\lambda \in \sigma(A')$ nicht zwingend regulär.

Es lässt sich jedoch zeigen, dass ein Eigenwert λ von A' nicht auch Eigenwert von D_1 und zusätzlich von D_2 sein kann.

Lemma 13 *Seien $D_j = diag(d_{j_1}, \ldots, d_{j_{\frac{n}{2}}})$ mit $d_{j_1} < \cdots < d_{j_{\frac{n}{2}}}$ für $j = 1, 2$ und seien $a, b \in \mathbb{R}^{\frac{n}{2}}$ mit $a_i, b_i \neq 0$ für alle $i \in \{1, \ldots, \frac{n}{2}\}$ und sei (λ, v) ein Eigenpaar der Matrix A' aus (4.11), wobei $v^T = (v_1^T, v_2^T)^T$ mit $v_1, v_2 \in \mathbb{R}^{\frac{n}{2}}$. Dann gilt*

$$\lambda \notin \sigma(D_1) \cap \sigma(D_2).$$

Beweis: Da (λ, v) ein Eigenpaar von A' ist, gilt

$$A'v = \lambda v \quad \Longleftrightarrow \quad \begin{bmatrix} D_1 & ab^T \\ ba^T & D_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

und daraus ergeben sich die Gleichungen

$$\begin{array}{l} D_1 v_1 + ab^T v_2 = \lambda v_1 \\ ba^T v_1 + D_2 v_2 = \lambda v_2 \end{array} \quad \Longleftrightarrow \quad \begin{array}{l} (D_1 - \lambda I) v_1 = -ab^T v_2 \\ (D_2 - \lambda I) v_2 = -ba^T v_1 \end{array} \tag{4.12}$$

Annahme: Es gilt $\lambda \in \sigma(D_1) \cap \sigma(D_2)$.

Da λ ein Eigenwert von D_1 und auch von D_2 ist, gilt $\lambda = d_{1_i}$ für ein $i \in \{1, \ldots, \frac{n}{2}\}$ und $\lambda = d_{2_j}$ für ein $j \in \{1, \ldots, \frac{n}{2}\}$ und mit (4.12) folgt für die i-te bzw. j-te Zeile

$$0 = (d_{1_i} - \lambda)v_{1_i} = -a_i(b^T v_2) \quad \text{und} \quad 0 = (d_{2_j} - \lambda)v_{2_j} = -b_j(a^T v_1).$$

Da nach Voraussetzung $a_i \neq 0$ und $b_j \neq 0$ ist, folgt daraus $b^T v_2 = 0$ und $a^T v_1 = 0$ und mit (4.12) dann

$$\begin{array}{l} (D_1 - \lambda I)v_1 = 0 \\ (D_2 - \lambda I)v_2 = 0 \end{array} \iff \begin{array}{l} D_1 v_1 = \lambda v_1 \\ D_2 v_2 = \lambda v_2 \end{array}$$

Also ist v_1 ein Eigenvektor von D_1 und v_2 ein Eigenvektor von D_2.

D_1 und D_2 sind Diagonalmatrizen mit nur einfachen Eigenwerten, also folgt daraus $v_1 \in \text{span}\{e_i\}$ und $v_2 \in \text{span}\{e_j\}$, insbesondere gilt $v_{1_i} \neq 0$ und $v_{2_j} \neq 0$.

Damit ist aber $0 = a^T v_1 = a_i v_{1_i}$ ein Widerspruch zu $a_i \neq 0$ und $0 = b^T v_2 = b_j v_{2_j}$ ein Widerspruch zu $b_j \neq 0$.

Also gilt $\lambda \notin \sigma(D_1) \cap \sigma(D_2)$. □

Wie mit den Fällen gemeinsamer Eigenwerte von A' und D_1 oder A' und D_2 umgegangen werden kann, wird später aus Satz 4 folgen.

Für das weitere Vorgehen wird zunächst angenommen, dass $D_1 - \lambda I$ und $D_2 - \lambda I$ für einen Eigenwert λ von A' immer regulär sind. Weiterhin wird noch folgendes Lemma benötigt.

Lemma 14 *Seien D_1, D_2, a, b, λ und v wie im Lemma 13. Zusätzlich seien $D_1 - \lambda I$ und $D_2 - \lambda I$ regulär. Dann folgt*

$$a^T v_1 \neq 0 \quad \textit{und} \quad b^T v_2 \neq 0.$$

Beweis: Es gelten die Gleichungen aus (4.12).

Annahme: Es gilt $a^T v_1 = 0$.

Dann gilt mit (4.12)

$$(D_2 - \lambda I)v_2 = 0.$$

Da $D_2 - \lambda I$ nach Voraussetzung regulär ist, folgt daraus $v_2 = 0$. Dann ist aber auch $b^T v_2 = 0$, und mit $(D_1 - \lambda I)v_1 = -ab^T v_2$ aus (4.12) folgt $v^T = (v_1^T, v_2^T)^T = 0$ im Widerspruch dazu, dass v ein Eigenvektor von A' ist. Also ist $a^T v_1 \neq 0$.

Annahme: Es gilt $b^T v_2 = 0$.

Dann gilt mit (4.12)

$$(D_1 - \lambda I)v_1 = 0.$$

Da $D_1 - \lambda I$ nach Voraussetzung regulär ist, folgt daraus $v_1 = 0$. Dann ist aber auch $a^T v_1 = 0$, also $v = 0$ im Widerspruch dazu, dass v ein Eigenvektor von A' ist. Also gilt $b^T v_2 \neq 0$. □

Seien nun D_1, D_2, a, b, λ und v wie im Lemma 14. Dann gelten die Gleichungen aus (4.12) und es existieren $(D_1 - \lambda I)^{-1}$ und $(D_2 - \lambda I)^{-1}$. Setze

$$\alpha := -b^T v_2 \quad \text{und} \quad \beta := -a^T v_1.$$

Dann gilt mit (4.12) weiter

$$\begin{array}{rcl} (D_1 - \lambda I)v_1 &=& \alpha a \\ (D_2 - \lambda I)v_2 &=& \beta b \end{array} \iff \begin{array}{rcl} v_1 &=& \alpha (D_1 - \lambda I)^{-1} a \\ v_2 &=& \beta (D_2 - \lambda I)^{-1} b \end{array}$$

und Multiplikation mit a^T bzw. b^T von links ergibt dann

$$\begin{array}{rcl} a^T v_1 &=& \alpha a^T (D_1 - \lambda I)^{-1} a \\ b^T v_2 &=& \beta b^T (D_2 - \lambda I)^{-1} b \end{array} \iff \begin{array}{rcl} -\beta &=& \alpha \, (\, a \, , \, (D_1 - \lambda I)^{-1} a \,) \\ -\alpha &=& \beta \, (\, b \, , \, (D_2 - \lambda I)^{-1} b \,) \end{array} .$$

Da $\alpha = -b^T v_2 \neq 0$ und $\beta = -a^T v_1 \neq 0$ nach Lemma 14 gilt, teile beide Seiten durch α bzw. β und erhalte

$$-\frac{\beta}{\alpha} = (\, a \, , \, (D_1 - \lambda I)^{-1} a \,) \tag{4.13}$$

und

$$-\frac{\alpha}{\beta} = (\, b \, , \, (D_2 - \lambda I)^{-1} b \,) \iff -\frac{\beta}{\alpha} = \frac{1}{(\, b \, , \, (D_2 - \lambda I)^{-1} b \,)} . \tag{4.14}$$

Gleichsetzen der beiden Gleichungen (4.13) und (4.14) liefert schließlich

$$(\, a \, , \, (D_1 - \lambda I)^{-1} a \,) = \frac{1}{(\, b \, , \, (D_2 - \lambda I)^{-1} b \,)}$$

oder komponentenweise

$$\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - \lambda} = \frac{1}{\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - \lambda}}$$

$$\iff \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - \lambda} \right) \left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - \lambda} \right) - 1 = 0 .$$

Damit ist das Eigenwertproblem, ähnlich wie in Abschnitt 4.1, auf das Lösen einer rationalen Gleichung reduziert, die zwar Ähnlichkeiten mit der Säkulärgleichung $f(x) = 0$ mit f aus (4.5) aufweist, aber keine so einfache Anordnung ihrer Polstellen und Nullstellen besitzt.

Das nachfolgende Kapitel 4.2.2 wird sich mit der Suche nach den Nullstellen der Funktion

$$r(x) \;:=\; \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - x}\right) \left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - x}\right) - 1 \tag{4.15}$$

befassen.

Die Nullstellen von r sind immer Eigenwerte von A', allerdings ist mit allen Nullstellen von r nicht unbedingt das ganze Spektrum von A' erfasst. Dieses wird mit dem folgenden Satz 4 deutlich. Für den Beweis dieses Satzes wird noch ein Lemma benötigt.

Lemma 15 *Seien $x, y \in \mathbb{R}^n$. Dann gilt:*

$$\det(I - xy^T) \;=\; 1 - y^T x.$$

Beweis: Setze

$$v_i \;:=\; \begin{pmatrix} x_1 y_i \\ \vdots \\ x_n y_i \end{pmatrix} \in \mathbb{R}^n .$$

Da die Determinante linear in jeder Komponente ist, gilt damit

$$\begin{aligned}
\det(I - xy^T) \;&=\; \det(e_1 - v_1, e_2 - v_2, \ldots, e_n - v_n) \\
&=\; \det(e_1, e_2 - v_2, \ldots, e_n - v_n) \\
&\qquad + \det(-v_1, e_2 - v_2, \ldots, e_n - v_n) \\
&=\; \det(e_1, e_2 - v_2, \ldots, e_n - v_n) \\
&\qquad + \det(-v_1, e_2, e_3 - v_3, \ldots, e_n - v_n) \\
&\qquad + \underbrace{\det(-v_1, -v_2, e_3 - v_3, \ldots, e_n - v_n)}_{\substack{= 0, \\ \text{da } v_1 \text{ und } v_2 \text{ linear abhängig}}}
\end{aligned}$$

Wiederholtes Ausnutzen der Multilinearität der Determinante und der Tatsache, dass je v_i und v_j mit $i \neq j$ linear abhängig sind, ergibt schließlich

$$\begin{aligned}
\det(I - xy^T) \;&=\; \det(e_1, \ldots, e_n) + \det(-v_1, e_2, \ldots, e_n) \\
&\qquad + \det(e_1, -v_2, e_3, \ldots, e_n) + \ldots \\
&\qquad + \det(e_1, \ldots, e_{n-2}, -v_{n-1}, e_n) + \det(e_1, \ldots, e_{n-1}, -v_n) \\
&=\; 1 + (-x_1 y_1) + (-x_2 y_2) + \cdots + (-x_{n-1} y_{n-1}) + (-x_n y_n) \\
&=\; 1 + (-1) \cdot \sum_{i=1}^{n} x_i y_i \\
&=\; 1 - y^T x.
\end{aligned}$$

□

Mit Hilfe dieses Lemmas lässt sich nun der folgende Satz beweisen.

Satz 4 (Charakteristisches Polynom von A') *Sei $A' \in \mathbb{R}^{n\times n}$ wie in (4.11) mit Diagonalmatrizen D_1, D_2 und Vektoren $a, b \in \mathbb{R}^{\frac{n}{2}}$ mit $a_i, b_i \neq 0$ für alle $i \in \{1, \ldots, \frac{n}{2}\}$. Sei $x \in \mathbb{R} \setminus (\sigma(D_1) \cup \sigma(D_2))$. Dann ist das charakteristische Polynom von A' gegeben durch*

$$p(x) \;=\; \det(D_1 - xI)\ \det(D_2 - xI)\ (-r(x)).$$

mit r aus (4.15).

Beweis: Es gilt

$$\begin{aligned} p(x) &= \det(A' - xI) \\ &= \det\left(\begin{bmatrix} D_1 - xI & ab^T \\ ba^T & D_2 - xI \end{bmatrix}\right) \\ &= \det(\, L\, R\,) \end{aligned}$$

mit

$$L \;:=\; \begin{bmatrix} I & 0 \\ ba^T(D_1 - xI)^{-1} & I \end{bmatrix}$$

und

$$R \;:=\; \begin{bmatrix} D_1 - xI & ab^T \\ 0 & (D_2 - xI) - ba^T(D_1 - xI)^{-1}ab^T \end{bmatrix}.$$

Da $\det(L) = 1$ gilt, folgt

$$\begin{aligned} p(x) &= \det(\, R\,) \\ &= \det(D_1 - xI) \cdot \det((D_2 - xI) - ba^T(D_1 - xI)^{-1}ab^T) \\ &= \det(D_1 - xI) \cdot \det(D_2 - xI) \\ &\quad \cdot \det(I - (D_2 - xI)^{-1}ba^T(D_1 - xI)^{-1}ab^T). \end{aligned}$$

Mit $x := (D_2 - xI)^{-1}b$ und $y := a^T(D_1 - xI)^{-1}ab^T$ in Lemma 15 folgt weiter

$$\begin{aligned} p(x) &= \det(D_1 - xI) \cdot \det(D_2 - xI) \\ &\quad \cdot (1 - a^T(D_1 - xI)^{-1}ab^T(D_2 - xI)^{-1}b) \\ &= \det(D_1 - xI) \cdot \det(D_2 - xI) \cdot (-r(x)). \end{aligned}$$

□

Das ganze Spektrum von A' ist also durch die Nullstellen von r und durch die d_{i_j} aus $\sigma(D_1)$ und $\sigma(D_2)$ gegeben, die keine Polstellen von r sind.

Die Singularitäten der rationalen Funktion r werden im charakteristischen Polynom p durch $\det(D_1 - \lambda I)$ und $\det(D_2 - \lambda I)$ abgefangen. Es ist

daher sinnvoll, nach den Nullstellen von r zu suchen und dann, in dem Fall dass r weniger als n Nullstellen liefert, die Einträge von D_1 und D_2, also $d_{1_1}, \ldots, d_{1_{\frac{n}{2}}}, d_{2_1}, \ldots, d_{2_{\frac{n}{2}}}$ zu überprüfen. Wenn ein d_{i_j} keine Singularität von r ist, hat p dort eine Nullstelle und damit ist dieses d_{i_j} dann ein weiterer Eigenwert von A'.

Für spätere Betrachtungen ist es noch von Bedeutung, dass die Rang-2-Störung von A' sehr einfach in zwei Rang-1-Störungen aufgeteilt werden kann, wie folgendes Lemma beweist.

Lemma 16 *Seien $a, b \in \mathbb{R}^{\frac{n}{2}}$ mit $\|a\|\|b\| \neq 0$. Dann gilt*

$$\begin{bmatrix} 0 & ab^T \\ ba^T & 0 \end{bmatrix} = z_1 z_1^T - z_2 z_2^T$$

mit

$$z_1 := \frac{1}{\sqrt{2\|a\|\|b\|}} \begin{pmatrix} \|b\|a \\ \|a\|b \end{pmatrix} \quad \textit{und} \quad z_2 := \frac{1}{\sqrt{2\|a\|\|b\|}} \begin{pmatrix} \|b\|a \\ -\|a\|b \end{pmatrix}.$$

Beweis: Seien z_1 und z_2 wie oben. Dann gilt:

$$z_1 z_1^T = \begin{bmatrix} \frac{\|b\|}{2\|a\|} aa^T & \frac{1}{2} ab^T \\ \frac{1}{2} ba^T & \frac{\|a\|}{2\|b\|} bb^T \end{bmatrix} \quad \text{und} \quad z_2 z_2^T = \begin{bmatrix} \frac{\|b\|}{2\|a\|} aa^T & -\frac{1}{2} ab^T \\ -\frac{1}{2} ba^T & \frac{\|a\|}{2\|b\|} bb^T \end{bmatrix},$$

und es folgt

$$z_1 z_1^T - z_2 z_2^T = \begin{bmatrix} 0 & ab^T \\ ba^T & 0 \end{bmatrix}.$$

□

4.2.2 Die Funktion r: Eigenschaften und Lösung von $r(x) = 0$

Eigenschaften von $r(x) = 0$

Die Gleichung $r(x) = 0$ hat keine so regelmäßige Anordnung ihrer Polstellen und Nullstellen wie die Säkulärgleichung, man kann aber ähnliche Eigenschaften zeigen.

Folgender Satz besagt, dass, wenn alle d_{1_i} und d_{2_j}, mit $i, j \in \{1, \ldots, \frac{n}{2}\}$, *ineinandersortiert* werden, zwischen je zwei dieser Stellen entweder eine Nullstelle, zwei oder gar keine Nullstellen zu finden sind. Darüberhinaus lässt sich ein Intervall angegeben, in dem alle Nullstellen von r liegen.

Satz 5 *Seien*

$$A' \;=\; \begin{bmatrix} D_1 & ab^T \\ ba^T & D_2 \end{bmatrix} \quad , \quad D \;=\; \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}$$

und $D_j := diag(d_{j_1}, \ldots, d_{j_{\frac{n}{2}}})$ *mit* $d_{j_1} < \cdots < d_{j_{\frac{n}{2}}}$ *für* $j = 1, 2$ *und seien* $a, b \in \mathbb{R}^{\frac{n}{2}}$ *mit* $a_i, b_i \neq 0$ *für alle* $i \in \{1, \ldots, \frac{n}{2}\}$ *und gelte* $\sigma(D_1) \cap \sigma(D_2) = \emptyset$. *Sei mit* $d_1, \ldots, d_n$ *die endliche Folge bezeichnet, die aus dem Sortieren aller* d_{1_i} *und* d_{2_i} *entsteht. Dann gilt:*

(i) Im Intervall (d_k, d_{k+1}) *für* $k \in \{1, \ldots, n-1\}$ *gibt es entweder eine, zwei oder keine Nullstellen von* r.

(ii) Alle Nullstellen von r *liegen im Intervall* $[\, d_1 - \|a\|\|b\| \;,\; d_n + \|a\|\|b\| \,]$, *insbesondere gilt* $\sigma(A') \subseteq [\, d_1 - \|a\|\|b\| \;,\; d_n + \|a\|\|b\| \,]$.

Beweis: Es gilt

$$A' \;=\; \begin{bmatrix} D_1 & ab^T \\ ba^T & D_2 \end{bmatrix} \;=\; \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} + \begin{bmatrix} 0 & ab^T \\ ba^T & 0 \end{bmatrix} .$$

Seien

$$z_1 \;:=\; \frac{1}{\sqrt{2\|a\|\|b\|}} \begin{pmatrix} \|b\|\, a \\ \|a\|\, b \end{pmatrix} \quad \text{und} \quad z_2 \;:=\; \frac{1}{\sqrt{2\|a\|\|b\|}} \begin{pmatrix} \|b\|\, a \\ -\|a\|\, b \end{pmatrix} .$$

Damit haben z_1 und z_2 keine Nulleinträge, da a und b nach Voraussetzung keine Nulleinträge haben, und es gilt nach Lemma 16

$$A' \;=\; D + z_1 z_1^T - z_2 z_2^T .$$

Um Satz 3 anwenden zu können, müssen die Einträge der Diagonalmatrix aufsteigend sortiert sein. Falls nur jeweils D_1 und D_2 sortiert sind, wird eine Permutation angewendet. Durch Permutation entstehen in z_1 und z_2 keine Nulleinträge, daher seien o.B.d.A. die Einträge der Diagonalmatrix D aufsteigend sortiert.

In Abschnitt 4.1.1 wurde gezeigt, dass die Eigenwerte von $D + z_1 z_1^T$ genau die Nullstellen μ_i der Säkulärgleichung mit

$$f(x) \;=\; 1 + \sum_{i=1}^{n} \frac{(z_1)_i^2}{d_i - x}$$

sind. Mit Satz 3 folgt dann, dass zwischen je zwei Polstellen davon genau eine Nullstelle μ_i zu finden ist, also

$$d_1 < \mu_1 < d_2 < \cdots < \mu_{n-1} < d_n < \mu_n < d_n + z_1^T z_1 \; . \tag{4.16}$$

Sei $D' = U(D + z_1 z_1^T)U^T$ die Diagonalisierung von $D + z_1 z_1^T$, also stehen die Eigenwerte von $D + z_1 z_1^T$ auf der Diagonalen von $D' = \operatorname{diag}(\mu_1, \ldots, \mu_n)$. Dann gilt für $A'' := UA'U^T$

$$\begin{aligned} A'' &= U(D + z_1 z_1^T - z_2 z_2^T)U^T \\ &= U(D + z_1 z_1^T)U^T - U z_2 z_2^T U^T \\ &= D' - z_2' {z_2'}^T \end{aligned}$$

mit

$$z_2' := U z_2 .$$

Fall 1: Falls z_2' keine Nulleinträge besitzt, kann nach Bemerkung 4 für $A'' = D' - z_2' {z_2'}^T$ die Gleichung $\hat{f}(x) = 0$ mit

$$\hat{f}(x) = 1 - \sum_{i=1}^{n} \frac{(z_2')_i^2}{\mu_i - x}$$

hergeleitet werden, und nach Bemerkung 5 trennen die Nullstellen λ_i von $\hat{f}$ die μ_i in der Form

$$\mu_1 - {z_2'}^T z_2' < \lambda_1 < \mu_1 < \lambda_2 < \cdots < \mu_{n-1} < \lambda_n < \mu_n . \tag{4.17}$$

Die λ_i sind damit die Eigenwerte von A'' (also auch von A').

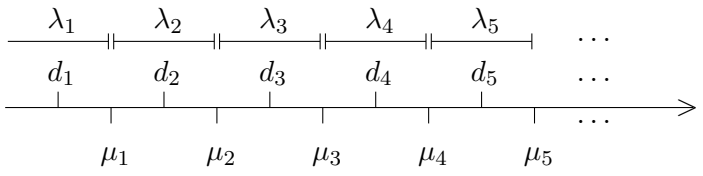
Setze $d_{n+1} := d_n + z_1^T z_1$ und $\mu_0 := \mu_1 - {z_2'}^T z_2'$. Dann folgt mit (4.16) und (4.17) für $i \in \{1, \ldots, n\}$

$$\lambda_i \in (\mu_{i-1}, \mu_i) \subset (d_{i-1}, d_{i+1})$$

Andererseits wird ein Intervall (d_i, d_{i+1}) von höchstens zwei Intervallen (die jeweils genau eine Nullstelle enthalten) überdeckt: (μ_{i-1}, μ_i) und (μ_i, μ_{i+1}).

Also gibt es im Intervall (d_i, d_{i+1})

$$\begin{cases} \text{keine Nullstelle,} & \text{falls } \lambda_i \in (\mu_{i-1}, d_i) \text{ und } \lambda_{i+1} \in (d_{i+1}, \mu_{i+1}), \\ \text{eine Nullstelle,} & \text{falls } \lambda_i \in (\mu_{i-1}, d_i) \text{ und } \lambda_{i+1} \in (\mu_i, d_{i+1}) \\ & \text{oder } \lambda_i \in (d_i, \mu_i) \text{ und } \lambda_{i+1} \in (d_{i+1}, \mu_{i+1}), \\ \text{zwei Nullstellen,} & \text{falls } \lambda_i, \lambda_{i+1} \in (d_i, d_{i+1}). \end{cases}$$



Damit folgt die Behauptung *(i)*.

Für die größte Nullstelle λ_n von r gilt also

$$\lambda_n < \mu_n < d_n + z_1^T z_1$$

und für die kleinste Nullstelle λ_1 von r

$$\mu_1 - {z_2'}^T z_2' < \lambda_1 < \mu_1$$

und da $d_1 < \mu_1$, ist auch $d_1 - {z_2'}^T z_2' < \mu_1 - {z_2'}^T z_2'$ und damit ist

$$d_1 - {z_2'}^T z_2' < \lambda_1 \ .$$

Nach Satz 4, ist $\sigma(A')$ eine Teilmenge der Menge der Nullstellen von r vereinigt mit den Spektren von D_1 und D_2. Da $\sigma(D_1)$ und $\sigma(D_2)$ aber Teilmengen der Menge aller d_i's sind, folgt

$$\sigma(A') \ \subseteq \ [\, d_1 - {z_2'}^T z_2' \ , \ d_n + z_1^T z_1 \,]$$

und da U unitär ist, gilt weiter

$$\begin{aligned} \sigma(A') \ &\subseteq \ [\, d_1 - (Uz_2)^T U z_2 \ , \ d_n + z_1^T z_1 \,] \\ &= \ [\, d_1 - z_2^T U^T U z_2 \ , \ d_n + z_1^T z_1 \,] \\ &= \ [\, d_1 - z_2^T z_2 \ , \ d_n + z_1^T z_1 \,] \ . \end{aligned}$$

Wegen $z_2^T z_2 = z_1^T z_1 = \frac{2\|a\|^2\|b\|^2}{2\|a\|\|b\|} = \|a\|\|b\|$ folgt schließlich *(ii)*:

$$\sigma(A') \ \subseteq \ [\, d_1 - \|a\|\|b\| \ , \ d_n + \|a\|\|b\| \,] \ .$$

Fall 2: Falls z_2' Nulleinträge hat, ist nach Lemma 7 für jedes $(z_2')_i = 0$ das entsprechende μ_i ein Eigenwert von $D' - z_2' z_2'^T$ und nach Lemma 10 existiert eine Permutationsmatrix P, so dass

$$\begin{aligned} P(D' - z_2' z_2'^T)P^T \ &= \ PD'P^T - Pz_2' z_2'^T P^T \\ &= \ \ \mathrm{diag}\,(\widetilde{\mu}_1, \ldots, \widetilde{\mu}_n) - ww^T \end{aligned}$$

wobei $w := Pz_2'$ mit $w_i \neq 0$ für $i \in \{1, \ldots, k\}$ und $w_i = 0$ für $i \in \{k+1, \ldots, n\}$ sowie $\widetilde{\mu}_1 < \cdots < \widetilde{\mu}_k$. Also kann die Gleichung $\widetilde{f}(x) = 0$ mit

$$\widetilde{f}(x) \ = \ 1 - \sum_{i=1}^{k} \frac{w_i^2}{\widetilde{\mu}_i - x}$$

aufgestellt werden, für deren Lösungen $\widetilde{\lambda}_i$ nach Bemerkung 5 gilt:

$$\widetilde{\mu}_1 - w^T w < \widetilde{\lambda}_1 < \widetilde{\mu}_1 < \widetilde{\lambda}_2 < \cdots < \widetilde{\mu}_{k-1} < \widetilde{\lambda}_k < \widetilde{\mu}_k \qquad (4.18)$$

Setze $\widetilde{\mu}_0 := \widetilde{\mu}_1 - w^T w$, sei d_{n+1} wie in Fall 1 und seien die $n-k$ mit Lemma 7 bestimmten Eigenwerte mit $\hat{\lambda}_l$ für $l \in \{1, \ldots, n-k\}$ bezeichnet.

Nach (4.16) wird ein Intervall (d_i, d_{i+1}) von höchstens zwei Intervallen (μ_{i-1}, μ_i) und (μ_i, μ_{i+1}) für $i \in \{1, \ldots, n-1\}$ überdeckt, also auch von höchstens zwei Intervallen $(\widetilde{\mu}_{j-1}, \widetilde{\mu}_j)$ und $(\widetilde{\mu}_j, \widetilde{\mu}_{j+1})$ für $j \in \{1, \ldots, k-1\}$.

Falls $\mu_i = \widetilde{\mu}_j$ für ein $j \in \{1, \ldots, k\}$ folgt mit (4.18) wie im Fall 1 die Behauptung *(i)*.

Falls $\mu_i = \hat{\lambda}_l$ für ein $l \in \{1, \ldots, n-k\}$, gilt

$$(d_i, d_{i+1}) \ \subset \ (\widetilde{\mu}_{j-1}, \widetilde{\mu}_j) \quad \text{für ein } j \in \{1, \ldots, k\},$$

mit (4.18) und $\hat{\lambda}_l \in (d_i, d_{i+1})$ folgt, dass es im Intervall (d_i, d_{i+1})

$$\begin{cases} \text{eine Nullstelle gibt,} & \text{falls } \widetilde{\lambda}_j \in (\widetilde{\mu}_{j-1}, \widetilde{\mu}_j) \setminus (d_i, d_{i+1}), \\ \text{zwei Nullstellen gibt,} & \text{falls } \widetilde{\lambda}_j \in (d_i, d_{i+1}). \end{cases}$$

Damit folgt die Behauptung *(i)*.

Für den größten Eigenwert $\lambda_{\max}$ gilt

$$\widetilde{\lambda}_k \leq \lambda_{\max} \leq \widetilde{\mu}_k \leq \mu_n < d_n + z_1^T z_1$$

und da $d_1 < \mu_1$ und $\mu_1 \leq \widetilde{\mu}_1$, gilt für den kleinsten Eigenwert $\lambda_{\min}$

$$d_1 - w^T w < \mu_1 - w^T w \leq \lambda_{\min} \leq \widetilde{\lambda}_1$$

also folgt wie in Fall 1 die Behauptung *(ii)*:

$$\begin{aligned} \sigma(A') \ &\subseteq \ [\, d_1 - w^T w \ , \ d_n + z_1'^T z_1' \,] \\ &= \ [d_1 - z_2'^T P^T P z_2' \ , \ d_n + z_1'^T z_1' \,] \\ &= \ [\, d_1 - \|a\|\|b\| \ , \ d_n + \|a\|\|b\| \,] \ . \end{aligned}$$

□

In den Bereichen links der kleinsten Polstelle und rechts der größten kann die Aussage von Satz 5 noch verschärft werden. Folgendes Lemma wird zeigen, dass dort jeweils eine Nullstelle von r existiert.

Lemma 17 *Sei $n \geq 4$ und seien $d_{1_i}, d_{2_j} \in \mathbb{R}$ für $i, j \in \{1, \ldots, \frac{n}{2}\}$ mit $d_{1_1} < \cdots < d_{1_{\frac{n}{2}}}$, $d_{2_1} < \cdots < d_{2_{\frac{n}{2}}}$ und seien $a, b \in \mathbb{R}^{\frac{n}{2}}$, wobei $a_i, b_j \neq 0$ für alle $i, j \in \{1, \ldots, \frac{n}{2}\}$. Definiere $d_{min} := min\{d_{1_1}, d_{2_1}\}$ und $d_{max} := max\{d_{1_{\frac{n}{2}}}, d_{2_{\frac{n}{2}}}\}$. Sei r wie in (4.15).*

Dann gilt für die kleinste Nullstelle λ_1 von r

$$d_{min} - \|a\|\|b\| \quad < \quad \lambda_1 \quad < \quad d_{min}$$

und für die größte Nullstelle λ_n von r gilt

$$d_{max} \quad < \quad \lambda_n \quad < \quad d_{max} + \|a\|\|b\|.$$

Beweis: Sei $\delta := \|a\|\|b\|$. Damit gilt

$$\begin{aligned} r(d_{\max}+\delta) &= \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - d_{\max} - \delta}\right)\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - d_{\max} - \delta}\right) - 1 \\ &= \left(-\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{\delta + d_{\max} - d_{1_i}}\right)\left(-\sum_{i=1}^{\frac{n}{2}} \frac{b_j^2}{\delta + d_{\max} - d_{2_j}}\right) - 1 \end{aligned}$$

und da $(d_{\max} - d_{1_i}) > 0$ für alle $i < \frac{n}{2}$ und $(d_{\max} - d_{2_j}) > 0$ für alle $j < \frac{n}{2}$ gilt dann weiter

$$\begin{aligned} r(d_{\max}+\delta) &< \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{\delta}\right)\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{\delta}\right) - 1 \\ &= \frac{1}{\delta^2}\left(\sum_{i=1}^{\frac{n}{2}} a_i^2\right)\left(\sum_{i=1}^{\frac{n}{2}} b_j^2\right) - 1 \\ &= \frac{\|a\|^2\|b\|^2}{(\|a\|\|b\|)^2} - 1 \\ &= 1 - 1 \\ &= 0\,. \end{aligned}$$

Sei $\varepsilon > 0$. Dann gilt

$$r(d_{\max}+\varepsilon) = \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - d_{\max} - \varepsilon}\right)\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - d_{\max} - \varepsilon}\right) - 1 \xrightarrow[\varepsilon\to 0]{} +\infty$$

da im Fall $d_{1_{\frac{n}{2}}} = d_{\max}$ die erste Summe gegen $-\infty$ geht und die zweite Summe immer < 0 ist und im Fall $d_{2_{\frac{n}{2}}} = d_{\max}$ die zweite Summe gegen $-\infty$ geht und die erste immer < 0 ist.

Weiter kann man an der Ableitung

$$r'(x) = \underbrace{\left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{(d_{1_i} - x)^2}\right)}_{>0}\underbrace{\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - x}\right)}_{\substack{<0 \\ \text{für } x > d_{\max}}} + \underbrace{\left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - x}\right)}_{\substack{<0 \\ \text{für } x > d_{\max}}}\underbrace{\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{(d_{2_j} - x)^2}\right)}_{>0}$$

ablesen, dass r' für $x > d_{\max}$ negativ ist, also r dort monoton fallend verläuft. Da also r bei $d_{\max}+\varepsilon$ gegen $+\infty$ geht, bei $d_{\max}+\delta$ negativ ist und dazwischen monoton fallend ist, folgt schließlich mit dem Zwischenwertsatz, dass eine Nullstelle λ von r existiert mit

$$d_{\max} \quad < \quad \lambda \quad < \quad d_{\max} + \|a\|\|b\|.$$

Analog zeigt man, dass ein λ existiert mit

$$d_{\min} - \|a\|\|b\| \quad < \quad \lambda \quad < \quad d_{\min}.$$

□

Lösung mit Bisektion und Newton-Verfahren

In Abschnitt 4.1.2 wurde für die Lösung der Säkulärgleichung ein iteratives Verfahren vorgestellt mit dem jeweils in einem Intervall zwischen zwei Polstellen die Nullstelle gefunden werden konnte. Dabei war in jedem Iterationsschritt eine quadratische Gleichung zu lösen, die in dem aktuellen Intervall jeweils nur eine Lösung hatte, die dann als neue Iterierte diente.

Dieses Verfahren lässt sich nur schwer auf die Gleichung $r(x) = 0$ mit r aus (4.15) übertragen. Es führt zu einer Gleichung vierten Grades, die in jedem Iterationsschritt zu lösen wäre, wenn sichergestellt sein soll, dass daraus eine eindeutige neue Iterierte resultiert. Da es nicht sehr effizient ist, eine Gleichung vierten Grades je Iterationsschritt zu lösen, wird hier auf das Newton-Verfahren zurückgegriffen.

Bevor das Newton-Verfahren überhaupt angewendet werden kann, muss festgestellt werden, wie viele Nullstellen in dem aktuellen Intervall (d_k, d_{k+1}) genau zu finden sind, nach Satz 5 können das eine, zwei oder keine Nullstellen sein.

Dazu wird das Verhalten der Funktion r nahe der Polstellen d_k und d_{k+1} untersucht. Wenn die Vorzeichen von r nahe der beiden Intervallgrenzen gleich sind, existiert eine Extremstelle von r in dem Intervall und wenn sich das Vorzeichen des Funktionswertes dieser Extremstelle von den Vorzeichen nahe der Intervallgrenzen unterscheidet, existieren zwei Nullstellen in (d_k, d_{k+1}), wenn es sich nicht unterscheidet, gibt es dort keine Nullstelle. In dem Fall, dass zwei Nullstellen in dem Intervall sind, konvergiert das Newton-Verfahren mit einem Startwert nahe der linken Intervallgrenze gegen die kleinere und mit Startwert nahe der rechten Intervallgrenze gegen die größere Nullstelle in dem Intervall, da die Funktion hier jeweils zwischen Polstelle und Extremwert monoton verläuft.

Wenn sich die Vorzeichen von r in (d_k, d_{k+1}) nahe der beiden Polstellen unterscheiden, ist genau eine Nullstelle in dem Intervall zu finden. Hier besteht weiterhin die Schwierigkeit, dass die Folge der Newton-Iterierten das Intervall verlassen könnte.

Um dem entgegenzuwirken könnte man nun zu dem Bisektionsverfahren übergehen, das in einem solchen Intervall, in dem nur eine Nullstelle existiert auf jeden Fall den gesuchten Wert liefern würde, allerdings nur mit linearer Konvergenz. Um also das Newton-Verfahren mit seiner lokalen quadratischen Konvergenz auch hier benutzen zu können, wird in diesem Fall

zunächst ein Bisektionsschritt ausgeführt, also eine Intervallhalbierung, um den Startwert für das Newton-Verfahren, der dann in der Mitte des neuen Intervalls gewählt wird, *näher an die Nullstelle heranzubringen.*

Auch wenn in der Praxis oft ein Bisektionsschritt ausreicht, um zu erreichen, dass das Newton-Verfahren gegen den gesuchten Wert konvergiert, muss jede Newton-Iterierte daraufhin überprüft werden, ob sie noch in dem Intervall liegt, in dem der Startwert lag. Wenn nicht, müssen das Newton-Verfahren abgebrochen und weitere Bisektionsschritte durchgeführt werden, bevor das Newton-Verfahren auf das neue, verkleinerte Intervall angewendet werden kann.

Das genaue Vorgehen mit dieser Kombination aus Bisektion und Newton-Verfahren in der Implementierung wird in Kapitel 5.1 beschrieben.

4.2.3 Berechnung der Eigenvektoren

Nachdem über die Nullstellensuche von r alle Eigenwerte von A' aus (4.11) (und damit von A aus (4.8)) berechnet wurden, bleibt noch die Suche nach den zugehörigen Eigenvektoren. Dazu wird hier auf die inverse Vektoriteration zurückgegriffen, wobei sich die spezielle Form von A', die fast Diagonalgestalt hat, gut ausnutzen lässt.

Inverse Vektoriteration

Bevor die inverse Vektoriteration auf das spezielle Problem mit A' angewendet wird, soll sie zunächst allgemein betrachtet werden.

Sei $M \in \mathbb{R}^{n \times n}$ eine Matrix, von der die Eigenwerte und Eigenvektoren berechnet werden sollen. Dazu wird $M - \mu I$ an Stelle von M betrachtet, wobei $\mu \notin \sigma(M)$ **Shiftparameter** genannt wird. Damit ist $M - \mu I$ regulär und es gilt das folgende Lemma.

Lemma 18 *Sei λ ein Eigenwert von M. Dann ist*

$$\lambda - \mu \quad \textit{ein Eigenwert von} \quad M - \mu I$$

und

$$\frac{1}{\lambda - \mu} \quad \textit{ein Eigenwert von} \quad (M - \mu I)^{-1}.$$

Außerdem sind die Eigenvektoren von $M - \mu I$ diesselben wie die von M.

Beweis: Sei x ein zu λ gehöriger Eigenvektor von M. Es gilt $Mx = \lambda x$ und damit weiter

$$(M - \mu I)x \;=\; Mx - \mu Ix \;=\; \lambda x - \mu x$$

und schließlich

$$(M - \mu I)x \;=\; (\lambda - \mu)x. \tag{4.19}$$

Also ist $(\lambda - \mu\,,\, x)$ ein Eigenpaar von $(M - \mu I)$.

Invertieren von (4.19) ergibt, dass $\frac{1}{\lambda - \mu}$ ein Eigenwert von $(M - \mu I)^{-1}$ ist. □

Definition 11 (Inverse Vektoriteration) *Definiere die* ***inverse Vektoriteration****, oder auch* ***inverse Iteration****, durch*

$$x^{(m)} \;:=\; (M - \mu I)^{-1}\, x^{(m-1)}.$$

Die approximativen Eigenwerte erhält man über den ***Rayleigh-Quotienten***

$$\lambda^{(m)} \;:=\; \frac{(Mx^{(m)}\,,\, x^{(m)})}{(x^{(m)}\,,\, x^{(m)})}.$$

Da man im Allgemeinen die Inverse nicht berechnen will, ist hier in jedem Iterationsschritt ein lineares Gleichungssystem

$$(M - \mu I)\, x^{(m)} \;=\; x^{(m-1)} \tag{4.20}$$

zu lösen. Eine Faktorisierung von $M - \mu I$ wäre dabei aber nur einmal auszuführen.

Für einen Shiftparameter μ, der sehr nahe an einem Eigenwert liegt, ist dieses Gleichungssystem (4.20) sehr schlecht konditioniert. Allerdings fällt die Fehlerverstärkung, die dadurch entsteht, in Richtung des Eigenvektors aus und ist damit harmlos. Genaueres dazu ist in [9, 4 - 3] zu finden.

Bemerkung 6 *Die inverse Vektoriteration liefert den Eigenwert, der μ am nächsten ist. Das bedeutet, dass jeder Eigenwert von M gefunden werden kann, wenn der Shiftparameter hinreichend nahe am gesuchten Eigenwert gewählt wird.*

Und je näher μ an einem Eigenwert liegt, desto schneller konvergiert die inverse Vektoriteration.

Nun sollen mit Hilfe der inversen Vektoriteration die Eigenvektoren für das spezielle Problem von A' aus (4.11) berechnet werden. Dabei ist es von großem Vorteil, dass zu diesem Zeitpunkt bereits alle Eigenwerte $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ von A' bekannt sind. So kann ein Wert nahe des jeweiligen Eigenwertes als Shiftparameter eingesetzt werden, um den zugehörigen Eigenvektor zu erhalten.

Dabei ergibt sich allerdings das Problem, dass $A' - \lambda_i I$ mit $i \in \{1, \ldots, n\}$ singulär ist. Um dem entgegenzuwirken, wird dem Eigenwert λ_i eine Störung hinzugefügt, so dass $\lambda_i + \varepsilon$ mit einem kleinen $\varepsilon > 0$ als Shiftparameter eingesetzt wird.

Bei der Wahl von ε ist noch zu beachten, dass der Abstand zum nächsten Eigenwert λ_{i+1} eine Rolle spielt. Wenn λ_i und λ_{i+1} sehr dicht beieinander liegen, besteht die Gefahr, dass die inverse Vektoriteration gegen den Eigenvektor von λ_{i+1} und nicht gegen den von λ_i konvergiert. Eine Möglichkeit das ε geeignet zu wählen wäre hier

$$\varepsilon \;:=\; \frac{|\lambda_{i+1} - \lambda_i|}{\delta} \quad \text{mit } \delta \gg 2.$$

Aus Bemerkung 6 ist bekannt, dass die Konvergenz besser wird, je kleiner dieses ε gewählt wird.

Eine geeignete Faktorisierung für $A' - (\lambda_i + \varepsilon)I$ zu finden, um das Gleichungssystem (4.20) zu lösen, ist Gegenstand des folgenden Abschnitts.

Ausnutzen der Matrixstruktur

Bei der inversen Vektoriteration mit einem der bereits berechneten Eigenwerte λ von A' in leicht gestörter Form $\widetilde{\lambda} := \lambda + \varepsilon$ als Shiftparameter, ist in jedem Iterationsschritt ein Gleichungssystem der Form

$$\begin{bmatrix} D_1 - \widetilde{\lambda} I & ab^T \\ ba^T & D_2 - \widetilde{\lambda} I \end{bmatrix} x^{(m)} \;=\; x^{(m-1)} \tag{4.21}$$

zu lösen. Dazu wird die Matrix mit einer Block-LR-Zerlegung faktorisiert.

Lemma 19 *Die Matrix A' aus (4.11) lässt sich in ein Produkt einer unteren linken Dreiecksmatrix L und einer oberen rechten Block-Dreiecksmatrix R zerlegen. Es gilt*

$$A' \;=\; LR \tag{4.22}$$

mit

$$L \;:=\; \begin{bmatrix} I & 0 \\ ba^T (D_1 - \widetilde{\lambda} I)^{-1} & I \end{bmatrix}$$

und

$$R \;:=\; \begin{bmatrix} D_1 - \widetilde{\lambda} I & ab^T \\ 0 & (D_2 - \widetilde{\lambda} I) - ba^T (D_1 - \widetilde{\lambda} I)^{-1} ab^T \end{bmatrix}.$$

Beweis: Es ist

$$\begin{aligned} LR &= \begin{bmatrix} D_1 - \widetilde{\lambda} I & ab^T \\ ba^T (D_1 - \widetilde{\lambda} I)^{-1} (D_1 - \widetilde{\lambda} I) & \begin{array}{c} ba^T (D_1 - \widetilde{\lambda} I)^{-1} ab^T + \\ (D_2 - \widetilde{\lambda} I) - ba^T (D_1 - \widetilde{\lambda} I)^{-1} ab^T \end{array} \end{bmatrix} \\ &= \begin{bmatrix} D_1 - \widetilde{\lambda} I & ab^T \\ ba^T & D_2 - \widetilde{\lambda} I \end{bmatrix} \\ &= A' \end{aligned}$$

□

Mit der Zerlegung $A' = LR$ kann (4.21) in zwei Schritten gelöst werden. Der erste Schritt ist das Lösen des unteren linken Gleichungssystems

$$L\, y \;=\; x^{(m-1)}$$

mit dem Lösungsvektor $y \in \mathbb{R}^n$. Genaueres dazu ist im Kapitel 5.2 zu finden.

Der zweite Schritt ist das Lösen des Gleichungssystems

$$R\, x^{(m)} \;=\; y$$

wobei R obere Block-Dreiecksgestalt hat, also von der Form

$$R \;=\; \left[\begin{array}{ccc|c} d_{1_1} - \widetilde{\lambda} & & & \\ & \ddots & & ab^T \\ & & d_{1_{\frac{n}{2}}} - \widetilde{\lambda} & \\ \hline & 0 & & * \end{array}\right]$$

ist. Dabei kann die *obere Hälfte* des Gleichungssystems wie gewohnt gelöst werden, wozu allerdings die Lösung der *unteren Hälfte* benötigt wird.

Dafür wird das Teilproblem

$$\left((D_2 - \widetilde{\lambda} I) - ba^T (D_1 - \widetilde{\lambda} I)^{-1}\, ab^T\right) \begin{pmatrix} x^{(m)}_{\frac{n}{2}+1} \\ \vdots \\ x^{(m)}_n \end{pmatrix} \;=\; \begin{pmatrix} y_{\frac{n}{2}+1} \\ \vdots \\ y_n \end{pmatrix}$$

betrachtet, das direkt gelöst werden soll, also

$$\begin{pmatrix} x^{(m)}_{\frac{n}{2}+1} \\ \vdots \\ x^{(m)}_n \end{pmatrix} \;=\; ((D_2 - \widetilde{\lambda} I) - ba^T (D_1 - \widetilde{\lambda} I)^{-1}\, ab^T)^{-1} \begin{pmatrix} y_{\frac{n}{2}+1} \\ \vdots \\ y_n \end{pmatrix}. \quad (4.23)$$

Um die Inverse von

$$X \;:=\; (D_2 - \widetilde{\lambda} I) - b a^T (D_1 - \widetilde{\lambda} I)^{-1}\, a b^T, \tag{4.24}$$

also einer Matrix $(D_2 - \widetilde{\lambda} I)$ mit einer Rang-1-Störung $b a^T (D_1 - \widetilde{\lambda} I)^{-1}\, a b^T$, zu berechnen, eignet sich die Sherman-Morrison-Formel.

Satz 6 (Sherman-Morrison-Formel) *Sei $M \in \mathbb{R}^{n\times n}$ eine reguläre Matrix und seien $u, v \in \mathbb{R}^n$, so dass $1 + v^T M^{-1} u \neq 0$. Dann gilt*

$$(M + uv^T)^{-1} \;=\; M^{-1} - \frac{M^{-1} u v^T M^{-1}}{1 + v^T M^{-1} u}. \tag{4.25}$$

Beweis: Es ist

$$\begin{aligned}
&(M + uv^T)\left(M^{-1} - \frac{M^{-1} u v^T M^{-1}}{1 + v^T M^{-1} u}\right) \\
&= MM^{-1} + uv^T M^{-1} - \frac{MM^{-1} u v^T M^{-1} + uv^T M^{-1} u v^T M^{-1}}{1 + v^T M^{-1} u} \\
&= I + uv^T M^{-1} - \frac{uv^T M^{-1} + u(v^T M^{-1} u) v^T M^{-1}}{1 + v^T M^{-1} u} \\
&= I + uv^T M^{-1} - \frac{(1 + v^T M^{-1} u) u v^T M^{-1}}{1 + v^T M^{-1} u} \qquad \text{da } v^T M^{-1} u \in \mathbb{R} \\
&= I + uv^T M^{-1} - uv^T M^{-1} \\
&= I
\end{aligned}$$

und ebenso ist

$$\left(M^{-1} - \frac{M^{-1} u v^T M^{-1}}{1 + v^T M^{-1} u}\right)\left(M + uv^T\right) \;=\; I\,,$$

und damit folgt die Behauptung. □

Wähle $M = D_2 - \widetilde{\lambda} I$, $u = -b$ und $v^T = a^T (D_1 - \widetilde{\lambda} I)^{-1} a b^T$. Da

$$1 + v^T M^{-1} u \;=\; 1 - a^T (D_1 - \widetilde{\lambda} I)^{-1} a b^T (D_2 - \widetilde{\lambda} I)^{-1} b \;=\; r(\widetilde{\lambda})$$

und $\widetilde{\lambda}$ gerade so gewählt wurde, dass dies kein Eigenwert ist, gilt $r(\widetilde{\lambda}) \neq 0$ und damit kann die Sherman-Morrison-Formel auf die Matrix X aus (4.24) angewendet werden. Dieses ergibt dann

$$X^{-1} \;=\; (D_2 - \widetilde{\lambda} I)^{-1} + \frac{(D_2 - \widetilde{\lambda})^{-1}\, b a^T (D_1 - \widetilde{\lambda} I)^{-1}\, a b^T (D_2 - \widetilde{\lambda} I)^{-1}}{1 - a^T (D_1 - \widetilde{\lambda} I)^{-1}\, a b^T (D_2 - \widetilde{\lambda} I)^{-1}\, b}$$

und mit

$$\alpha := a^T (D_1 - \widetilde{\lambda} I)^{-1}\, a \quad (\in \mathbb{R}), \quad \beta := b^T (D_2 - \widetilde{\lambda} I)^{-1}\, b \quad (\in \mathbb{R}) \tag{4.26}$$

und

$$\widehat{D} \;:=\; (D_2 - \widetilde{\lambda} I)^{-1}\, bb^T (D_2 - \widetilde{\lambda} I)^{-1} \quad (\in \mathbb{R}^{\frac{n}{2}\times\frac{n}{2}})$$

folgt für (4.23) schließlich

$$\begin{pmatrix} x^{(m)}_{\frac{n}{2}+1} \\ \vdots \\ x^{(m)}_n \end{pmatrix} \;=\; \left((D_2 - \widetilde{\lambda} I)^{-1} + \frac{\alpha}{1-\alpha\beta}\; \widehat{D} \right) \begin{pmatrix} y_{\frac{n}{2}+1} \\ \vdots \\ y_n \end{pmatrix} . \tag{4.27}$$

Diese Gleichung kann in der Implementierung gut umgesetzt werden. Ausführlicher wird darauf in Kapitel 5.2 eingegangen.

Der so berechnete Lösungsvektor $x^{(m)}$ ist die neue Iterierte für die inverse Vektoriteration mit Shift. Diese konvergiert nach Lemma 18 schließlich gegen den Eigenvektor x zum Eigenwert λ von A'.

Gesucht war der Eigenvektor von A, aber A' ist durch eine unitäre Ähnlichkeitstransformation aus A entstanden und daher gilt mit U aus (4.9)

$$A'x \;=\; \lambda x \quad\Longleftrightarrow\quad U^T A U x \;=\; \lambda x \quad\Longleftrightarrow\quad A(Ux) \;=\; \lambda (Ux).$$

Also ist $v := Ux$ der gesuchte Eigenvektor zum Eigenwert λ von A.

Kapitel 5

Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung der Ergebnisse aus Kapitel 4 in der Implementierung.

Im Unterkapitel 5.1 wird auf die Kombination aus Bisektion und Newton-Verfahren zur Bestimmung der Nullstellen der Funktion r aus Kapitel 4.2.2 eingegangen und es wird beschrieben, wie mit einer weiteren Methode dann alle Eigenwerte der Eingabematrix bestimmt werden können.

Abschnitt 5.2 wird sich mit der Berechnung der Eigenvektoren, oder genauer mit der Umsetzung der in Kapitel 4.2.3 beschriebenen Faktorisierung für die inverse Vektoriteration befassen.

Der zusammengesetzte Algorithmus aus den Teilen der Abschnitte 5.1 und 5.2 wird in Abschnitt 5.3 vorgestellt.

Wie der Algorithmus aus Kapitel 5.3 schließlich getestet wurde erläutert Abschnitt 5.4.

5.1 Eigenwerte

Die Berechnung der Eigenwerte erfolgt, wie bereits in Abschnitt 4.2.2 beschrieben, mit einer Kombination aus Bisektion und dem Newton-Verfahren, da die Methode, die in Abschnitt 4.1.2 zur Lösung der Säkulärgleichung vorgeschlagen wurde, aufgrund eines zu hohen Aufwandes nicht geeignet ist.

Zunächst wird in Abschnitt 5.1.1 beschrieben, wie die Bisektion und das Newton-Verfahren im Programm umgesetzt wurden. In Abschnitt 5.1.2 geht es um die Verwendung dieser Verfahren bei der Suche nach allen Nullstellen der Funktion r und in Abschnitt 5.1.3 wird schließlich beschrieben, wie die Eigenwerte bestimmt werden.

5.1.1 Bisektion und Newton-Verfahren

Die Bisektion wie auch das Newton-Verfahren werden hier nur dann auf einem Intervall angewendet, wenn zuvor sichergestellt ist, dass dort genau

eine Nullstelle existiert.

Die Bisektion ist als Funktion `bisection` wie folgt implementiert.

```
function bisection(function,l,r,var x)
    if (r - l > ε)
        t := ( r + l ) / 2;
        if (|function(t)| < δ)
            x := t;
        end;
        if (function(l) * function(t) < 0)
            x := bisection(function,l,t,x);
        else
            x := bisection(function,t,r,x);
        end;
    else
        x := ( r + l ) / 2;
    end;
    return x;
end;
```

Es werden eine Funktion `function`, von der die Nullstelle bestimmt werden soll und die linke Intervallgrenze `l` sowie die rechte Intervallgrenze `r` übergeben. Falls das Intervall schon klein genug ist, wird der Mittelpunkt davon als Ergebnis in `x` zurückgegeben. Andernfalls wird der Mittelpunkt berechnet, geprüft, in welcher Hälfte des Intervalls ein Vorzeichenwechsel vorliegt und auf diese dann erneut `bisection` angewendet. Dieser Vorgang wird wiederholt bis entweder der Funktionswert des berechneten Mittelpunktes nahe genug an der Null liegt (also betragsmäßig kleiner als ein $\delta > 0$ wird) oder die Größe des Intervalls schon in der gewünschten Genauigkeit liegt (also wenn die Länge des Intervalls kleiner als ein $\varepsilon > 0$ ist). Dabei kann die Schranke δ in Abhängigkeit von ε gewählt werden, zum Beispiel in der Form $\delta = f'(t) \cdot \varepsilon$. Wenn bei dieser Wahl die Eigenschaft der Bisektion, keine Ableitungsinformationen zu benötigen, erhalten bleiben soll, kann statt $f'(t)$ auch eine Approximation an die Ableitung verwendet werden.

Die Funktion `bisection` wird zur Bestimmung des Extremwertes der Funktion r in Intervallen, in denen r parabelförmig verläuft, benutzt indem `bisection` auf die Ableitung r' angewendet wird.

Obwohl das Newton-Verfahren mit seiner lokalen quadratischen Konvergenz dem Bisektions-Verfahren vorzuziehen wäre, ist dieses Vorgehen hier sinnvoll, da es die Berechnung der zweiten Ableitung r'', die für die Newton-Iteration nötig wäre, vermeidet.

Das Newton-Verfahren für die Funktion r auf einem bestimmten Intervall

ist als Funktion `newtonLR` implementiert und bekommt eine linke Intervallgrenze `left` und eine rechte Intervallgrenze `right` sowie einen Startwert `x` innerhalb dieses Intervalls übergeben.

Solange der Funktionswert `r(x)` nicht in der gewünschten Genauigkeit nahe genug an Null liegt, wird die Newton-Iteration durchgeführt, also das alte `x` mit `x - r(x)/r'(x)` überschrieben. Falls `r'(x)` dabei zu klein wird oder das `x` nicht (mehr) im Intervall liegt, wird vorzeitig mit einem Fehler abgebrochen und die Art des Fehlers in der Variable `error` gespeichert.

```
function newtonLR(var x,left,right,var error)
    error := ERR_OK;
    while (|r(x)| > ε)
        //falls die Ableitung nahe 0 ist, als fehlgeschlagen abbrechen
        if (|r'(x)| < ε)
            error := ERR_ZERODERI;
            break;
        end
        else
            //falls x außerhalb des Intervalls liegt, als fehlgeschlagen
            abbrechen
            if (x <= left or x >= right)
                error := ERR_OUTOFBOUNDS;
                break;
            end;
        end;
        x := x - r(x)/r'(x);
        end while;
    return x;
end;
```

Um zu vermeiden, dass die Newton-Iterierten aus dem Intervall herauslaufen, reicht es oft, den Startwert näher an die gesuchte Nullstelle heran zu bringen.

Dazu werden hier einzelne Bisektionsschritte verwendet.

Ein Bisektionsschritt ist in der Funktion `onebisection` implementiert. Diese bekommt als Eingabeparameter wieder die Funktion `function`, von der die Nullstelle bestimmt werden soll, eine linke Intervallgrenze `l` und eine rechte Intervallgrenze `r`. Es wird die Mitte des Intervalls berechnet, überprüft, in welcher Hälfte ein Vorzeichenwechsel auftritt und diese Intervallhälfte dann als neues Intervall zurückgegeben in Form einer neuen linken Intervallgrenze `lnew` und einer neuen rechten Intervallgrenze `rnew`.

Ferner wird noch der Mittelpunkt des neuen Intervalls zurückgegeben. Dieser kann dann als Startwert für das Newton-Verfahren verwendet werden, und falls `newtonLR` mit einem Fehler abbricht, kann erneut `onebisection` aufgerufen werden.

```
function onebisection(function,l,r,var lnew,var rnew,var x)
    t := (l + r) / 2;
    if (function(l) * function(t) < 0)
        lnew := l;
        rnew := t;
        x := (l + t) / 2;
    end
    else
        lnew := t;
        rnew := r;
        x := (t + r) / 2;
    end;
    return x;
end;
```

5.1.2 Nullstellen von r

Um die Nullstellen der Funktion r zu finden, wird jedes Intervall zwischen den Polstellen sowie die eingeschränkten Bereiche außerhalb der Polstellen einzeln betrachtet. Aus Lemma 17 ist bekannt, wie die Intervalle unterhalb der kleinsten und oberhalb der größten Polstelle zu wählen sind und dass dort jeweils genau eine Nullstelle zu finden ist.

```
procedure rLeftRoot(dmin,d[0],var lambda,var count)
    //Epsilon-Abstand zur ersten Polstelle wählen
    εdist := getEpsilonLeft(dmin,d[0]);
    //Newton mit d[0]-εdist auf dem Intervall (dmin,d[0])
    if (r(dmin) * r(d[0]-εdist) < 0)
        lambda[count] := newtonLR(d[0]-εdist,dmin,d[0],
                                  error);
        count := count + 1;
    end;
end;
```

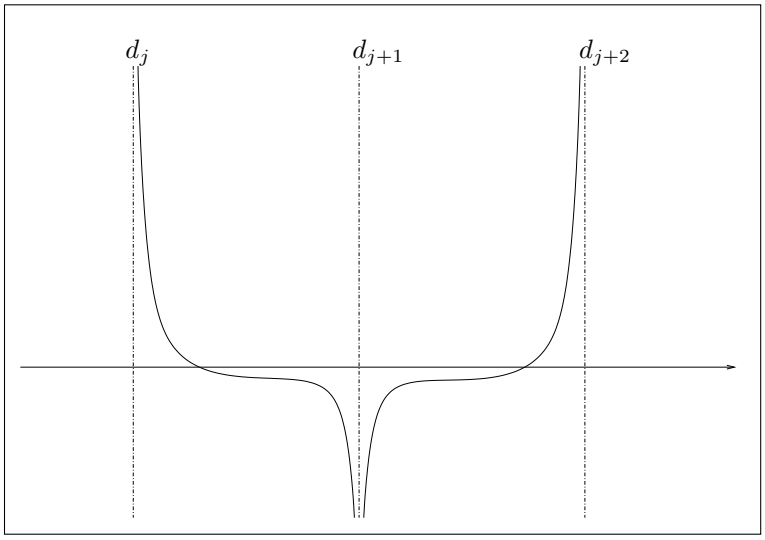
Die Prozedur `rLeftRoot` sucht nach der Nullstelle, die kleiner als die kleinste Polstelle ist. Dazu wird die kleinste Polstelle `d[0]` übergeben und ein Wert `dmin`, der gemäß der Ergebnisse aus Lemma 17 die untere Schranke für die kleinste Nullstelle von r angibt. Zunächst wird ein ε-Abstand ε`dist` von

der ersten Polstelle im Verhältnis zum Abstand von `dmin` und `d[0]` gewählt, um dann die Nullstelle mit Hilfe des Newton-Verfahrens auf dem Intervall `(dmin,d[0])` mit dem Startwert `d[0]` - ε`dist` zu berechnen. Das Ergebnis wird in ein Array `lambda` an die Stelle des Wertes des Nullstellenzählers `count` geschrieben, und danach wird `count` hochgezählt.

Die Überprüfung, ob es in dem Intervall überhaupt einen Vorzeichenwechsel gibt, ist nach Lemma 17 zwar nicht nötig, kann aber hilfreich sein, falls der ε-Abstand zur Polstelle nicht groß genug gewählt wurde.

```
procedure rRightRoot(dmax,d[m-1],var lambda,var count)

    //Epsilon-Abstand zur letzten Polstelle wählen
    εdist := getEpsilonRight(d[m-1],dmax);

    //Newton mit d[m-1]+εdist auf dem Intervall (d[m-1],dmax)
    if (r(dmax) * r(d[m-1]+εdist) < 0)
        lambda[count] := newtonLR(d[m-1]+εdist,dmax,d[m-1],
                                  error);
        count := count + 1;
    end;

end;
```

Die Suche nach der größten Nullstelle mit der Prozedur `rRightRoot` funktioniert sehr ähnlich. Es wird die größte Polstelle `d[m-1]` und die obere Schranke für die größte Nullstelle `dmax` übergeben, ein ε-Abstand ε`dist` zur Polstelle gewählt und dann das Newton-Verfahren mit dem Startwert `d[m-1]` + ε`dist` auf dem Intervall `(d[m-1],dmax)` angewendet.



Die Nullstellensuche zwischen zwei Polstellen ist etwas komplexer, da nach Satz 5 hier jeweils bis zu zwei Nullstellen in dem Intervall auftreten können. Genauer gibt es zwei Fälle zu unterscheiden: Zum Einen den Fall

mit genau einer Nullstelle in dem Intervall zwischen den Polstellen und zum Anderen den Fall mit zwei oder keinen Nullstellen im Intervall.

Der erste Fall wird mit der Prozedur `oneRoot` behandelt. Diese bekommt Werte nahe der linken und rechten Intervallgrenze sowie Referenzen auf das Array `lambda` und den Zähler `count` übergeben. Für die Suche nach genau einer Nullstelle können die Nullstellenverfahren aus Abschnitt 2.4 angewendet werden. Die Funktion r verläuft in diesem Fall zwischen den Polstellen wie in der Abbildung skizziert.

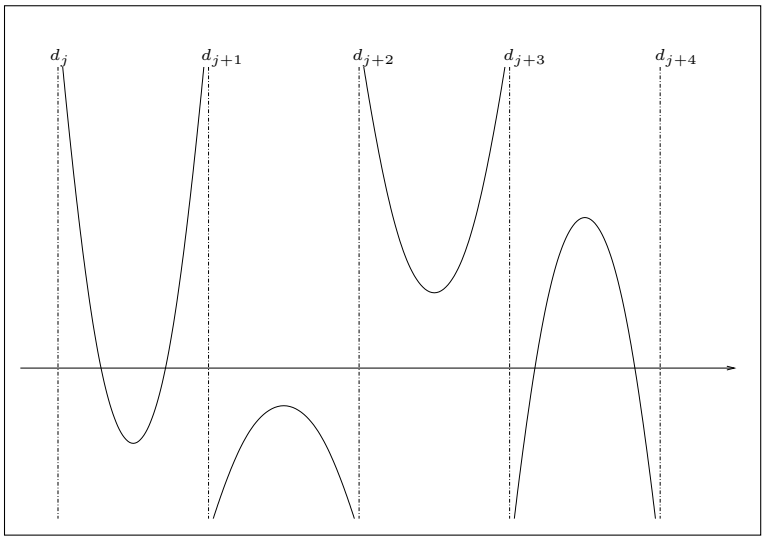
Wenn der Kurvenverlauf dabei sehr flach ist, kann das Newton-Verfahren schnell aus dem Intervall herausspringen. Daher wird hier wie bereits in Abschnitt 5.1.1 beschrieben, eine Kombination aus Bisektion und dem Newton-Verfahren verwendet.

Zunächst wird ein Bisektionsschritt auf dem Intervall mit `onebisection` durchgeführt, dann wird das Newton-Verfahren mit `newtonLR` auf die Hälfte des Intervalls angewandt, die der Bisektionsschritt liefert. Falls die Prozedur `newtonLR` mit einem Fehler abbricht, wird ein weiterer Bisektionsschritt ausgeführt, um danach auf dem erneut halbierten Intervall nochmals die Newton-Iteration zu starten. Solange `newtonLR` nicht erfolgreich ein Ergebnis liefert und das Intervall noch groß genug ist, wird dies wiederholt.

```
procedure oneRoot(dleft_ε,dright_ε,var lambda,var count)

    newl := dleft_ε;
    newr := dright_ε;

    //ein Bisektionsschritt und dann Newton
    l := onebisection(r,dleft_ε,dright_ε,newl,newr,l);
    lambda[count] := newtonLR(l,newl,newr,error);
    //Falls Newton aus dem Intervall springt, einen weiteren Bisek-
    tionsschritt, dann wieder Newton versuchen
    while ( error != ERR_OK )

        l := onebisection(r,newl,newr,newl,newr,l);

        if (|newl - newr| < ε)
            lambda[count] := newl;
            error := ERR_OK;
        end

        else
            lambda[count] :=newtonLR(l,newl,newr,error);
        end;

    end while;

    count := count + 1;

end;
```

In den meisten getesteten Fällen reichten allerdings sehr wenige Bisektionsschritte aus, damit das Newton-Verfahren gegen die gewünschte Null-

stelle konvergiert.



Der zweite Fall wird mit der Prozedur `twoRoots` behandelt. Hier werden wie im ersten Fall Werte nahe der linken und rechten Intervallgrenze, Referenzen auf das Array `lambda` und den Zähler `count` und zusätzlich die *echten* Intervallgrenzen als Schranken für das Newton-Verfahren übergeben.

```
procedure twoRoots(dleft_ε,dright_ε,dleft,dright,
                   var lambda,var count)

    //Bisektion mit r' liefert den Extremwert x von r
    x := bisection(r',dleft_ε,dright_ε,x);

    //Falls r(x) ein anderes Vorzeichen als r(dleft_ε) hat, wird Newton
    auf beide Intervallhälften angewendet.
    if (r(dleft_ε) * r(x) < 0)

        lambda[count] := newtonLR(dleft_ε,dleft,x,error);
        count := count + 1;
        lambda[count] :=newtonLR(dright_ε,x,dright,error);
        count := count + 1;

    end;

    //Falls r(x) das gleiche Vorzeichen wie r(dleft_ε) hat, gibt es keine
    Nullstelle im Intervall.
end;
```

In diesem Fall verläuft die Funktion r parabelförmig zwischen den Polstellen, wie in der Abbildung beispielhaft skizziert ist, und es gibt entweder zwei oder keine Nullstellen in dem Intervall. Um dies zu entscheiden, wird der Extremwert von r auf diesem Intervall berechnet, indem die Bisektion auf die Ableitung r' angewendet wird. Wenn der Funktionswert des Extremwertes das gleiche Vorzeichen hat wie `r(d[j]+`ε`dist)` und `r(d[j+1]-`ε`dist)`,

existiert keine Nullstelle auf dem Intervall. Wenn es sich aber davon unterscheidet, sind dort zwei Nullstellen zu finden.

Die eine wird mit dem Newton-Verfahren auf dem Intervall von `d[j]` bis zum Extremwert mit Startwert bei `d[j]` + ε`dist` (hier $\texttt{dleft}_\varepsilon$) gefunden und die andere mit dem Newton-Verfahren auf dem Intervall vom Extremwert bis zu `d[j+1]` mit dem Startwert `d[j+1]` - ε`dist` (hier $\texttt{dright}_\varepsilon$).

Die berechneten Nullstellen von `oneRoot` und `two Roots` werden jeweils in das Array `lambda` an die Stelle geschrieben, auf die der Zähler `count` weist, der danach jeweils hochgezählt wird.

```
procedure rBetweenRoots(d[j],d[j+1],var lambda,var count)

    //Epsilon-Abstand zu den Polstellen wählen
    εdist := getEpsilonBetween(d[j],d[j+1]);

    decision := CASE_UNKNOWN;

    while ( decision = CASE_UNKNOWN )

        //Fall 1: eine Nullstelle im Intervall
        if ( (r(d[j]+εdist) * r(d[j+1]-εdist) < 0)
            and (r'(d[j]+εdist) * r'(d[j+1]-εdist) > 0) )

            oneRoot(d[j]+εdist,d[j+1]-εdist,lambda,count);
            decision = CASE1;

        end

        else

            //Fall 2: zwei oder keine Nullstellen im Intervall
            if ((r(d[j]+εdist) * r(d[j+1]-εdist) > 0)
                and (r'(d[j]+εdist)*r'(d[j+1]-εdist) < 0))

                twoRoots(d[j]+εdist,d[j+1]-εdist,
                          d[j],d[j+1],lambda,count);
                decision = CASE2;

            end

            else

                //Fall 3: noch keine Entscheidung getroffen
                εdist := εdist * 0.125;

            end;

        end;

    end while;

end;
```

Insgesamt ist die Nullstellensuche zwischen zwei Polstellen mit der Prozedur `rBetweenRoots` realisiert. Dieser werden die beiden Polstellen übergeben, die das Intervall `(d[j],d[j+1])` eingrenzen, in dem gesucht werden

soll. Wie zuvor wird ein geeigneter Abstand ε`dist` zu den Polstellen gewählt, so dass die Funktion r an den Stellen `d[j]` + ε`dist` und `d[j+1]` - ε`dist` ohne Probleme auszuwerten ist.

Wenn die Funktionswerte dort verschiedene Vorzeichen haben und die Steigung in diesen Punkten das gleiche Vorzeichen aufweist, tritt der erste Fall ein, bei dem genau eine Nullstelle in dem Intervall existiert, die dann mit `oneRoot` berechnet wird. Wenn die Funktionswerte gleiche Vorzeichen haben und die Steigung in diesen Punkten ein unterschiedliches Vorzeichen, so tritt der zweite Fall ein, bei dem es entweder zwei oder keine Nullstellen in dem Intervall gibt, und es wird `twoRoots` aufgerufen. Falls keiner dieser Fälle zutrifft, muss der Abstand zu den Polstellen verkleinert werden, da Nullstellen von r sehr nahe an den Polstellen liegen können.

Dieses Vorgehen wird wiederholt, bis eine Entscheidung getroffen wurde, um welchen Fall es sich hier handelt. Die Gefahr dabei ist natürlich, dass der Abstand so klein werden kann, dass das Auswerten der Funktion so nahe einer Polstelle schwierig wird.

Die Prozedur, mit der alle Nullstellen von r berechnet werden, ist mit `rRoots` benannt und bekommt als Eingabe die Arrays `a`, `b`, `d1` und `d2` und das Array `lambda`, in das alle gefundenen Nullstellen geschrieben werden sollen, sowie einen Nullstellenzähler `count`.

```
procedure rRoots(a,b,d1,d2,var lambda,var count)
    //Polstellen von d1 und d2 ohne Duplikate ineinandersortieren in
    d (mit Länge m)
    d := merge(d1,d2,m);
    dmin := d[0] - ‖a‖‖b‖;
    dmax := d[m-1] + ‖a‖‖b‖;

    rLeftRoot(dmin,d[0],count);

    for i := 0 to m - 2 do
        rBetweenRoots(d[j],d[j+1],count);
    end for;

    rRightRoot(dmax,d[m-1],count);

end;
```

Die (sortierten) Arrays `d1` und `d2` werden mit der Methode `merge` in ein Array `d` sortiert. Dabei werden Werte, die mehrfach vorkommen, nur einmal aufgeführt, so dass die Einträge in `d` streng monoton steigend sortiert sind und jeweils die Intervallgrenzen der nach Nullstellen abzusuchenden Intervalle beinhalten. Die Länge $m \leq n$ des Arrays `d` wird in `m` gespeichert.

Als nächstes werden die untere Schranke für die kleinste Nullstelle `dmin` und die obere Schranke für die größte Nullstelle von r gemäß Lemma 17 berechnet. Dann wird mit `rLeftRoot` die Nullstelle zwischen `dmin` und dem ersten Eintrag in `d` berechnet und anschließend mit `rBetweenRoots` jeweils

in den Intervallen `(d[j],d[j+1])` nach Nullstellen gesucht. Zuletzt wird die Nullstelle zwischen dem letzten Eintrag von `d` und `dmax` mit `rRightRoot` berechnet.

Es wird immer eine Referenz auf den Zähler `count` übergeben, so dass dieser am Ende die Anzahl der gefundenen Nullstellen angibt.

5.1.3 Bestimmung der Eigenwerte

Mit der Prozedur `eigenValues` werden alle Eigenwerte einer Matrix A' der Form (4.11) berechnet.

Diese Prozedur bekommt als Eingabe die Arrays `a` und `b`, in denen die Vektoren a und b aus (4.10) gespeichert sind, sowie die Arrays `d1` und `d2`, in denen die $d_{1_1}, \ldots, d_{1_{\frac{n}{2}}}$ und $d_{2_1}, \ldots, d_{2_{\frac{n}{2}}}$ aus Kapitel 4.2 gespeichert sind. Außerdem wird noch ein vorinitialisiertes Array `lambda` der Länge `n` übergeben, in das dann alle gefundenen Eigenwerte geschrieben werden sollen.

Als erstes wird die Integer-Variable `count`, mit der die gefundenen Eigenwerte gezählt werden sollen, mit `0` initialisiert.

Dann wird der in Lemma 12 untersuchte Spezialfall behandelt, bei dem innerhalb der Diagonalmatrizen D_1 und D_2 mehrfache Eigenwerte vorkommen können. Hierzu werden die Arrays `d1` und `d2` auf Mehrfach-Einträge überprüft. Wenn ein solcher gefunden wird, ist dieser nach Lemma 12 bereits ein Eigenwert und wird in das Array `lambda` an die Stelle des Wertes von `count` geschrieben und `count` inkrementiert.

Ein weiterer Spezialfall tritt auf, falls in den Arrays `a` und `b` Nulleinträge auftreten. Nach Lemma 11 sind dann die zugehörigen Einträge von `d1` bzw. `d2` Eigenwerte, falls also `a[i] = 0` ist, wird `d1[i]` als gefundener Eigenwert in `lambda[count]` geschrieben und `count` hochgezählt. Analoges gilt im Fall `b[i] = 0` für `d2[i]`.

Nachdem diese Spezialfälle behandelt wurden, werden alle Nullstellen der Funktion r mit der Prozedur `rRoots` aus dem vorangegangenen Abschnitt 5.1.2 berechnet, die nach den Ergebnissen aus Kapitel 4.2 ebenfalls Eigenwerte von A' sind und damit in `lambda` gespeichert werden.

Falls nach den Spezialfällen und mit den Nullstellen von r noch immer nicht alle n Eigenwerte gefunden wurden, also der Zähler `count < n` ist, wird geprüft, ob die Einträge von `d1` und `d2` jeweils Polstellen von r sind. Nach Satz 4 setzt sich das Spektrum von A' aus der Menge der Nullstellen von r und der Menge derjenigen d_{i_j} zusammen, die eben keine Polstellen von r sind.

Falls die Funktion `isPole`, die einen Wert darauf überprüft, ob er eine Polstelle von r ist, bei einem `d1[i]` oder `d2[i]` also `false` liefert, ist dies ein weiterer Eigenwert von A'. Nach jedem so bestimmten Eigenwert wird die weitere Suche abgebrochen, falls `count = n` ist, also falls alle `n` Eigenwerte der $n \times n$-Matrix A' gefunden wurden.

```
procedure eigenValues(a,b,d1,d2,var lambda)

    count := 0;

    //gleiche Eigenwerte innerhalb einer Teilmatrix: Eigenwert der
    Matrix gefunden und weniger Intervalle abzusuchen
    for i := 0 to n/2 - 1 do

        if ((d1[i+1] - d1[i]) < ε)
            lambda[count] := d1[i];   count := count + 1;
        end;

        if ((d2[i+1] - d2[i]) < ε)
            lambda[count] := d2[i];   count := count + 1;
        end;

    end for;

    //falls a oder b Nulleinträge haben, zerfällt das Problem weiter
    und die zugehörigen d[i] sind Eigenwerte
    for i := 0 to n/2 - 1 do

        if (|a[i]| < ε)
            lambda[count] := d1[i];   count := count + 1;
        end;

        if (|b[i]| < ε)
            lambda[count] := d2[i];   count := count + 1;
        end;

    end for;

    rRoots(a,b,d1,d2,lambda,count);

    //falls nicht genug Eigenwerte gefunden, d1 und d2 prüfen
    if (count < n)

        for i := 0 to n/2 - 1 do
            if (not isPole(d1[i]))
                lambda[count] := d1[i];
                count := count + 1;
                if (count = n) break;
            end;

            if (not isPole(d2[i]))
                lambda[count] := d2[i];
                count := count + 1;
                if (count = n) break;
            end;

        end for;

    end;

    sort(lambda);

end;
```

Da nur die durch `rRoots` berechneten Eigenwerte sortiert in `lambda` gespeichert sind, die durch die Spezialfälle bestimmten jedoch dazwischen liegen, wird auf das Array `lambda` zum Schluss noch ein Sortieralgorithmus `sort` angewandt.

5.2 Eigenvektoren

Ein Eigenvektor zu einem der bereits bekannten Eigenwerte wird mit der inversen Vektoriteration mit einem Shiftparameter nahe des Eigenwertes berechnet. Für das Lösen des Gleichungssystems in jedem Iterationsschritt wird, wie in Abschnitt 4.2.3 beschrieben, die Struktur der Matrix ausgenutzt. Mit der Faktorisierung $A' = LR$ aus Lemma 19 sind in jedem Iterationsschritt ein unteres linkes Gleichungssystem und ein oberes rechtes Block-Gleichungssystem zu lösen.

5.2.1 Lösen des unteren Dreieckssystems

Das Lösen des unteren linken Dreieckssystems $Ly = x^{(m-1)}$ oder genauer von

$$\begin{bmatrix} I & 0 \\ ba^T(D_1 - \widetilde{\lambda}I)^{-1} & I \end{bmatrix} y = x^{(m-1)} \tag{5.1}$$

ist mit der Prozedur `solveLower` realisiert. Diese bekommt die Dimension `n`, die Arrays `a`, `b` und `d1` der Länge $\frac{n}{2}$ übergeben, sowie das Array `x` der Länge n, das die alte Iterierte beinhaltet und das vorinitialisierte Array `y`, ebenfalls der Länge n, in das das Ergebnis dieses Gleichungssystems geschrieben werden soll.

Da sich die obere Hälfte des Gleichungssystems (5.1) auf

$$\begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix} \begin{pmatrix} y_0 \\ \vdots \\ y_{\frac{n}{2}-1} \end{pmatrix} = \begin{pmatrix} x_0^{(m-1)} \\ \vdots \\ x_{\frac{n}{2}-1}^{(m-1)} \end{pmatrix}$$

reduziert, ist hier nichts zu berechnen und die erste Hälfte des Lösungsarrays `y` wird mit den Einträgen der ersten Hälfte der alten Iterierten `x` gefüllt, also `y[i] := x[i]` für $i < \frac{n}{2}$.

Die untere Hälfte des Gleichungssystems (5.1) ist gegeben durch

$$ba^T(D_1 - \widetilde{\lambda}I)^{-1} \begin{pmatrix} y_0 \\ \vdots \\ y_{\frac{n}{2}-1} \end{pmatrix} + \begin{pmatrix} y_{\frac{n}{2}} \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} x_{\frac{n}{2}}^{(m-1)} \\ \vdots \\ x_{n-1}^{(m-1)} \end{pmatrix}$$

und es folgt

$$\begin{pmatrix} b_0 \\ \vdots \\ b_{\frac{n}{2}-1} \end{pmatrix} \left(a_0, \ldots, a_{\frac{n}{2}-1}\right) \begin{pmatrix} \frac{y_0}{d_{1_0}-\widetilde{\lambda}} \\ \vdots \\ \frac{y_{\frac{n}{2}-1}}{d_{1_{\frac{n}{2}-1}}-\widetilde{\lambda}} \end{pmatrix} + \begin{pmatrix} y_{\frac{n}{2}} \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} x_{\frac{n}{2}}^{(m-1)} \\ \vdots \\ x_{n-1}^{(m-1)} \end{pmatrix}$$

Da die $y_0, \ldots, y_{\frac{n}{2}-1}$ bereits aus der Lösung der oberen Hälfte des Gleichungssystems bekannt sind, lassen sich die $y_{\frac{n}{2}}, \ldots, y_{n-1}$ berechnen durch

$$\begin{pmatrix} y_{\frac{n}{2}} \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} x_{\frac{n}{2}}^{(m-1)} \\ \vdots \\ x_{n-1}^{(m-1)} \end{pmatrix} - \begin{pmatrix} b_0 \\ \vdots \\ b_{\frac{n}{2}-1} \end{pmatrix} \cdot \left(\sum_{j=0}^{\frac{n}{2}-1} \frac{a_j y_j}{d_{1_j} - \widetilde{\lambda}} \right)$$

also

$$y_{\frac{n}{2}+i} \;=\; x_{\frac{n}{2}+i}^{(m-1)} - b_i \sum_{j=0}^{\frac{n}{2}-1} \frac{a_j}{d_{1_j} - \widetilde{\lambda}} \cdot y_j$$

für $i = 0, \ldots, \frac{n}{2} - 1$. Da die Summe dabei nicht von i abhängt, kann diese außerhalb der For-Schleife über i berechnet werden.

```
procedure solveLower(x,n,a,b,d1,var y)
    for i := 0 to n/2 - 1 do
        y[i] := x[i];
    end for;
    temp := sum_{j=0}^{n/2-1} a[j] / (d1[j] - shift) * y[j];
    for i := 0 to n/2 - 1 do
        y[n/2 + i] := x[n/2 + i] - (b[i] * temp);
    end for;
end;
```

5.2.2 Lösen des oberen Block-Gleichungssystems

Das obere rechte Gleichungssystem $Rx^{(m)} = y$ oder genauer

$$\begin{bmatrix} D_1 - \widetilde{\lambda} I & ab^T \\ 0 & (D_2 - \widetilde{\lambda} I) - ba^T (D_1 - \widetilde{\lambda} I)^{-1} ab^T \end{bmatrix} x^{(m)} \;=\; y$$

wird mit der Prozedur `solveUpper` gelöst. Als Eingabe bekommt diese die Dimension `n`, die Arrays `a`, `b`, `d1`, `d2` und das zuvor mit `solveLower` berechnete Array `y` sowie die alte Iterierte `x`, die dann mit der neuen überschrieben werden wird. Für die Anwendung der Sherman-Morrison-Formel

werden zusätzlich noch die Werte `alpha` und `beta` übergeben, die α und β aus (4.26) entsprechen, sowie ein Wert `gamma`, der die Zahl

$$\gamma := \frac{\alpha}{1 - \alpha\beta} \tag{5.2}$$

enthält.

Zunächst muss die untere Hälfte des Block-Gleichungssystems gelöst werden, da die Lösung davon zur Berechnung der oberen Hälfte benötigt wird. Die untere Hälfte ist ein vollbesetztes Gleichungssystem, das direkt gelöst werden soll:

$$\begin{pmatrix} x_{\frac{n}{2}}^{(m)} \\ \vdots \\ x_{n-1}^{(m)} \end{pmatrix} = ((D_2 - \widetilde{\lambda} I) - ba^T (D_1 - \widetilde{\lambda} I)^{-1} ab^T)^{-1} \begin{pmatrix} y_{\frac{n}{2}} \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Dazu wird, wie in Abschnitt 4.2.3 beschrieben, die Sherman-Morrison-Formel (4.25) verwendet, um die Inverse zu berechnen. Mit Formel (4.27) und γ aus (5.2) gilt

$$\begin{pmatrix} x_{\frac{n}{2}}^{(m)} \\ \vdots \\ x_{n-1}^{(m)} \end{pmatrix} = \left((D_2 - \widetilde{\lambda} I)^{-1} + \gamma (D_2 - \widetilde{\lambda} I)^{-1} b b^T (D_2 - \widetilde{\lambda} I)^{-1} \right) \begin{pmatrix} y_{\frac{n}{2}} \\ \vdots \\ y_{n-1} \end{pmatrix}$$

und damit folgt für $i = 0, \ldots, \frac{n}{2} - 1$

$$x_{\frac{n}{2}+i}^{(m)} := \frac{y_{\frac{n}{2}+i}}{d_{2_i} - \widetilde{\lambda}} + \gamma \cdot \frac{b_i}{d_{2_i} - \widetilde{\lambda}} \cdot \sum_{j=0}^{\frac{n}{2}-1} \frac{b_j}{d_{2_j} - \widetilde{\lambda}} \cdot y_{\frac{n}{2}+j}.$$

Hier ist die Summe nicht von i abhängig und kann daher außerhalb der For-Schleife über i berechnet werden.

Mit den dadurch berechneten $x_{\frac{n}{2}}^{(m)}, \ldots, x_{n-1}^{(m)}$ kann nun die obere Hälfte

$$\begin{bmatrix} d_{1_0} - \widetilde{\lambda} & & 0 & a_0 b_0 & \ldots & a_0 b_{\frac{n}{2}-1} \\ & \ddots & & \vdots & & \vdots \\ 0 & & d_{1_{\frac{n}{2}-1}} - \widetilde{\lambda} & a_{\frac{n}{2}-1} b_0 & \ldots & a_{\frac{n}{2}-1} b_{\frac{n}{2}-1} \end{bmatrix} \begin{pmatrix} x_0^{(m)} \\ \vdots \\ x_{n-1}^{(m)} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

des Gleichungssystems berechnet und damit die $x_0^{(m)}, \ldots, x_{\frac{n}{2}-1}^{(m)}$ bestimmt werden. Dazu ist

$$(d_{1_i} - \widetilde{\lambda}) x_i^{(m)} + \sum_{j=0}^{\frac{n}{2}-1} a_i \, b_j \, x_{\frac{n}{2}+j} = y_i$$

für $i = 0, \ldots, \frac{n}{2} - 1$ zu lösen, also ist die Lösung durch

$$x_i^{(m)} := \frac{y_i - a_i \cdot \sum_{j=0}^{\frac{n}{2}-1} b_j \, x_{\frac{n}{2}+j}}{d_{1_i} - \widetilde{\lambda}}$$

gegeben. Hier ist wie oben die Summe nicht von i abhängig und kann auch außerhalb der For-Schleife über i berechnet werden.

Der so berechnete Vektor x wird in das Array `x` geschrieben und stellt die neue Iterierte der Vektoriteration dar.

```
procedure solveUpper(y,alpha,beta,gamma,n,a,b,d1,d2,var x)
    //mit Sherman Morrison die untere Hälfte direkt berechnen
    temp := sum_{j=0}^{n/2-1} b[j] * y[n/2 + j] / d2[j] - shift;
    for i := 0 to n/2 - 1 do
        x[n/2 + i] := y[n/2 + i]/(d2[i]-shift)
                      + gamma*((b[i]/(d2[i]-shift))*temp);
    end for;
    //obere Hälfte des oberen Dreieckssystems berechnen
    temp := sum_{j=0}^{n/2-1} b[j] * x[n/2 + j];
    for i := 0 to  n/2 - 1 do
        x[i] := (y[i] - a[i]*temp) / (d1[i] - shift);
    end for;
end;
```

5.2.3 Inverse Vektoriteration

Die vollständige Berechnung eines Eigenvektors der Matrix A der Form (4.8) zu einem schon bekannten Eigenwert wird mit der Prozedur `eigenVector` durchgeführt, welche die inverse Vektoriteration zunächst auf die Matrix A' aus (4.11) anwendet und dann den Ergebnisvektor zu einem Eigenvektor von A transformiert.

Dazu wird die Dimension `n` von A, die Arrays `a`, `b` und die Arrays, die die Eigenwerte der Teilprobleme enthalten, `d1` und `d2` übergeben, sowie die Arrays `U1` und `U2`, die die Matrizen aus (4.9) mit den Eigenvektoren der Teilprobleme enthalten. Als weitere Eingabe erhält die Prozedur einen Shiftparameter, für den dann später der um einen Epsilon-Wert gestörte Eigenwert, zu dem der Eigenvektor berechnet werden soll, eingesetzt wird und den zugehörigen Zähler, der angibt an welcher Stelle im sortierten Array der Eigenwerte dieser Eigenwert steht. Auf diese Weise kann zum Schluss der

berechnete Eigenvektor an die *richtige Stelle* im Array `E`, in dem die Matrix aus allen Eigenvektoren gespeichert werden soll, geschrieben werden.

Als Startvektor für die inverse Vektoriteration wird hier einfach ein normierter Vektor der Form

$$x := \left.\begin{pmatrix} \frac{1}{\sqrt{n}} \\ \vdots \\ \frac{1}{\sqrt{n}} \end{pmatrix}\right\} n$$

gewählt und als alte Iterierte der erste Einheitsvektor.

```
procedure eigenVector(shift,count,n,a,b,d1,d2,U1,U2,var E)

    //Startvektor mit Norm 1
    x := [1/sqrt(n),...,1/sqrt(n)];
    x_old := [1,0,...,0];

    //Alpha und Beta für Sherman Morrison sind immer fest

    alpha := Σ_{i=0}^{n/2-1} a[i] * a[i] / (d1[i] - shift);

    beta := Σ_{i=0}^{n/2-1} b[i] * b[i] / (d2[i] - shift);

    gamma := alpha / (1 - alpha*beta);

    //Abbruch, wenn der Winkel zwischen der aktuellen und vorherigen Iterierten klein genug ist
    while (|⟨ x , x_old ⟩| < 1 - ε)

        x_old := x;

        //Löse L y = x^(m-1) (unteres Dreieckssystem)
        solveLower(x_old,n,a,b,d1,y);

        //Löse R x^(m) = y (oberes Block-Gleichungssystem)
        solveUpper(y,alpha,beta,gamma,n,a,b,d1,d2,x);

        x := x / ‖x‖;

    end while;

    //Berechne den Eigenvektor zu A durch v = U x
    for i := 0 to n/2 - 1 do
        v[i] := U1 * x[i];
        v[n/2 + i] := U2 * x[n/2 + i];
    end for;

    //in die Ergebnismatrix E schreiben
    for i := 0 to n - 1 do
        E[count * n + i] := v[i];
    end for;

end;
```

α und β aus (4.26) und γ aus (5.2) für die Sherman-Morrison-Formel

sind unabhängig von der Iterierten `x` und daher können die Arrays `alpha`, `beta` und `gamma` vor der eigentlichen Iteration berechnet werden.

Die inverse Vektoriteration für A' wird solange durchgeführt, bis der Winkel zwischen der aktuellen Iterierten `x` und der vorherigen Iterierten $\mathtt{x}_{\mathtt{old}}$ in der gewünschten Genauigkeit klein genug ist, das heißt solange bis

$$|\cos \sphericalangle(\mathtt{x}, \mathtt{x}_{\mathrm{old}})| \;=\; \frac{|\langle \mathtt{x}, \mathtt{x}_{\mathrm{old}} \rangle|}{\|\mathtt{x}\| \|\mathtt{x}_{\mathrm{old}}\|} \;=\; |\langle \mathtt{x}, \mathtt{x}_{\mathrm{old}} \rangle| \;\geq\; 1 - \varepsilon$$

ist. Dazu wird zunächst das Array $\mathtt{x}_{\mathtt{old}}$, das die alte Iterierte enthält, mit der aktuellen gefüllt. Dann wird die Iteration in zwei Schritten durchgeführt. Zuerst berechnet die Prozedur `solveLower` aus Abschnitt 5.2.1 das untere Dreieckssystem $Ly = x^{(m-1)}$ und dann wird mit `solveUpper` das obere Block-Gleichungssystem $Rx^{(m)} = y$ gelöst. Am Ende jedes Iterationsschrittes wird `x` normiert.

Da die inverse Vektoriteration hier aufgrund der gut ausnutzbaren Form auf die Matrix A' angewandt wurde, ist mit `x` auch ein Eigenvektor zu A' berechnet worden. Da A' aber durch eine unitäre Ähnlichkeitstransformation mit U aus (4.9) aus der Matrix A hervorgegangen ist, kann der gewünschte Eigenvektor v von A durch

$$v \;=\; U\,x \;=\; \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} \;=\; \begin{pmatrix} U_1 \begin{pmatrix} x_0 \\ \vdots \\ x_{\frac{n}{2}-1} \end{pmatrix} \\ U_2 \begin{pmatrix} x_{\frac{n}{2}} \\ \vdots \\ x_{n-1} \end{pmatrix} \end{pmatrix}$$

berechnet werden.

Schließlich wird das so erhaltene Array `v` mit dem Eigenvektor v von A in das Array `E`, in dem die Matrix aller Eigenvektoren von A gespeichert werden soll, an die durch den Zähler `count` bestimmte Stelle geschrieben.

5.3 Gesamtalgorithmus

Dieses Kapitel stellt den Algorithmus zur Lösung eines Eigenwertproblems von $\mathcal{H}_l(1)$-Matrizen vor, der die in den vorangegangenen Abschnitten 5.1 und 5.2 vorgestellten Prozeduren benutzt.

Abschnitt 5.3.1 erklärt die Berechnung der Vektoren a und b aus (4.10) und Abschnitt 5.3.2 zeigt den Gesamtalgorithmus.

5.3.1 Berechnung von a und b durch Kreuzapproximation

Zu einer Eingabematrix A der Form

$$A \;=\; \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix}$$

sollen die Vektoren $\hat{a}$ und $\hat{b}$ mit $R = \hat{a}\,\hat{b}^{\,T}$ berechnet werden.

Dazu wird die Prozedur `crossApproximation` benutzt. Diese bekommt das Array `R`, in dem die Matrix R gespeichert ist, und die Dimension `dim` von R übergeben.

Zunächst wird ein Matrixeintrag $R_{ij} \neq 0$ gesucht, am besten eignet sich der betragsgrößte Eintrag

$$R_{ij} \;:=\; \max_{l,k\,=\,0,\ldots,\texttt{dim}} |R_{lk}|\,,$$

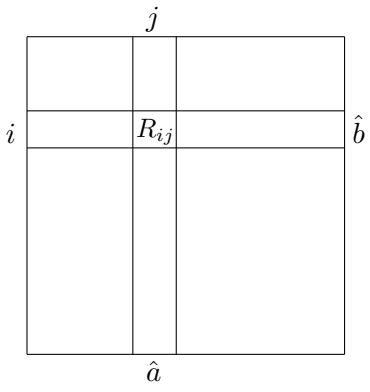
den die Prozedur `getMaxEntry` liefert und der in `Rij` gespeichert wird.

Es ist

$$R_{ij} \;=\; \hat{a}_i\,\hat{b}_j$$

zu lösen. Da R eine Rang-1-Matrix ist, kann dazu einfach $\hat{b}_j$ auf 1 gesetzt werden, so dass damit alle Einträge von $\hat{a}$ mit der j-ten Spalte bestimmt werden können durch $\hat{a}_k = R_{kj}$.

Damit können dann auch mit $\hat{a}_i\,\hat{b}_k \;=\; R_{ik}$ die restlichen Einträge von $\hat{b}$ bestimmt werden. Das Schema für diese Berechnung lässt sich verdeutlichen anhand der folgenden Grafik.



Es wird $\hat{a}$ auf die j-te Spalte von R gesetzt mit `a[k] := R[k][j]` für alle `k` und $\hat{b}$ auf die i-te Zeile von R dividiert durch R_{ij}, also `b[k] := R[i][k]/Rij` für alle `k`.

```
procedure crossApproximation(dim,R,var a,var b)
    //suche betragsgrößten Eintrag Rij = R_ij ≠ 0 von R
    Rij := getMaxEntry(R);
    for k := 0 to dim - 1 do
        a[k] := R[k][j];
        b[k] := R[i][k] / Rij;
    end for;
end;
```

5.3.2 Algorithmus zum Eigenwertproblem

Der Algorithmus als Ganzes wird mit der Prozedur `eigenProblem` durchgeführt. Als Eingabe erhält die Prozedur eine $\mathcal{H}_l(1)$-Matrix `A` der Form

$$A = \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix}$$

und deren Dimension `n`, die nach Definition der $\mathcal{H}_l$-Matrixklasse eine Zweierpotenz sein muss. Es wird davon ausgegangen, dass bereits vor Aufruf von `eigenProblem` überprüft wurde, ob `A` wirklich eine $\mathcal{H}_l(1)$-Matrix mit Zweierpotenzdimension ist.

Am Ende werden die berechneten Eigenwerte aufsteigend sortiert in einem Array `lambda` zurückgegeben und die zugehörigen Eigenvektoren, nach Eigenwerten sortiert, spaltenweise in einem Matrix-Array `E`.

Wie in Kapitel 4.2.1 beschrieben, wird, wenn die Dimension `n` klein genug ist (in diesem Fall wurde dazu die Grenze `n == 2` gewählt), auf die üblichen Verfahren, wie zum Beispiel das QR-Verfahren, zurückgegriffen.

Falls `n` größer ist, wird `eigenProblem` nach dem Divide-and-Conquer-Prinzip rekursiv für die Teilmatrizen `A1` und `A2` aufgerufen.

Die Eigenwerte von `A1` werden dabei im Array `d1` und die Eigenvektoren im Array `U1` gespeichert, die Eigenwerte von `A2` in `d2` und die Eigenvektoren in `U2`.

Um die Lösungen der Teilprobleme zur Lösung des Eigenwertproblems von `A` zusammenfügen zu können, werden zunächst die Vektoren $\hat{a}$ und $\hat{b}$ mit $R = \hat{a}\,\hat{b}^T$ durch die Prozedur `crossApproximation` aus dem vorangegangenen Abschnitt 5.3.1 berechnet und in die Arrays `â` und `b̂` geschrieben.

Da auf A in Kapitel 4.2.1 eine Ähnlichkeitstransformation $U^T A U = A'$ mit

$$U = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix}$$

angewendet wurde, müssen auch $\hat{a}$ und $\hat{b}$ mit U_1 bzw. U_2 transformiert werden, so dass mit `a := U1`^{T} `* â` und `b := U2`^{T} `* b̂` alle benötigten Werte

vorliegen, um mit der Funktion r, die die Prozedur `eigenValues` ausnutzt, rechnen zu können.

```
procedure eigenProblem(n,A,var lambda,var E)

    if (n == 2)
        //berechne Eigenwerte und Eigenvektoren mit einem der
        üblichen Verfahren
        2x2eigenProblem(A);
    end

    else
        //Rekursion
        eigenProblem(n/2,A1,d1,U1);
        eigenProblem(n/2,A2,d2,U2);

        //a und b berechnen
        crossApproximation(n/2,R,â,b̂);
        a := U1^T * â;
        b := U2^T * b̂;

        //Eigenwerte berechnen
        eigenValues(a,b,d1,d2,lambda);

        //Eigenvektoren berechnen
        for i := 0 to n - 1 do
            shift := lambda + getShiftEpsilon;
            eigenVector(shift,count,n,a,b,d1,d2,U1,U2,E);
        end for;

    end;

end;
```

Die mit `eigenValues` berechneten Eigenwerte werden im Array `lambda` gespeichert. Dieses wird dann durchlaufen, um zu jedem Eigenwert mit der Prozedur `eigenVector` einen Eigenvektor zu berechnen. Hierfür wird zunächst ein geeigneter Shiftparameter für die inverse Vektoriteration nahe des Eigenwertes berechnet, indem auf den Eigenwert ein Epsilon addiert wird. Dieses wird von der Methode `getShiftEpsilon` geliefert. Diese berechnet das Epsilon in Abhängigkeit des Abstandes vom aktuell betrachteten Eigenwert zum nächsten. Durch `eigenVector` werden die Eigenvektoren im Array `E` gespeichert.

Damit ist durch `lambda` und `E` die Lösung des Eigenwertproblems von `A` gegeben.

5.4 Aufwandsbetrachtungen

In diesem Abschnitt soll die Komplexität des in den vorangegangenen Abschnitten beschriebenen Algorithmus betrachtet werden. Dazu wird in Abschnitt 5.4.1 zunächst der Aufwand der in Kapitel 5.1 beschriebenen Teile

des Algorithmus betrachtet, die zur Eigenwertberechnung nötig sind, und danach in Abschnitt 5.4.2 der Aufwand der in Kapitel 5.2 vorgestellten Teile, die zur Eigenvektorberechnung notwendig sind.

In Abschnitt 5.4.3 folgt dann die Komplexitätsbetrachtung des gesamten Algorithmus, die in Abschnitt 5.5 mit einem Testbeispiel aus der Praxis illustriert wird.

5.4.1 Aufwand der Eigenwertberechnung

In diesem Abschnitt wird der Aufwand der in Kapitel 5.1 beschriebenen Prozeduren betrachtet.

Bemerkung 7 *Die Auswertung der Funktion r benötigt $4n+3$ Rechenoperationen und die Auswertung von r' braucht $9n+3$ Rechenoperationen.*

Begründung: Die Auswertung von

$$r(x) \;=\; \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - x}\right)\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - x}\right) - 1$$

erfordert für jede Summe $\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1) = 2n - 1$ Rechenoperationen, plus eine für die Initialisierung der Summe, eine weitere Multiplikation und eine Addition, also insgesamt $2 \cdot (2n - 1 + 1) + 2 = 4n + 2$ Rechenoperationen.

Die Auswertung von

$$r'(x) = \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{(d_{1_i} - x)^2}\right)\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{d_{2_j} - x}\right) + \left(\sum_{i=1}^{\frac{n}{2}} \frac{a_i^2}{d_{1_i} - x}\right)\left(\sum_{j=1}^{\frac{n}{2}} \frac{b_j^2}{(d_{2_j} - x)^2}\right)$$

erfordert $2 \cdot (\frac{n}{2} \cdot 4 + (\frac{n}{2} - 1) + \frac{n}{2} \cdot 3 + (\frac{n}{2} - 1)) = 9n - 4$ Operationen für die Berechnung aller Summen, plus eine je Summe für die Initialisierung, also $9n$ und 2 weitere Multiplikationen und eine Addition, so dass die Auswertung von r' insgesamt $9n + 3$ Rechenoperationen benötigt. □

Die in Abschnitt 5.1.1 vorgestellten Funktionen zur Bisektion und zum Newton-Verfahren, angewandt auf die Funktion r beziehungsweise auf ihre Ableitung r', haben einen Aufwand, der in $\mathcal{O}(n)$ liegt.

Bemerkung 8 *Die Funktionen* `onebisection` *und* `newtonLR` *angewandt auf r und* `bisection` *angewandt auf r' haben einen Aufwand in $\mathcal{O}(n)$.*

Begründung: Mit Bemerkung 7 gilt für die einzelnen Funktionen:

- `onebisection`(angewandt auf r): Es fallen 2 Mittelpunktsberechnungen (je 2 Rechenoperationen) und 2 Funktionsauswertungen von r an. Mit insgsamt $2 \cdot 2 + 2 \cdot (4n + 2) = 8n + 8$ Rechenoperationen liegt der Aufwand für `onebisection` in $\mathcal{O}(n)$.

- `bisection`(angewandt auf r'): Je Iterationsschritt fällt eine neue Auswertung von r' und eine Mittelpunktsberechnung (2 Operationen) an, also $9n + 3 + 2 = 9n + 5$ Rechenoperationen.

  Da die Anzahl der Iterationsschritte abzuschätzen ist durch $\log_2 \frac{l-r}{\varepsilon}$ (für Intervallgrenzen l und r und Abbruchgenauigkeit ε), liegt der Aufwand für `bisection` in $\mathcal{O}(n)$.

- `newtonLR`(angewandt auf r): Je Iterationsschritt fallen eine Funktionsauswertung und eine Ableitungsauswertung sowie eine Division und eine Subtraktion an, also $4n + 2 + 9n + 3 + 1 + 1 = 13n + 7$ Rechenoperationen.

  Da das Newton-Verfahren nahe einer Nullstelle aufgerufen wird und quadratisch konvergiert, sind nur sehr wenige Iterationsschritte nötig, so dass der Gesamtaufwand für `newtonLR` in $\mathcal{O}(n)$ liegt.

□

Damit kann nun der Aufwand der Prozeduren aus Abschnitt 5.1.2 bestimmt werden. Der Aufwand der Suche nach Nullstellen von r in den einzelnen Intervallen zwischen den Polstellen und links der kleinsten Polstelle und rechts der größten Polstelle liegt jeweils in $\mathcal{O}(n)$.

Bemerkung 9 *Die Prozeduren* `rLeftRoot`*,* `rBetweenRoots` *und* `rRightRoots` *haben jeweils einen Aufwand in* $\mathcal{O}(n)$.

Begründung:

- `rLeftRoot`: Es fällt eine konstante Anzahl von Operationen für `getEpsilonLeft`, also zur Bestimmung des ε-Abstandes, und eine Operation zur Inkrementierung des Nullstellenzählers (also bis hier in $\mathcal{O}(1)$) an, sowie 2 Auswertungen von r (nach Bemerkung 7 in $\mathcal{O}(n)$) und ein Aufruf von `newtonLR` (nach Bemerkung 8 in $\mathcal{O}(n)$). Damit hat `rLeftRoot` auch einen Aufwand in $\mathcal{O}(n)$.

- `rRightRoot`: Analog zu `rLeftRoot`.

- `rBetweenRoots`: Es fällt eine konstante Anzahl von Operationen für `getEpsilonBetween`, also für den ε-Abstand an (damit in $\mathcal{O}(1)$), 2 Auswertungen von r und 2 Auswertungen von r' (nach Bemerkung 8 jeweils in $\mathcal{O}(n)$) an, sowie ein Aufruf von `oneRoot` oder `twoRoots`. Die

While-Schleife wird in der Regel nur einmal durchlaufen, im schlechtesten Fall bis ε`dist` unter Maschinengenauigkeit fällt.

`oneRoot`: Es fällt je ein Aufruf von `onebisection` und von `newtonLR` an (beide nach Bemerkung 8 in $\mathcal{O}(n)$). Für den Fall, dass `newtonLR` mit einem Fehler abbricht, wird `onebisection` erneut aufgerufen, bis `newtonLR` erfolgreich ist. Im schlechtesten Fall addiert sich hier der Aufwand von `newtonLR` und `bisection`. Damit liegt `oneRoot` wiederum in $\mathcal{O}(n)$.

`twoRoots`: Es fällt ein Aufruf von `bisection` für r' zur Bestimmung des Extremwertes von r an, sowie 2 Funktionsauswertungen von r und im schlechtesten Fall 2 Aufrufe von `newtonLR` und 2 Rechenoperationen zur Inkrementierung des Nullstellenzählers. Insgesamt liegt `twoRoots` damit aber auch in $\mathcal{O}(n)$.

Damit hat `rBetweenRoots` einen Gesamtaufwand in $\mathcal{O}(n)$.

□

Daraus folgt ein Aufwand von $\mathcal{O}(n^2)$ für die Suche nach allen Nullstellen von r.

Bemerkung 10 *Die Prozedur* `rRoots` *hat einen Aufwand in* $\mathcal{O}(n^2)$.

Begründung: Es wird zunächst `merge` mit `d1` und `d2` aufgerufen. `merge` nutzt aus, dass diese Arrays (jeweils der Länge $\frac{n}{2}$) bereits sortiert sind, durchläuft daher diese beim Zusammensortieren nur einmal und hat damit einen Aufwand in $\mathcal{O}(n)$.

Die Berechnung von `dmin` und `dmax` benötigt eine Normberechnung für je `a` und `b` (Arrays der Länge $\frac{n}{2}$, also jeweils n Rechenoperationen), und jeweils eine Multiplikation und eine Addition bzw. Substraktion, also für beide zusammen $n + 4$ Operationen.

Es fällt je ein Aufruf von `rLeftRoot` und `rRightRoot` (nach Bemerkung 9 jeweils in $\mathcal{O}(n)$) an und bis zu n Aufrufe von `rBetweenRoots` (nach Bemerkung 9 auch jeweils in $\mathcal{O}(n)$, also $n \cdot \mathcal{O}(n)$).

Damit liegt der Gesamtaufwand für `rRoots` in $\mathcal{O}(n^2)$. □

Der Gesamtaufwand der Prozedur `eigenValues` aus Abschnitt 5.1.3 zur Bestimmung aller Eigenwerte liegt damit in $\mathcal{O}(n^2)$.

Bemerkung 11 *Die Prozedur* `eigenValues` *hat einen Aufwand in* $\mathcal{O}(n^2)$.

Begründung: Zur Überprüfung der Spezialfälle aus Lemma 11 und 12 werden die Arrays `d1`, `d2`, `a` und `b` (jeweils der Länge $\frac{n}{2}$) ganz durchlaufen und insgesamt $2n$ Vergleiche durchgeführt.

`rRoots` wird einmal aufgerufen (nach Bemerkung 10 mit Aufwand in $\mathcal{O}(n^2)$). Falls damit nicht alle Eigenwerte gefunden sind, werden im schlechtesten Fall nochmals die Arrays `d1` und `d2` ganz durchlaufen.

Das abschließende Sortieren des Arrays `lambda` (der Länge n) erfolgt mit einem Quicksort-Algorithmus. Dieser hat nach [11, Eigenschaft 9.1] einen Aufwand in $\mathcal{O}(n \log n)$.

Insgesamt ergibt sich ein Aufwand von $\mathcal{O}(n^2)$ für `eigenValues`. □

5.4.2 Aufwand der Eigenvektorberechnung

In diesem Abschnitt wird der Aufwand der in Kapitel 5.2 beschriebenen Prozeduren betrachtet.

Die in den Abschnitten 5.2.1 und 5.2.2 vorgestellten Prozeduren zum Lösen des unteren Dreieckssytems und des oberen Block-Gleichungssystems haben jeweils einen Aufwand in $\mathcal{O}(n)$.

Bemerkung 12 *Die Prozeduren* `solveLower` *und* `solveUpper` *haben jeweils einen Aufwand in* $\mathcal{O}(n)$.

Begründung:

- `solveLower`: Es fallen $\frac{n}{2}$ Zuweisungen an und für die Berechnung der Hilfsvariablen

$$\texttt{temp := } \sum_{j=0}^{\frac{n}{2}-1} \frac{\texttt{a[j]}}{\texttt{d1[j] - shift}} \texttt{ * y[j];}$$

  $\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1)$ Rechenoperationen (und eine weitere für die Initialisierung der Summe) und $\frac{n}{2}$ Schleifendurchläufe mit jeweils 2 Rechenoperationen. Also sind insgesamt $\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1) + 1 + \frac{n}{2} \cdot 2 = 3n$ Rechenoperationen nötig und damit liegt `solveLower` in $\mathcal{O}(n)$.

- `solveUpper`: Für den ersten Teil sind für die Berechnung der Hilfsvariablen

$$\texttt{temp}_1 \texttt{ := } \sum_{j=0}^{\frac{n}{2}-1} \frac{\texttt{b[j] * y[n/2 + j]}}{\texttt{d2[j] - shift}}\texttt{;}$$

  $\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1)$ Operationen (und eine weitere für die Initialisierung der Summe) und $\frac{n}{2}$ Schleifendurchläufe mit jeweils 7 Rechenoperationen durchzuführen, also insgesamt $\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1) + 1 + \frac{n}{2} \cdot 7 = 5n + \frac{n}{2}$ Operationen.

  Für den zweiten Teil fallen für die Hilfsvariable

$$\texttt{temp := } \sum_{j=0}^{\frac{n}{2}-1} \texttt{ b[j] * x[n/2 + j];}$$

$\frac{n}{2} + (\frac{n}{2} - 1)$ Operationen (und eine weitere für die Initialisierung der Summe) und $\frac{n}{2}$ Schleifendurchläufe mit jeweils 3 Rechenoperationen an, also insgesamt $\frac{n}{2} + (\frac{n}{2} - 1) + 1 + \frac{n}{2} \cdot 3 = 2n + \frac{n}{2}$ Operationen.

Der erste und zweite Teil zusammen braucht damit $5n + \frac{n}{2} + 2n + \frac{n}{2} = 8n$ Rechenoperationen und damit liegt der Aufwand von `solveUpper` in $\mathcal{O}(n)$.

□

Damit kann nun der Aufwand der inversen Vektoriteration betrachtet werden. In diesem speziellen Fall liegt dieser in $\mathcal{O}(n^2)$.

Bemerkung 13 *Die Prozedur* `eigenVector` *hat einen Aufwand in* $\mathcal{O}(n^2)$.

Begründung: Die Berechnung der Hilfsvariablen

$$\texttt{alpha} := \sum_{i=0}^{\frac{n}{2}-1} \frac{\texttt{a[i] * a[i]}}{\texttt{d1[i] - shift}}$$
$$\texttt{beta} := \sum_{i=0}^{\frac{n}{2}-1} \frac{\texttt{b[i] * b[i]}}{\texttt{d2[i] - shift}}$$

benötigt je $\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1)$ Operationen (plus eine für die Initialisierung der Summe) und die Berechnung von `gamma` benötigt 3 weitere Operationen, so dass insgesamt $2 \cdot (\frac{n}{2} \cdot 3 + (\frac{n}{2} - 1) + 1) + 3 = 4n + 3$ Rechenoperationen für die Hilfsvariablen nötig sind.

In jedem Iterationsschritt der inversen Vektoriteration gibt es einen Aufruf von `solveLower` und einen von `solveUpper`, beide nach Bemerkung 12 mit einem Aufwand in $\mathcal{O}(n)$, sowie eine Normierung der Iterierten (die Norm eines Arrays der Länge n zu berechnen, kostet $2n$ Rechenoperationen, und jeden Eintrag des Arrays durch die Norm zu teilen kostet weitere n Operationen, damit benötigt die Normierung insgesamt $3n$ Operationen und die Komplexität liegt auch in $\mathcal{O}(n)$).

Da der Shiftparameter für die inverse Vektoriteration sehr nahe an einem Eigenwert gewählt wird, sind nur wenige Iterationsschritte nötig, so dass es bis hier bei einem Aufwand in $\mathcal{O}(n)$ bleibt.

Die anschließende Matrix-Vektor-Multiplikation $v = Ux$ hat einen Aufwand in $\mathcal{O}(n^2)$, so dass dadurch der Gesamtaufwand von `eigenValues` auch in $\mathcal{O}(n^2)$ liegt. □

5.4.3 Gesamtaufwand des Algorithmus

In diesem Abschnitt geht es um den Aufwand des gesamten Algorithmus, wie er in Kapitel 5.3 beschrieben wird.

Dazu wird zunächst der Aufwand der Kreuzapproximation aus Abschnitt 5.3.1 betrachtet.

Bemerkung 14 *Die Prozedur* `crossApproximation` *hat einen Aufwand in* $\mathcal{O}(n^2)$.

Begründung: Um den betragsgrößten Eintrag in der Matrix $R \in \mathbb{R}^{\frac{n}{2} \times \frac{n}{2}}$ zu finden, muss im schlechtesten Fall die gesamte Matrix durchsucht, also $\frac{n}{2} \cdot \frac{n}{2}$ Einträge überprüft werden. Damit liegt der Aufwand für `getMaxEntry` in $\mathcal{O}(n^2)$.

Dann gibt es $\frac{n}{2}$ Schleifendurchläufe mit jeweils einer Zuweisung und danach $\frac{n}{2}$ Schleifendurchläufe mit jeweils einer Rechenoperation (also $\frac{n}{2}$ Operationen). Damit liegt der Aufwand für `crossApproximation` insgesamt in $\mathcal{O}(n^2)$. □

Bemerkung 15 *Mit c_A wird im Folgenden eine Konstante bezeichnet, mit der sich der Aufwand zur Berechnung der Schur-Zerlegung mit Hilfe des QR-Verfahrens für symmetrische Matrizen (bis auf Maschinengenauigkeit), nach oben durch $c_A\, n^3$ abschätzen lässt.*

Eine solche Konstante existiert nach [6, Algorithm 8.3.3].

Mit den Aufwandsabschätzungen zu den einzelnen Teilen des Algorithmus lässt sich nun eine Aussage über den Gesamtaufwand formulieren:

Danach hat der Algorithmus zur Lösung eines Eigenwertproblems für symmetrische $\mathcal{H}$-Matrizen mit lokalem Rang 1 im einfachen Modellformat (also für Matrizen aus $\mathcal{H}_l(1)$) in Form der Prozedur `eigenProblem` einen Aufwand in $\mathcal{O}(n^3)$.

Satz 7 *Die Prozedur* `eigenProblem` *angewandt auf eine symmetrische Matrix $A \in \mathcal{H}_l(1)$ hat einen Aufwand $\leq cn^3 + \mathcal{O}(n^2)$ mit $c := \max\{c_A\,,\, \frac{4}{3}\, c_4\}$, wobei $c_A > 0$ die Konstante aus Bemerkung 15 und $c_4 > 0$ die Konstante für den Aufwand von* `eigenVector` *ist.*

Beweis: Nach Definition 10 ist $n = 2^l$ die Dimension von $A \in \mathcal{H}_l(1)$.

Für die unterste Rekursionsstufe (hier bei $l = 1$, also $n = 2$) wird das Eigenwertproblem mit herkömmlichen Verfahren gelöst. O.B.d.A. wird daher der Aufwand des QR-Verfahrens für symmetrische Matrizen für diesen Schritt angenommen. Nach Bemerkung 15 ist der Rechenaufwand dafür $\leq c_A\, n^3$.

Für jede andere Rekursionsstufe (also $l > 1$ und $n = 2^l$) gibt es hintereinander zwei rekursive Aufrufe von `eigenProblem` mit Teilmatrizen von A der Größe $\frac{n}{2} \times \frac{n}{2}$.

Die Matrix-Vektor-Multiplikationen `a := U1`^{T} `* â` und `b := U2`^{T} `* b̂` mit unitären $\frac{n}{2} \times \frac{n}{2}$-Matrizen liegen jeweils in $\mathcal{O}(\,(\frac{n}{2})^2\,)$ (siehe [6, Kapitel 1.1]).

Der Aufruf von `eigenValues` hat nach Bemerkung 11 einen Aufwand in $\mathcal{O}(n^2)$.

Dann gibt es n Schleifendurchläufe mit jeweils einer Rechenoperation, je einem Aufruf von `getShiftEpsilon` mit einem konstanten Aufwand und je einen Aufruf von `eigenVector` mit Aufwand in $\mathcal{O}(n^2)$ nach Bemerkung 13.

Die Operationen, deren Aufwand sich hier addiert sind also

$2 \cdot$ `eigenProblem`
$2 \cdot$ Matrix-Vektor-Multiplikation
$1 \cdot$ `eigenValues`
$n \cdot$ eine Rechenoperation
$n \cdot$ `getShiftEpsilon`
$n \cdot$ `eigenVector`

und es existieren Konstanten c_1, c_2, c_3, c_4 mit

$$\left\{ \begin{array}{l} \text{Matrix-Vektor-Multiplikation} \\ \texttt{eigenValues} \\ \texttt{getShiftEpsilon} \\ \texttt{eigenVector} \end{array} \right\} \text{ hat obere Schranke } \left\{ \begin{array}{c} c_1 \frac{n^2}{4} \\ c_2\, n^2 \\ c_3 \\ c_4\, n^2 \end{array} \right\}$$

so dass sich der Aufwand ohne die rekursiven Aufrufe von `eigenProblem` abschätzen lässt durch

$$\begin{aligned} & 2 \cdot c_1 \frac{n^2}{4} + c_2\, n^2 + n + n \cdot c_3 + n \cdot c_4\, n^2 \\ = \;& c_4\, n^3 + \left(\frac{c_1}{2} + c_2 \right) n^2 + (1 + c_3)\, n \\ \leq \;& c_4\, n^3 + \left(\frac{c_1}{2} + c_2 + c_3 + 1 \right) n^2 \\ = \;& c_4\, n^3 + c_R\, n^2 \,. \end{aligned}$$

mit $c_R := \frac{c_1}{2} + c_2 + c_3 + 1$.

Setze

$$c \;:=\; \max \left\{ c_A \,,\, \frac{4}{3}\, c_4 \right\} \quad \text{und} \quad \widetilde{c} \;:=\; 2\, c_R \,.$$

Es bleibt zu zeigen, dass der Aufwand für `eigenProblem` $\leq c\, n^3 + \widetilde{c}\, n^2$ ist.

Induktion über l:

Induktionsanfang: Sei $l = 1$, also $n = 2$.
Dann ist der Aufwand $\leq c_A\, n^3 \;\leq\; c\, n^3 \;\leq\; c\, n^3 + \widetilde{c}\, n^2$.

Induktionsvoraussetzung: Gelte die Behauptung für ein $l - 1 \in \mathbb{N}$.

Induktionsschluss: Sei $l > 1$:
Dann gilt nach der Induktionsvoraussetzung für die beiden rekursiven Aufrufe von `eigenProblem` mit Matrizen der Größe $\frac{n}{2} \times \frac{n}{2}$, dass der Aufwand dafür jeweils $\leq\; c\, \frac{n^3}{8} + \widetilde{c}\, \frac{n^2}{4}$ ist.

Also ist der Gesamtaufwand abzuschätzen durch

$$\begin{aligned}
& c_4\, n^3 \,+\, c_R\, n^2 \,+\, 2\left(c\,\frac{n^3}{8} + \widetilde{c}\,\frac{n^2}{4}\right) \\
&= \; c_4\, n^3 \,+\, c_R\, n^2 \,+\, \frac{c}{4}\, n^3 \,+\, \frac{\widetilde{c}}{2}\, n^2 \\
&= \; \left(c_4 + \frac{1}{4}\, c\right) n^3 \,+\, \left(c_R + \frac{1}{2}\, \widetilde{c}\right) n^2 \\
&\leq \; \left(\frac{3}{4}\, c \,+\, \frac{1}{4}\, c\right) n^3 \,+\, \left(\frac{1}{2}\, \widetilde{c} + \frac{1}{2}\, \widetilde{c}\right) n^2 \\
&= \; c\, n^3 + \widetilde{c}\, n^2 \,.
\end{aligned}$$

Also ist $c\,n^3 + \widetilde{c}\,n^2$ eine obere Schranke für `eigenProblem`.

Damit ist der Aufwand für `eigenProblem` $\leq c\,n^3 + \mathcal{O}(n^2)$. □

Bemerkung 16 *In Satz 7 ist vorausgesetzt, dass die Eingabematrix bereits im $\mathcal{H}_l(1)$-Format gegeben ist, daher entfällt hier der Aufruf der Prozedur* `crossApproximation` *in* `eigenProblem`.

Auch wenn man davon ausgehen kann, dass dies der Standardfall ist, wird in dem in Abschnitt 5.5 vorgestellten praktischen Beispiel die $\mathcal{H}_l(1)$-Form erst innerhalb des Programms berechnet. Dadurch kommt in jedem Rekursionsschritt ein Aufruf von `crossApproximation` *für eine Teilmatrix von $A \in \mathbb{R}^{n\times n}$ der Größe $\frac{n}{2} \times \frac{n}{2}$ hinzu. Der Aufwand dafür liegt nach Bemerkung 14 in $\mathcal{O}(\,(\frac{n}{2})^2\,)$ und hat damit keinen Einfluss auf den Gesamtaufwand von $c\,n^3 + \mathcal{O}(n^2)$ von* `eigenProblem`.

Bemerkung 17 *Satz 7 zeigt, dass der Aufwand von* `eigenProblem` *direkt vom Aufwand der Eigenvektor-Berechnung abhängt, da die Konstante c_4 von* `eigenVector` *in der Endabschätzung im n^3-Term für den Aufwand von* `eigenProblem` *vorkommt.*

Der Gesamtalgorithmus liegt mit dem Aufwand in $\mathcal{O}(n^3)$, also in derselben Klasse wie andere Eigenwertlöser, die nicht den Vorteil der Niedrigrang-Matrizen auf den Nebendiagonalblöcken ausnutzen können. Dies bedeutet jedoch nicht, dass es gar keine Verbesserung im Aufwand gibt.

Die Eigenwertberechnung allein hat einen quadratischen Aufwand. Genauso hat die Berechnung eines Eigenvektors einen quadratischen Aufwand, da mit der Faktorisierung der Matrix in der inversen Vektoriteration die Gleichungssysteme jeweils (dank der Niedrigrang-Matrizen in den Nebendiagonalen) mit linearem Aufwand berechnet werden können.

Der einzige Faktor, der letztendlich für den kubischen Aufwand verantwortlich ist, liegt in der Rücktransformation nach der inversen Vektoriteration:

Um die günstige Struktur der Matrix ausnutzen zu können, wird die durch Ähnlichkeitstransformationen veränderte Matrix für die inverse Vektoriteration benutzt. Der so berechnete Eigenvektor gehört aber zu der veränderten Matrix. Dadurch wird die Rücktransformation in Form einer Matrix-Vektor-Multiplikation mit einer unitären Matrix notwendig. Da der Aufwand für diese Matrix-Vektor-Multiplikation quadratisch ist, ist auch der Aufwand der Berechnung eines Eigenvektors quadratisch.

Und da n Eigenvektoren berechnet werden müssen, kommt der kubische Aufwand des Algorithmus zustande.

5.5 Beispiel einer praktischen Anwendung

Dieses Kapitel befasst sich mit der praktischen Implementierung des Eigenwertlösers für symmetrische $\mathcal{H}_l(1)$-Matrizen nach dem Pseudocode aus den Abschnitten 5.1 bis 5.3.

Der Algorithmus wurde in der Programmiersprache C umgesetzt und benutzt LAPACK(**L**inear **A**lgebra **PACK**age)-Routinen aus der Sun Performance Library. So wird in dem entstandenen Programm für die unterste Rekursionsstufe, also für das kleinste Teilproblem, das Eigenwertproblem mit der Routine `dspevd` für symmetrische Matrizen gelöst und jede im Algorithmus auftretende Matrix-Vektor-Multiplikation mit der Routine `dgemv` berechnet.

Abschnitt 5.5.1 gibt einige Beispiele für Matrizen an, mit denen das Programm getestet wurde und in Abschnitt 5.5.2 wird die Laufzeit des Programms für diese Beispielmatrizen betrachtet.

5.5.1 Testmatrizen

Als Testbeispiele für das Programm werden zunächst die Tridiagonalmatrizen tridiag(-1,2,-1) aus dem 1D-Modellproblem (Poisson Gleichung) und tridiag(-1,4,-1), Teilmatrix aus dem 2D-Modellproblem, und eine Variation davon benutzt.

Die Größe des atomaren Teilproblems ist dabei auf $n_0 = 2$ gesetzt, das heißt, dass die Teilprobleme für 2×2-Matrizen mit der LAPACK-Routine `dspevd` gelöst werden. Im Folgenden wird die Genauigkeit der Berechnung für die Beispiel-Matrizen der Dimension $n = 2^{11} = 2048$ betrachtet. Durch die Wahl von n_0 und n wird eine möglichst große Rekursionstiefe für eine relativ kleine Matrix-Dimension erreicht.

Da $Ax = \lambda x$ für ein *echtes* Eigenpaar (λ, x) von A gilt, wird der Fehler der berechneten Eigenpaare mit $\|Ax - \lambda x\|$ angegeben.

Tridiagonalmatrizen

Tridiagonalmatrizen sind spezielle $\mathcal{H}_l(1)$-Matrizen. Daher wird zunächst der Fehler für Tridiagonalmatrizen untersucht.

Tabelle 5.5.1 zeigt für $A = \text{tridiag}(-1, 2, -1)$ und $n = 2048$ den Fehler der ersten und letzten 8 berechneten Eigenpaare. Der Fehler liegt dabei, wie in Abbildung 5.5.1 dargestellt, stets deutlich unter 10^{-8}. Das ist damit zu erklären, dass diese Größe als Abbruchkriterium für die Vektoriteration gewählt wurde.

Da in diesem Fall für

$$A = \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix}$$

die Teilprobleme A_1 und A_2 identisch sind, ergibt sich für die Polstellen-Nullstellen-Verteilung der Funktion r aus (4.15), dass in jedem Intervall zwischen zwei Polstellen immer genau zwei Nullstellen zu finden sind. Damit an dieser Stelle auch der Fall mit nur einer Nullstelle in einem solchen Intervall auftreten kann, wird als nächstes ein Beispiel mit $A_1 \neq A_2$ konstruiert.

Dazu wird die Tridiagonalmatrix

$$A = \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix} \quad \text{mit} \quad \begin{array}{rcl} A_1 & = & \text{tridiag}(-1, 2, -1) \\ A_2 & = & \text{tridiag}(-1, 4, -1) \end{array} \tag{5.3}$$

und Dimension $n = 2048$ gewählt. Im Folgenden wird diese Matrix auch als *gemischte Tridiagonalmatrix* bezeichnet.

In Tabelle 5.5.2 ist der Fehler der ersten und letzten 8 berechneten Eigenpaare dieser Matrix zu sehen und Abbildung 5.5.2 zeigt, dass der Fehler auch hier immer unter 10^{-8} bleibt.

An diesen Beispielen wird deutlich, dass der Algorithmus für Tridiagonalmatrizen gut funktioniert. Da aber einfachere Algorithmen (wie zum Beispiel der von Dongarra und Sorensen) für Eigenwertprobleme von Tridiagonalmatrizen existieren, wird das Programm als nächstes für $\mathcal{H}_l(1)$-Matrizen, die keine Tridiagonalmatrizen sind, getestet.

$\mathcal{H}_l(1)$-Matrizen

Für die Konstruktion von $\mathcal{H}_l(1)$-Matrizen mit möglichst vollbesetzten Rang-1-Nebendiagonalblöcken werden folgende Lemmas benötigt.

Lemma 20 *Für eine Matrix $A \in \mathcal{H}_l(1)$ der Form*

$$A = \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix} = \begin{bmatrix} A_1 & ab^T \\ ba^T & A_2 \end{bmatrix}$$

Tabelle 5.5.1: Fehler für tridiag(-1,2,-1)

	berechneter Eigenwert λ	Fehler $\|Ax - \lambda x\|$
1.	2.3508003284 e-06	1.2744970205 e-10
2.	9.4031958004 e-06	1.6918801311 e-11
3.	2.1157169832 e-05	1.2669222424 e-10
4.	3.7612694794 e-05	1.6148511237 e-11
5.	5.8769732001 e-05	1.2631552842 e-10
6.	8.4628231717 e-05	1.6711662519 e-11
7.	0.00011518813315	1.2653320962 e-10
8.	0.00015044936447	1.3587769819 e-11
⋮	⋮	⋮
2041.	3.99984955063544	1.1954933785 e-11
2042.	3.99988481186677	1.2754856500 e-10
2043.	3.99991537176819	1.5527116572 e-11
2044.	3.99994123026792	1.2775728716 e-10
2045.	3.99996238730512	1.4557278118 e-11
2046.	3.99997884283009	1.2786938668 e-10
2047.	3.99999059680411	1.4667277083 e-11
2048.	3.99999764919959	1.2767727977 e-10

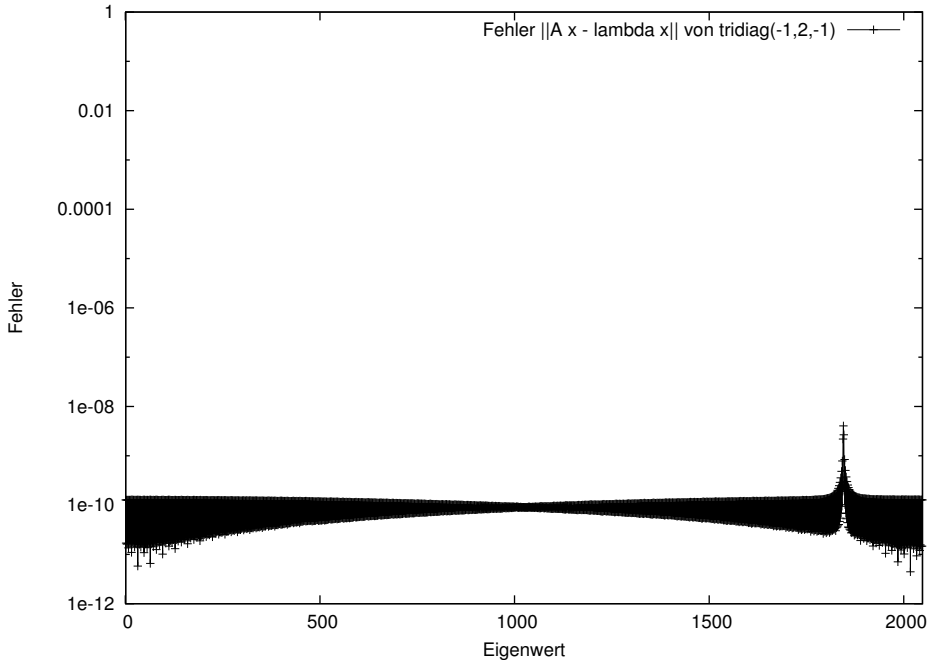


Abbildung 5.5.1: Fehlerbetrachtung für $A = \text{tridiag}(-1, 2, -1)$

Tabelle 5.5.2: Fehler für die gemischte Tridiagonalmatrix aus (5.3)

	berechneter Eigenwert λ	Fehler $\|Ax - \lambda x\|$
1.	9.3873186192 e-06	1.6948441564 e-11
2.	3.7549186367 e-05	1.6236591218 e-11
3.	8.4485338876 e-05	1.6857598577 e-11
4.	0.00015019533554	1.3919930694 e-11
5.	0.00023467855952	1.7087371772 e-11
6.	0.00033793421775	1.7121415068 e-11
7.	0.00045996134092	1.7856570503 e-11
8.	0.00060075878354	1.2542957116 e-11
9.	0.00076032522389	1.9140072433 e-11
10.	0.00093865916407	1.8464541925 e-11
⋮	⋮	⋮
2039.	5.99906134083594	2.7805175524 e-11
2040.	5.99923967477612	2.8061812856 e-11
2041.	5.99939924121647	2.3381840975 e-11
2042.	5.99954003865909	2.6902715907 e-11
2043.	5.99966206578226	2.5202622976 e-11
2044.	5.99976532144048	2.6065213852 e-11
2045.	5.99984980466447	2.2499228537 e-11
2046.	5.99991551466113	2.4231358105 e-11
2047.	5.99996245081364	2.2019131358 e-11
2048.	5.99999061268139	2.3159358999 e-11

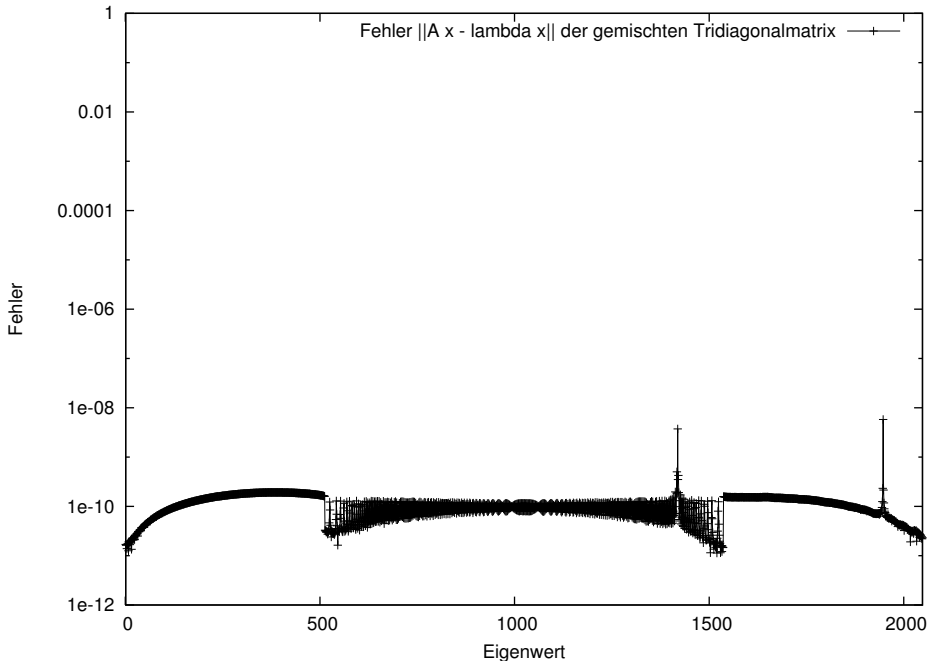


Abbildung 5.5.2: Fehlerbetrachtung für die Matrix aus (5.3)

ist die Inverse A^{-1} gegeben durch

$$A^{-1} = \begin{bmatrix} (A_1 - ab^T A_2^{-1} ba^T)^{-1} & -A_1^{-1} ab^T (A_2 - ba^T A_1^{-1} ab^T)^{-1} \\ -A_2^{-1} ba^T (A_1 - ab^T A_2^{-1} ba^T)^{-1} & (A_2 - ba^T A_1^{-1} ab^T)^{-1} \end{bmatrix}.$$

Beweis: Es gilt

$$\begin{aligned} A \cdot & \begin{bmatrix} (A_1 - ab^T A_2^{-1} ba^T)^{-1} & -A_1^{-1} ab^T (A_2 - ba^T A_1^{-1} ab^T)^{-1} \\ -A_2^{-1} ba^T (A_1 - ab^T A_2^{-1} ba^T)^{-1} & (A_2 - ba^T A_1^{-1} ab^T)^{-1} \end{bmatrix} \\ &= \begin{bmatrix} A_1 (A_1 - ab^T A_2^{-1} ba^T)^{-1} - ab^T A_2^{-1} ba^T (A_1 - ab^T A_2^{-1} ba^T)^{-1} \\ ba^T (A_1 - ab^T A_2^{-1} ba^T)^{-1} - A_2 A_2^{-1} ba^T (A_1 - ab^T A_2^{-1} ba^T)^{-1} \end{bmatrix} \\ &\quad \begin{matrix} -A_1 A_1^{-1} ab^T (A_2 - ba^T A_1^{-1} ab^T)^{-1} + ab^T (A_2 - ba^T A_1^{-1} ab^T)^{-1} \\ -ba^T A_1^{-1} ab^T (A_2 - ba^T A_1^{-1} ab^T)^{-1} + A_2 (A_2 - ba^T A_1^{-1} ab^T)^{-1} \end{bmatrix} \\ &= \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}. \end{aligned}$$

□

Lemma 21 *Die Inverse einer symmetrischen Tridiagonalmatrix aus $\mathbb{R}^{n \times n}$ mit $n = 2^l$ ist eine symmetrische $\mathcal{H}_l(1)$-Matrix.*

Beweis: Induktion über l.

Induktionsanfang: Sei $l = 0$. Dann ist $n = 2^0 = 1$. Sei $A \in \mathbb{R}^{1 \times 1}$ symmetrische Tridiagonalmatrix $A = [\alpha]$ mit $\alpha \in \mathbb{R}$. Dann ist $A^{-1} = [\alpha^{-1}] \in \mathcal{H}_0(1)$.

Induktionsvoraussetzung: Gelte die Behauptung für ein $l - 1$.

Induktionsschluss: $l - 1 \to l$: Es ist $n = 2^l$. Sei $A \in \mathbb{R}^{n \times n}$ eine symmetrische Tridiagonalmatrix. Dann gilt

$$A = \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix}$$

wobei $A_1, A_2 \in \mathbb{R}^{\frac{n}{2} \times \frac{n}{2}}$ symmetrische Tridiagonalmatrizen und $R = ab^T$ mit $a, b \in \mathbb{R}^{\frac{n}{2}}$ und $a^T = (0, \ldots, 0, 1)^T$ und $b^T = (\beta, 0, \ldots, 0)^T$ mit $\beta \in \mathbb{R}$.

Nach Lemma 20 gilt

$$A^{-1} = \begin{bmatrix} (A_1 - ab^T A_2^{-1} ba^T)^{-1} & -A_1^{-1} ab^T (A_2 - ba^T A_1^{-1} ab^T)^{-1} \\ -A_2^{-1} ba^T (A_1 - ab^T A_2^{-1} ba^T)^{-1} & (A_2 - ba^T A_1^{-1} ab^T)^{-1} \end{bmatrix}.$$

Die Inverse einer symmetrischen Matrix ist wieder symmetrisch, also ist A^{-1} symmetrisch.

Eine Matrix-Multiplikation mit einer Rang-1-Matrix ergibt wieder eine Rang-1-Matrix. Damit sind die Nebendiagonalblöcke

$$-A_1^{-1} \underbrace{ab^T}_{\text{Rang 1}} (A_2 - ba^T A_1^{-1} ab^T)^{-1} \text{ und } -A_2^{-1} \underbrace{ba^T}_{\text{Rang 1}} (A_1 - ab^T A_2^{-1} ba^T)^{-1}$$

von A^{-1} auch Rang-1-Matrizen.

Weiter gilt für die Hauptdiagonalblöcke

$$\begin{aligned} A_1 - ab^T A_2^{-1} ba^T &= A_1 - a\,\gamma\,a^T \qquad \text{mit } \gamma := b^T A_2^{-1} b \in \mathbb{R} \\ &= A_1 - \gamma \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} (0, \ldots, 0, 1) \\ &= A_1 - \begin{pmatrix} 0 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & \gamma \end{pmatrix} \\ &:= \tilde{A}_1 \end{aligned}$$

Offensichtlich ist $\tilde{A}_1$ wieder eine symmetrische Tridiagonalmatrix (da A_1 eine ist) und nach der Induktionsvoraussetzung ist $\tilde{A}_1{}^{-1} \in \mathcal{H}_{l-1}(1)$. Damit ist $A^{-1} \in \mathcal{H}_l(1)$.

□

Die Idee ist, die Inversen der bisherigen Beispielmatrizen zu testen: Die Tridiagonalmatrizen tridiag$(-1, 2, -1)$ und tridiag$(-1, 4, -1)$ sind spezielle $\mathcal{H}_l(1)$-Matrizen und da sie symmetrisch sind, sind nach Lemma 21 ihre Inversen auch aus $\mathcal{H}_l(1)$, allerdings mit mehr als nur einem Eintrag in den Nebendiagonalblöcken.

Tabelle 5.5.3 zeigt für $A = (\,\text{tridiag}(-1, 4, -1))^{-1}$ und $n = 2048$ den Fehler der ersten und letzten 8 berechneten Eigenpaare. Der Fehler liegt dabei, wie auch in Abbildung 5.5.3 dargestellt, für alle Eigenpaare hier sogar unter 10^{-10}.

Wie schon bei den Tridiagonalmatrizen erklärt, soll auch hier der Fall $A_1 \neq A_2$ betrachtet werden. Dafür wird die Inverse der Matrix aus (5.3) verwendet.

Tabelle 5.5.4 und Abbildung 5.5.4 stellen den Fehler der Eigenpaare der Inversen von A aus (5.3) dar. Auch hier bleibt der Fehler klein, er liegt immer unter 10^{-8} trotz der stark ansteigenden Eigenwerte am oberen Ende des Spektrums.

Tabelle 5.5.3: Fehler für die Inverse von tridiag(-1,4,-1)

	berechneter Eigenwert λ	Fehler $\|Ax - \lambda x\|$
1.	0.16666673196669	4.0773514439 e-12
2.	0.16666692786695	4.0896436842 e-12
3.	0.16666725436789	4.1116844850 e-12
4.	0.16666771147028	4.0849177462 e-12
5.	0.16666829917520	4.1220765983 e-12
6.	0.16666901748403	4.1187820556 e-12
7.	0.16666986639845	4.1249040799 e-12
8.	0.16667084592046	4.0820969510 e-12
⋮	⋮	⋮
2041.	0.49996239048805	1.4444799499 e-11
2042.	0.49997120462516	1.4463462706 e-11
2043.	0.49997884383728	1.4477871934 e-11
2044.	0.49998530799873	1.4496945501 e-11
2045.	0.49999059700314	1.4475656443 e-11
2046.	0.49999471076350	1.4504023145 e-11
2047.	0.49999764921210	1.4496742271 e-11
2048.	0.49999941230061	1.4496654931 e-11

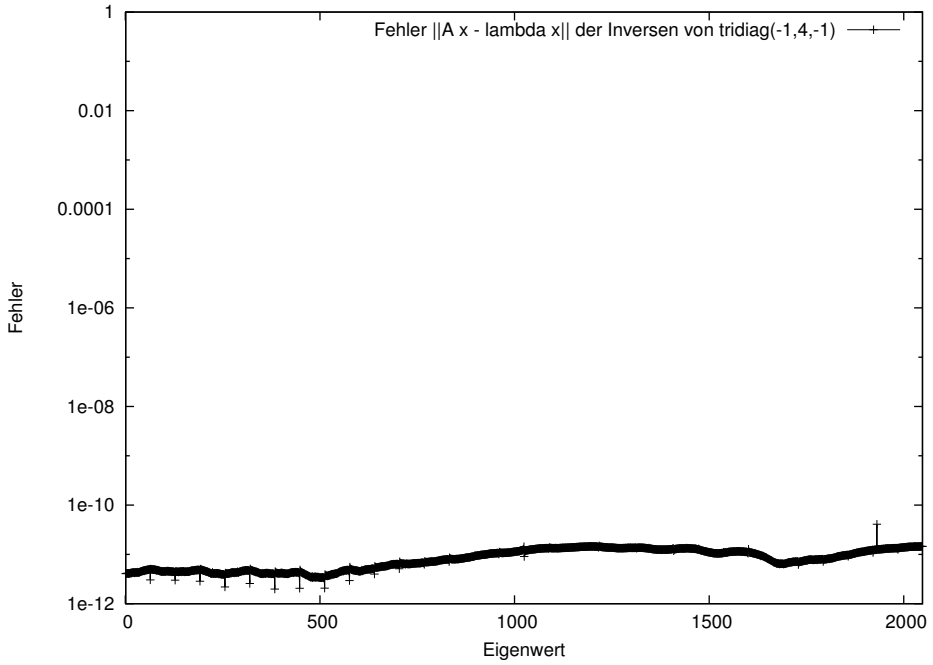


Abbildung 5.5.3: Fehlerbetrachtung für $A = (\,\mathrm{tridiag}(-1, 4, -1))^{-1}$

Tabelle 5.5.4: Fehler für die Inverse der Matrix aus (5.3)

	berechneter Eigenwert λ	Fehler $\|Ax - \lambda x\|$
1.	0.16666692742591	4.0885932010 e-12
2.	0.16666770970614	4.0817422544 e-12
3.	0.16666901351467	4.0940180383 e-12
4.	0.16667083886375	4.0640692272 e-12
5.	0.16667318577051	4.0909598428 e-12
6.	0.16667605425699	4.1415600885 e-12
7.	0.16667944435011	4.2340984202 e-12
8.	0.16668335608169	4.1564063657 e-12
⋮	⋮	⋮
2041.	1664.56159676701736	7.7164935217 e-10
2042.	2174.09575768803870	7.0509819284 e-10
2043.	2959.15579861420838	7.9938677259 e-10
2044.	4261.14768228672710	9.2112244521 e-10
2045.	6657.99637766989599	1.4317963526 e-09
2046.	11836.37318916958611	1.8201097912 e-09
2047.	26631.73550832982801	2.6632827566 e-09
2048.	106526.69203269435093	5.2012692496 e-09

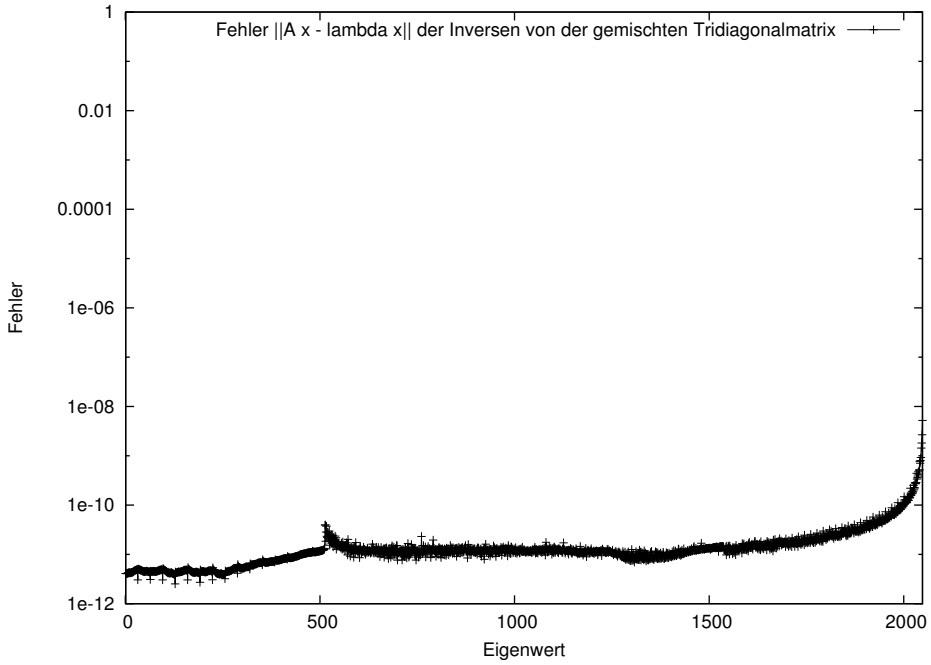


Abbildung 5.5.4: Fehlerbetrachtung für die Inverse von (5.3)

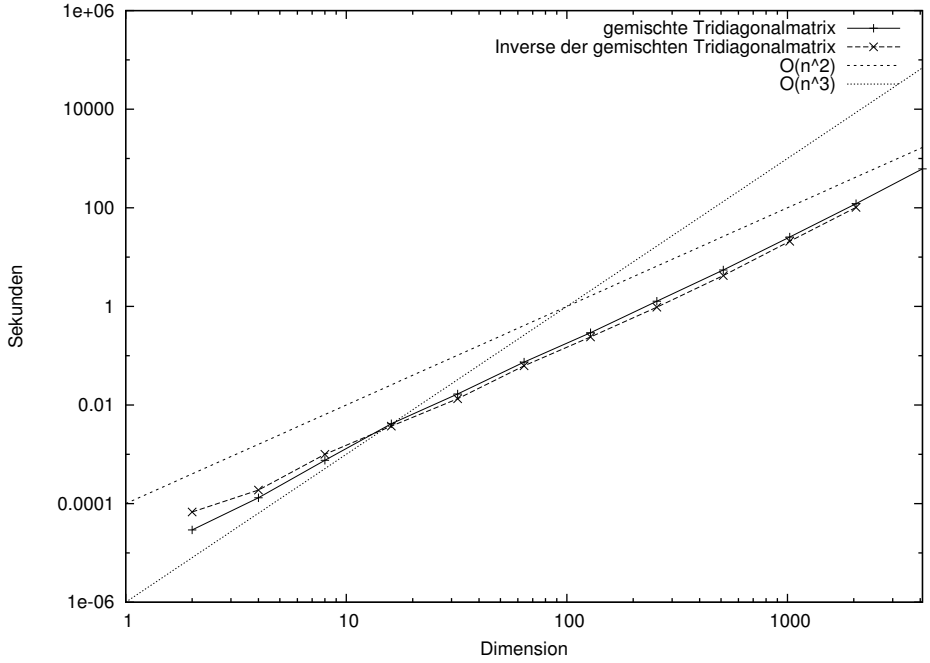


Abbildung 5.5.5: Laufzeitvergleich für (5.3) und ihre Inverse

5.5.2 Laufzeit der Beispiele

In diesem Abschnitt wird die Laufzeit des Programms in Hinblick auf die Ergebnisse aus Kapitel 5.4 zum Aufwand des Algorithmus betrachtet.

Nach Satz 7 liegt der Aufwand zur Lösung eines Eigenwertproblems für eine $n \times n$-Matrix $A \in \mathcal{H}_l(1)$ mit `eigenProblem` aus Abschnitt 5.3.2 in $\mathcal{O}(n^3)$. Danach ist für die Laufzeit ein in etwa kubischer Anstieg für größer werdende Dimensionen n zu erwarten.

Als Laufzeitbeispiele werden hier die Zeiten für die Berechnung der Eigenwertprobleme der Matrix aus (5.3) und ihrer Inversen für jeweils $n = 2^l$ und $l = 1, \ldots, 12$ gewählt.

Abbildung 5.5.5 zeigt die Kurven, die daraus entstehen, unterlegt mit einer Kurve für eine kubisch ansteigende Laufzeit und zum Vergleich noch einer Kurve für eine quadratisch steigende Laufzeit.

An der Abbildung wird deutlich, dass für diese relativ kleinen Dimensionen die Laufzeitkurven für die Testbeispiele sogar noch unter der quadratischen Kurve liegen, sich aber mit steigender Dimension der kubischen annähern.

Damit sind die Ergebnisse aus der Theorie an dem praktischen Beispiel wiederzuerkennen, wie auch die abschließenden Überlegungen aus Kapitel 5.4.3, wonach *nur die eine* Matrix-Vektor-Multiplikation am Ende jeder Eigenvektor-Berechnung für den kubischen Aufwand verantwortlich ist.

Kapitel 6

Zusammenfassung und Ausblick

In diesem letzten Kapitel werden die wichtigsten Grundgedanken und Ergebnisse dieser Arbeit in Unterkapitel 6.1 zusammengefasst und in 6.2 einige Ideen geäußert, wie das Thema weiter behandelt werden könnte.

6.1 Zusammenfassung

Die Aufgabe hinter dieser Arbeit, das Lösen eines Eigenwertproblems hierarchischer Matrizen, wurde anhand des einfachen Modellformats für $\mathcal{H}$-Matrizen betrachtet. Noch weiter vereinfacht wurde ein Algorithmus zur Lösung eines Eigenwertproblems für nur symmetrische hierarchische Matrizen in dem Modellformat mit lokalem Rang 1 entwickelt.

Die Konstruktion dieses Algorithmus beruht auf der Idee, dass eine $\mathcal{H}_l(1)$-Matrix Ähnlichkeiten zu einer Tridiagonalmatrix aufweist, eine Tridiagonalmatrix sogar eine spezielle $\mathcal{H}_l(1)$-Matrix ist. Daher diente ein Algorithmus für Tridiagonalmatrizen, entwickelt von Cuppen und Dongarra/Sorensen, als Vorbild. Der Divide-and-Conquer-Teil, welcher auf eine rationale Gleichung führt, konnte größtenteils übertragen werden. Für das Lösen dieser Gleichung musste aber eine andere Methode benutzt werden als die von Sorensen und Dongarra verwendete.

Die Berechnung der Eigenvektoren der Teilprobleme ist absolut notwendig für die Bestimmung der Eigenwerte mit diesem Algorithmus. Also auch wenn nur die Eigenwerte gewünscht sind, müssen die Eigenvektoren (zumindest die für alle Teilprobleme) berechnet werden. Allerdings ist durch die Struktur der Matrix mit den Rang-1-Nebendiagonalblöcken die inverse Vektoriteration günstig durchzuführen bis auf die Rücktransformierung in Form einer Matrix-Vektor-Multiplikation je Eigenvektor.

Anhand eines praktischen Beispiels wurde gezeigt, dass der Algorithmus eine Möglichkeit bietet, alle Eigenwerte sowie alle Eigenvektoren von

Matrizen dieser speziellen Matrixklasse $\mathcal{H}_l(1)$ mit guter Genauigkeit zu bestimmen.

Leider ist der Aufwand dazu weiterhin kubisch, aufgrund der notwendigen Rücktransformation mit quadratischem Aufwand für jeden der n Eigenvektoren. Damit liegt der Algorithmus in derselben Komplexitätsklasse wie andere Eigenwertlöser, die nicht den Vorteil einer besonderen Matrixstruktur ausnutzen.

Es sind noch einige Fragen offen, auf die in dem folgenden Unterkapitel 6.2 jeweils kurz eingegangen wird:

- Kann der Aufwand noch reduziert werden?
  Könnte die Rücktransformation des Eigenvektors umgangen oder günstiger berechnet werden?
- Der Algorithmus ist für $\mathcal{H}_l(1)$-Matrizen entwickelt.
  Gibt es eine Möglichkeit der Erweiterung für $\mathcal{H}_l(k)$-Matrizen mit einem allgemeinen $k > 1$?
- Das eigentliche Problem ist hier nicht gelöst.
  Geben diese Betrachtungen einen Aufschluss darüber, wie man an ein Eigenwertproblem für allgemeine $\mathcal{H}$-Matrizen herangehen könnte?

6.2 Ausblick

Hier soll versucht werden, Antworten auf die Fragen am Ende des vorangegangenen Abschnitt 6.1 zu finden. Dazu werden nur Ideen vorgestellt, auf die im Detail aber nicht mehr eingegangen wird.

6.2.1 Reduktion des Aufwands

Wie bereits in Kapitel 5.4 gesehen, liegt die kritische Stelle für den Aufwand des $\mathcal{H}_l(1)$-Algorithmus in der Rücktransformation des Eigenvektors am Ende der inversen Vektoriteration. Falls genaueres über die Transformationsmatrix bekannt wäre, könnte die Matrix-Vektor-Multiplikation unter Umständen verbessert werden.

Die Transformationsmatrix ist die Matrix

$$U = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix}$$

mit den Teilmatrizen U_1 und U_2, die spaltenweise die Eigenvektoren der beiden (bereits gelösten) Teilprobleme enthalten. Wenn sich diese Matrizen effizient darstellen ließen, etwa in Form von $\mathcal{H}$-Matrizen, könnte sich der Aufwand für die Rücktransformation verringern lassen.

Mit einer effizienten Rücktransformation wäre für den gesamten $\mathcal{H}_l(1)$-Algorithmus ein quadratischer Aufwand zu erreichen.

6.2.2 Erweiterung für das $\mathcal{H}$-Matrix-Modell von allgemeinem lokalen Rang k

Bisher wurden nur Matrizen $A \in \mathcal{H}_l(k)$ mit $k = 1$ der Form aus (4.8), also

$$A = \begin{bmatrix} A_1 & R \\ R^T & A_2 \end{bmatrix} = \begin{bmatrix} A_1 & \hat{a}\hat{b}^T \\ \hat{b}\hat{a}^T & A_2 \end{bmatrix},$$

betrachtet. Dann wurden die Teilprobleme für A_1 und A_2 gelöst und mit einer Ähnlichkeitstransformation wurde A auf die Form aus (4.11), also

$$A' = \begin{bmatrix} D_1 & ab^T \\ ba^T & D_2 \end{bmatrix},$$

gebracht. Das ist eine Diagonalmatrix D mit einer Rang-2-Störung, die sich nach Lemma 16 in 2 Rang-1-Störungen aufteilen lässt:

$$A' = D + z_1 z_1^T - z_2 z_2^T .$$

In Abschnitt 4.1.1 wurde gezeigt, wie die Suche nach Eigenwerten einer Diagonalmatrix mit einer Rang-1-Störung auf die Säkulärgleichung führt, die dann, wie in Abschnitt 4.1.2 beschrieben, gelöst werden kann.

Das Eigenwertproblem von A kann also nicht nur mit dem vorgestellten $\mathcal{H}_l(1)$-Algorithmus berechnet werden, sondern auch mit zweimaligem Lösen der Säkulärgleichung (siehe Beweis von Satz 5 und wegen des negativen Vorzeichens siehe Bemerkung 5).

Die Idee für die Erweiterung auf einen größeren lokalen Rang ist, genau dieses Vorgehen auf eine Matrix $M \in \mathcal{H}_l(k)$ mit $k > 1$ zu übertragen. Für

$$M = \begin{bmatrix} M_1 & R \\ R^T & M_2 \end{bmatrix} = \begin{bmatrix} M_1 & \hat{A}\hat{B}^T \\ \hat{B}\hat{A}^T & M_2 \end{bmatrix}$$

werden die Teilprobleme M_1 und M_2 gelöst und die Ähnlichkeitstransformation wie in (4.11) durchgeführt. Damit erhält man

$$M' = \begin{bmatrix} D_1 & AB^T \\ BA^T & D_2 \end{bmatrix},$$

also eine Diagonalmatrix D mit einer Rang-$2k$-Störung, auf die k-mal Lemma 16 angewendet werden kann, so dass daraus $2k$ Rang-1-Störungen werden:

$$M' = D + \sum_{i=1}^{k} z_{1_i} z_{1_i}^T - z_{2_i} z_{2_i}^T . \tag{6.1}$$

Also kann das Eigenwertproblem einer $\mathcal{H}_l(k)$-Matrix gelöst werden, indem entweder k-mal der $\mathcal{H}_l(1)$-Algorithmus angewendet wird, oder $2k$-mal die

Säkulärgleichung gelöst wird. Allerdings muss für die erste Variante k-mal der gesamte $\mathcal{H}_l(1)$-Algorithmus in jedem Rekursionsschritt durchgeführt, oder für die zweite Variante in jedem Rekursionsschritt $2k$-mal das Hebden-ähnliche Verfahren aus 4.1.2 zum Lösen der Säkulärgleichung angewendet werden.

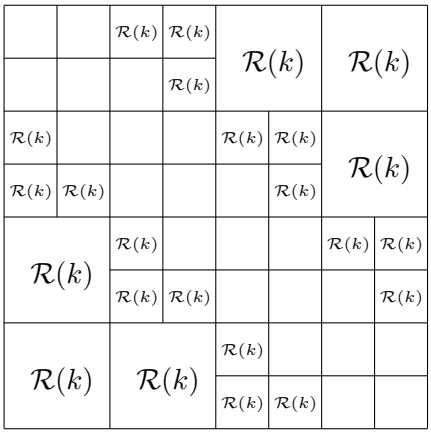
Insgesamt stellt sich die Frage, ob ein Algorithmus, der nach dieser Idee arbeitet, effizient umzusetzen wäre.

6.2.3 Erweiterung für eine allgemeine $\mathcal{H}$-Matrix-Struktur

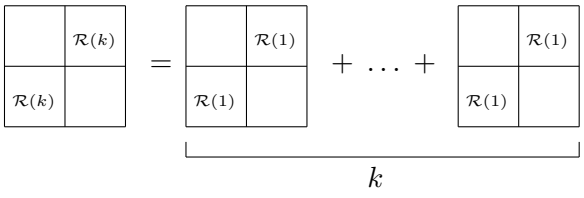
Die Idee für die Erweiterung auf allgemeine (symmetrische) $\mathcal{H}$-Matrizen soll hier nur grob skizziert werden.

Die Struktur einer hierarchischen Matrix M mit festem lokalen Rang k hat etwa folgende Form:

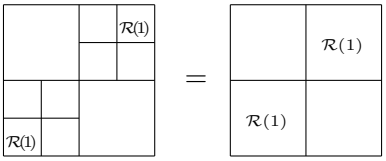
$M =$



Jede Rang-k-Matrix kann dabei als Summe von k Rang-1-Matrizen geschrieben werden. Anschaulich kann ein Block in folgender Form aufgeteilt werden:



Weiter gilt



mit entsprechend aufgefüllten Matrizen A und B für die Rang-k-Darstellung aus (3.1). Damit lässt sich M dann ähnlich wie in (6.1) schreiben als

$$M = \begin{array}{|cc|cc|cc|}\hline & & 0 & 0 & \multicolumn{2}{c|}{\multirow{2}{*}{0}} & \multirow{2}{*}{0} \\ & & & 0 & & & \\ \hline 0 & & & & 0 & 0 & \multirow{2}{*}{0} \\ 0 & 0 & & & & 0 & \\ \hline \multicolumn{2}{|c|}{\multirow{2}{*}{0}} & 0 & & & & 0 \quad 0 \\ & & 0 & 0 & & & \quad 0 \\ \hline \multicolumn{2}{|c|}{\multirow{2}{*}{0}} & \multicolumn{2}{c|}{\multirow{2}{*}{0}} & 0 & & \\ & & & & 0 & 0 & \\ \hline \end{array} + \sum \text{Rang-}2k\text{-Störungen}$$

Die *störenden* vollbesetzten Blöcke in der Mitte sind im Vergleich zur Größe der ganzen Matrix M klein. Sei die Größe dieser Blöcke $m \times m$. Dann können diese auch höchstens einen Rang m haben, also geschrieben werden als

$$\begin{array}{|c|c|}\hline & 0 \\ \hline 0 & \\ \hline \end{array} = \begin{array}{|c|c|}\hline & \mathcal{R}(m) \\ \hline \mathcal{R}(m) & \\ \hline \end{array}$$

Folglich gilt für die ganze Matrix

$$M = \left[\begin{array}{c|c} M_1 & 0 \\ \hline 0 & M_2 \end{array}\right] + \text{Rang-}2m\text{-Störung} + \sum_{\#\mathcal{R}(k)} \text{Rang-}2k\text{-Störungen}\,.$$

Nach den Überlegungen aus Abschnitt 6.2.2 könnten die Teilprobleme M_1 und M_2 gelöst werden, die Ähnlichkeitstransformation ausgeführt werden und dann wäre das Eigenwertproblem für M durch $m + \sum k$ Anwendungen des $\mathcal{H}_l(1)$-Algorithmus oder durch $(2m + \sum 2k)$-maliges Lösen der Säkulärgleichung zu berechnen.

Dabei treten allerdings weitere Probleme auf, für die bisher keine zufriedenstellende Lösung gefunden wurde und welche in dieser Arbeit nicht mehr diskutiert werden.

Literaturverzeichnis

[1] S. Börm, L. Grasedyck, and W. Hackbusch. *Hierarchical Matrices*. Max-Planck-Institut für Mathematik in den Naturwissenschaften, Leipzig, 2003.

[2] J. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36:177–195, 1981.

[3] W. Dahmen and A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Verlag, Berlin, 2.Auflage edition, 2008.

[4] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM Journal on Scientific and Statistical Computing*, 8(2):139–154, 1987.

[5] G. Fischer. *Lineare Algebra*. Vieweg Verlag, Braunschweig/Wiesbaden, 10.Auflage edition, 1997.

[6] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[7] W. Hackbusch. Numerische Lösung von Eigenwertaufgaben. (Vorlesungsskript), 2005.

[8] W. Hackbusch. *Hierarchische Matrizen - Algorithmen und Analysis*. Springer-Verlag, Berlin, 2009.

[9] B. N. Parlett. *The Symmetric Eigenvalue Problem*. The Society for Industrial and Applied Mathematics, siam edition, 1998.

[10] R. Plato. *Numerische Mathematik kompakt*. Vieweg Verlag, Braunschweig/Wiesbaden, 2000.

[11] R. Sedgewick. *Algorithmen*. Addison-Wesley (Deutschland) GmbH, 1.Auflage edition, 1991.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Literatur angefertigt habe. Alle Zitate sind als solche kenntlich gemacht.

Kiel, den 18.5.2009