Numerik für Informatiker

Lars Grüne
Lehrstuhl für Angewandte Mathematik
Mathematisches Institut
Universität Bayreuth
95440 Bayreuth
lars.gruene@uni-bayreuth.de
www.math.uni-bayreuth.de/∼lgruene/

Karl Worthmann
Gewönliche Differentialgleichungen
Institut für Mathematik
Technische Universität Ilmenau
98693 Ilmenau
karl.worthmann@tu-ilmenau.de
http://www.tu-ilmenau.de/de/analysis/team/karl-worthmann/

Vorlesungsskript

Sommersemester 2017

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Numerische Mathematik — oder kurz Numerik — beschäftigt sich mit der Entwicklung und der Analyse von Algorithmen, mit denen mathematische Probleme am Computer gelöst werden können. Im Gegensatz zu symbolischen oder analytischen Rechnungen will man hier keine geschlossenen Formeln oder algebraischen Ausdrücke als Ergebnisse erhalten, sondern quantitative Zahlenwerte[1], daher der Name „Numerische Mathematik“.

In dieser Grundvorlesung zur Numerik werden die folgenden mathematischen Probleme behandelt:

- Lösung linearer Gleichungssysteme
- Berechnung von Eigenwerten
- Interpolation
- Integration

Wichtige Bereiche, die in dieser Aufzählung fehlen, sind die nichtlinearen Gleichungen / Gleichungssysteme sowie die Differentialgleichungen; diese werden jedoch aus Zeitgründen nicht in der Vorlesung vorkommen.

Die Fülle unterschiedlicher Probleme aus der Analysis und der linearen Algebra bringt es mit sich, dass ganz verschiedene mathematische Techniken aus diesen Gebieten zur numerischen Lösung verwendet werden. Aus diesem Grund wird gerade die erste Numerik-Vorlesung oft als „Gemischtwarenladen“ aufgefasst, in dem verschiedene Themen scheinbar zusammenhanglos abgehandelt werden. Tatsächlich gibt es aber – trotz der unterschiedlichen Mathematik – eine Reihe von Grundprinzipien, die in der Numerik wichtig sind. Bevor wir im nächsten Kapitel mit der *harten* Mathematik beginnen, wollen wir diese Prinzipien in dieser Einführung kurz und informell erläutern.

- **Korrektheit**: Eine der wesentlichen Aufgaben der numerischen Mathematik ist es, die Korrektheit von Algorithmen zu überprüfen, das heißt sicherzustellen, dass und

[1] die dann natürlich oft grafisch aufbereitet werden

unter welchen Voraussetzungen an die Problemdaten tatsächlich das richtige Ergebnis berechnet wird. Diese Überprüfung soll natürlich mit mathematischen Methoden durchgeführt werden, d.h. am Ende steht ein formaler mathematischer Beweis, der die korrekte Funktion eines Algorithmus sicherstellt. In vielen Fällen wird ein Algorithmus kein exaktes Ergebnis in endlich vielen Schritten liefern, sondern eine Näherungslösung bzw. eine Folge von Näherungslösungen. In diesem Fall ist zusätzlich zu untersuchen, wie groß der Fehler der Näherungslösung in Abhängigkeit von den vorhandenen Parametern ist bzw. wie schnell die Folge von Näherungslösungen gegen den exakten Wert konvergiert.

- **Effizienz**: Hat man sich von der Korrektheit eines Algorithmus überzeugt, so stellt sich im nächsten Schritt die Frage nach seiner Effizienz. Liefert der Algorithmus ein exaktes Ergebnis in endlich vielen Schritten, so ist im Wesentlichen die Anzahl der Operationen „abzuzählen“; falls eine Folge von Näherungslösungen berechnet wird, so muss die Anzahl der Operationen pro Näherungslösung und die Konvergenzgeschwindigkeit gegen die exakte Lösung untersucht werden.

  Oft gibt es viele verschiedene Algorithmen zur Lösung eines Problems, die je nach den weiteren Eigenschaften des Problems unterschiedlich effizient sind.

- **Robustheit und Kondition**: Selbst wenn ein Algorithmus in der Theorie in endlich vielen Schritten ein exaktes Ergebnis liefert, wird dies in der numerischen Praxis nur selten der Fall sein. Der Grund hierfür liegt in den sogenannten *Rundungsfehlern*: Intern kann ein Computer nur endlich viele Zahlen darstellen, es ist also unmöglich, jede beliebige reelle (ja nicht einmal jede rationale) Zahl exakt darzustellen. Wir wollen diesen Punkt etwas formaler untersuchen. Für eine gegebene *Basis* $B \in \mathbb{N}$ kann jede reelle Zahl $x \in \mathbb{R}$ als

  $$x = m \cdot B^e$$

  dargestellt werden, wobei $m \in \mathbb{R}$ die *Mantisse* und $e \in \mathbb{Z}$ der *Exponent* genannt wird. Durch geeignete Wahl von e reicht es dabei, Zahlen der Form $m = \pm 0.m_1m_2m_3\ldots$ mit den Ziffern $m_1, m_2, \ldots$ mit $m_i \in \{0, 1, \ldots, B-1\}$ zu benutzen. Computerintern wird üblicherweise die Basis $B = 2$ verwendet, da die Zahlen als *Binärzahlen* dargestellt werden. Im Rechner stehen nun nur endlich viele Stellen für m und e zur Verfügung, z.B. l Stellen für m und n Stellen für e. Wir schreiben $m = \pm 0.m_1m_2m_3\ldots m_l$ und $e = \pm e_1e_2\ldots e_n$. Unter der zusätzlichen *Normierungs-Bedingung* $m_1 \neq 0$ ergibt sich eine eindeutige Darstellung der sogenannten *maschinendarstellbaren Zahlen*

  $$\mathcal{M} = \{x \in \mathbb{R} \mid \pm 0.m_1m_2m_3\ldots m_l \cdot B^{\pm e_1e_2\ldots e_n}\} \ \cup \ \{0\}.$$

  Zahlen, die nicht in dieser Menge $\mathcal{M}$ liegen, müssen durch Rundung in eine maschinendarstellbare Zahl umgewandelt werden.

  Die dadurch entstehenden Ungenauigkeiten beeinflussen offensichtlich das Ergebnis numerischer Algorithmen. Die Robustheit eines Algorithmus ist nun dadurch bestimmt, wie stark diese Rundungsfehler sich im Ergebnis auswirken. Tatsächlich betrachtet man die Robustheit mathematisch für allgemeine Fehler, so dass egal ist, ob diese durch Rundung oder weitere Fehlerquellen (Eingabe- bzw. Übertragungsfehler, Ungenauigkeiten in vorausgegangenen Berechnungen etc.) hervorgerufen worden sind.

Ein wichtiges Hilfsmittel zur Betrachtung dieser Problemstellung ist der Begriff der *Kondition* eines mathematischen Problems. Wenn wir das Problem abstrakt als Abbildung $\mathcal{A} : D \to L$ der Problemdaten D (z.B. Matrizen, Messwerte...) auf die Lösung L des Problems betrachten, so gibt die Kondition an, wie stark sich kleine Änderungen in den Daten D in der Lösung L auswirken, was durch die Analyse der Ableitung von $\mathcal{A}$ quantitativ bestimmt wird. Wenn kleine Änderungen in D große Änderungen in L verursachen können spricht man von einem *schlecht konditionierten* Problem.

Beispiel [6, Kapitel 7]: für einen Datenvektor $x \in \mathbb{R}^n$, $n \in \mathbb{N}$, gelte $\mathcal{A}(x) = y \in \mathbb{R}$. Stört man die Daten (absoluter Feher) um Δx, verursacht dies einen (absoluten) Fehler Δy im Ergebnis: $\mathcal{A}(x + \Delta x) = y + \Delta y$. Eine Taylorentwicklung liefert

$$y + \Delta y = \mathcal{A}(x + \Delta x) = \mathcal{A}(x) + \mathcal{A}'(x)\Delta x + \mathcal{O}(\|\Delta x\|^2).$$

Entsprechend gilt $\Delta y \approx \mathcal{A}'(x)\Delta x$ unter Vernachlässigung von Termen höherer Ordnung. Für $y \neq 0$, $x \neq 0_{\mathbb{R}^n}$ und die euklidische Norm erhält man so

$$\frac{\|\Delta y\|}{\|y\|} \approx \frac{\|x\| \cdot \|\mathcal{A}'(x)\Delta x\|}{\|y\| \cdot \|x\|} \leq \frac{\|x\| \cdot \|\mathcal{A}'(x)\|}{\|y\|} \cdot \frac{\|\Delta x\|}{\|x\|} =: \mathcal{K}_{\mathcal{A}}(x)\frac{\|\Delta x\|}{\|x\|},$$

also eine Abschätzung des relativen Fehlers im Ergebnis in Abhängigkeit des relativen Fehlers in den Eingabedaten, wobei $\mathcal{K}_{\mathcal{A}}$ einen Verstärkungsfaktor bezeichnet. Dieser wächst proportional mit dem Quotienten (Verhältnis) $\|x\|/\|y\|$ und der Norm der Ableitung $\mathcal{A}'(x)$.

Übung: berechnen Sie den Verstärkungsfaktor für die Elementaroperationen (Addition, Subtraktion, Multiplikation und Division) und überlegen Sie sich, wann $\mathcal{K}_{\mathcal{A}}$ besonders groß wird (Stichwort: Auslöschung). Welche Konsequenz leiten Sie daraus für den Algorithmenentwurf ab?

Berechnen Sie den Verstärkungsfaktor $\mathcal{K}_f$ für die Funktion $f(x) := 1 - \sqrt{1 - x^2}$. Berechnen Sie anschließend $f(x)$ für $x = 0.1 \cdot 10^{-i}$ für große $i \in \mathbb{N}$ mit einem Taschenrechner. Was beobachten Sie? Interpretieren Sie Ihre Ergebnisse.[2] Geben Sie eine numerisch stabile Berechnungsvorschrift an.[3]

Beachte, dass die Kondition eine Eigenschaft des gestellten Problems und damit unabhängig von dem verwendeten Algorithmus ist. Allerdings ist die Robustheit eines Algorithmus besonders bei schlecht konditionierten Problemen wichtig, da hier kleine Fehler im Algorithmus große Fehler im Ergebnis verursachen können. Am schönsten entwickelt ist diese Theorie bei linearen Gleichungssystemen; in diesem Rahmen werden wir sie auch ausführlich untersuchen.

[2] Es treten große relative Fehler im Ergebnis auf trotz eines kleinen Verstärkungsfaktors (Auslöschung), der Algorithmus (Berechnungsmethode) ist numerisch instabil.

[3] Erweitern Sie mit $1 + \sqrt{1 - x^2}$ und vereinfachen Sie dann den Zähler so weit wie möglich.

Kapitel 2

Lineare Gleichungssysteme

Algorithmen zur Lösung linearer Gleichungssysteme bilden die Basis für viele Anwendungen der Numerik und stehen daher traditionell am Anfang vieler Numerik-Vorlesungen. Ausführlich aufgeschrieben besteht das Problem darin, Zahlen $x_1, \ldots, x_n \in \mathbb{R}$ zu bestimmen, für die das Gleichungssystem

$$\begin{array}{ccccccccc} a_{1\,1}x_1 & + & a_{1\,2}x_2 & + & \ldots & + & a_{1\,n}x_n & = & b_1 \\ & & & & & & & \vdots & \\ a_{n\,1}x_1 & + & a_{n\,2}x_2 & + & \ldots & + & a_{n\,n}x_n & = & b_n \end{array} \tag{2.1}$$

erfüllt ist. Die ausführliche Schreibweise in (2.1) ist etwas unhandlich, weswegen wir lineare Gleichungsysteme in der üblichen Matrix-Form schreiben werden, nämlich als

$$Ax = b, \tag{2.2}$$

mit

$$A = \begin{pmatrix} a_{1\,1} & a_{1\,2} & \ldots & a_{1\,n} \\ \vdots & \vdots & & \vdots \\ a_{n\,1} & a_{n\,2} & \ldots & a_{n\,n} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}. \tag{2.3}$$

Diese Schreibweise hat nicht nur den Vorteil, dass man ein Gleichungssystem viel kürzer aufschreiben kann, es wird sich auch zeigen, dass gewisse Eigenschaften der Matrix A entscheiden, was für ein Verfahren zur Lösung von (2.2) sinnvollerweise einzusetzen ist.

Einige kurze Bemerkungen zur Notation: Wir werden Matrizen typischerweise mit großen Buchstaben bezeichnen (z.B. A) und Vektoren mit kleinen (z.B. b). Ihre Einträge werden wir mit indizierten Kleinbuchstaben bezeichnen, wie in (2.3). Mit einem hochgestellten „T“ bezeichnen wir transponierte Matrizen und Vektoren, für A und x aus (2.3) also z.B.

$$x = (x_1, \ldots, x_n)^T, \quad A^T = \begin{pmatrix} a_{1\,1} & a_{2\,1} & \ldots & a_{n\,1} \\ \vdots & \vdots & & \vdots \\ a_{1\,n} & a_{2\,n} & \ldots & a_{n\,n} \end{pmatrix}.$$

Da die Anzahl n der Gleichungen in (2.1) gleich der Anzahl der Unbekannten $x_1, \ldots, x_n$ ist, ist A in (2.2) eine quadratische Matrix der Dimension $n \times n$. Für quadratische Matrizen

ist aus der linearen Algebra bekannt, dass genau dann eine eindeutige Lösung von (2.2) existiert, wenn die Matrix invertierbar ist. Wir werden in diesem Kapitel immer annehmen, dass dies der Fall sei.

In den Übungen werden Beispiele von Problemen aus den Wirtschaftswissenschaften und der Physik behandelt, die auf die Lösung linearer Gleichungssysteme führen. Hier werden wir uns mit neuronalen Netzen beschäftigen, deren numerische Behandlung ebenfalls auf die Lösung eines linearen Gleichungssystems führt.

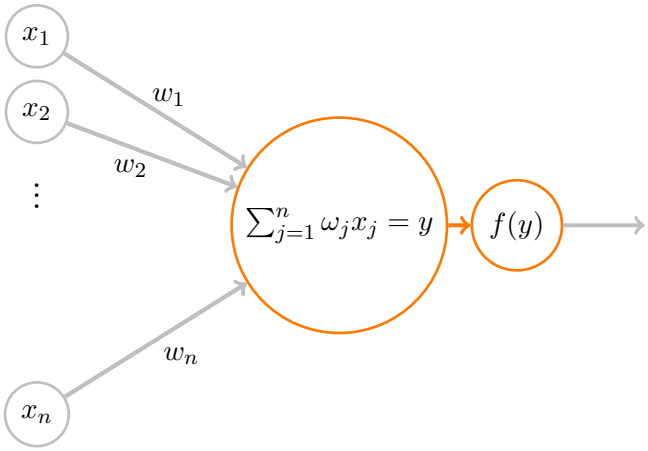
2.1 Anwendung: Ausgleichsrechnung

Das erste unserer Anwendungsbeispiele ist für viele praktische Zwecke besonders wichtig, weswegen wir es etwas genauer untersuchen wollen. Zur Motivation betrachten wir *neuronale Netze*[1] wie sie im Kurs *Maschninelles Lernen*[2] von Prof. Dr. H.K. Büning und Dr. U. Bubeck oder [6, Seiten 146 und 147] behandelt werden und einen für die Naturwissenschaften typischen Versuchsaufbau. Beide Problemstellungen führen jeweils auf das *Ausgleichsproblem*.

Neuronale Netze modellieren komplexe Systeme und sind der Funktionsweise des Gehirns nachempfunden. Dabei werden Synapsen (Knoten / Eingänge x_j, $j \in \{1, 2, \ldots, n\}$, $n \in \mathbb{N}$) in einer vereinfachten Darstellung einer Nervenzelle über Dendriten (Kanten) mit einem Zellkörper verbunden. Die Synapsen geben elektrische Impulse unterschiedlicher Stärke (Gewichte ω_j, $j \in \{1, 2, \ldots, n\}$) ab, die im Zellkörper aufsummiert werden. Falls eine (bestimmte) Reizschwelle $\theta \in \mathbb{R}$ überschritten wird, leitet der Zellkörper ein Signal über das Axon (Ausgang) weiter. Dazu dient die Entscheidungsfunktion

$$f\left(\sum_{j=1}^{n} \omega_j x_j\right) = \begin{cases} 0 & \text{falls } \sum_{j=1}^{n} \omega_j x_j < \theta \\ 1 & \text{falls } \sum_{j=1}^{n} \omega_j x_j \geq \theta \end{cases}.$$

Dieses Modell von Rosenblatts Perzeptron [1958] kann wie folgt visualisiert werden (einschichtiges *feedforward*-Netz):



[1] Siehe auch *http://de.wikipedia.org/wiki/Künstliches_neuronales_Netz*.

[2] Die Vorlesungsunterlagen zur entsprechenden Veranstaltung finden Sie unter *http://www.cs.uni-paderborn.de/fachgebiete/fg-kleine-buening/lehre/maschinelles-lernen.html*.

So kann ein neuronales Netz beispielsweise dazu verwendet werden, bestimmte Eigenschaften der Eingabewerte x_j, $j \in \{1, 2, \ldots, n\}$, zu erkennen. Charakterisierend für das Perzeptron Modell sind die Gewichte ω_j, $j \in \{1, 2, \ldots, n\}$, die folglich so zu bestimmen sind, dass das Netz die gesuchte Problemlösung finden und angeben kann. Zur Bestimmung kann in komplizierten Fällen ein Lernalgorithmus angewendet werden. In jedem Fall werden Tests $x^{(i)} = (x_1^{(i)}\ x_2^{(i)}\ \ldots\ x_n^{(i)})$, $i \in \{1, 2, \ldots, m\}$, $m \in \mathbb{N}$, mit bekannten Ergebnissen $y^{(i)}$, $i \in \{1, 2, \ldots, m\}$, benötigt, um die Gewichte zu bestimmen. Diese sollten so gewählt werden, dass $\sum_{j=1}^{n} \omega_j x_j^{(i)} - y^{(i)}$ für alle $i \in \{1, 2, \ldots, m\}$ möglichst klein wird, also $\omega = (\omega_1, \omega_2, \ldots, \omega_n)^T$ das (lineare) Gleichungssystem

$$\underbrace{\begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & & & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}}_{\widetilde{A} :=} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \tag{2.4}$$

möglichst gut löst. Beachte, dass typischerweise mehr Tests durchgeführt werden als Gewichte zu bestimmen sind ($m \gg n$), das lineare Gleichungssystem (2.4) also im Allgemeinen überbestimmt und somit nicht exakt lösbar ist.

In den Naturwissenschaften werden häufig Experimente in Abhängigkeit verschiedener Eingabewerte $t_1, t_2, \ldots, t_m$ durchgeführt, deren Ergebnisse in Form von Messwerten $z_1, z_2, \ldots, z_m$ festgehalten werden. Auf Grund von theoretischen Überlegungen (z.B. auf Grund eines zugrundeliegenden physikalischen Gesetzes), kennt man eine Funktion $f(t)$, für die $f(t_i) = z_i$ gelten sollte. Diese Funktion wiederum hängt aber nun von unbekannten Parametern $x_1, \ldots, x_n$ ab; wir schreiben $f(t; x)$ für $x = (x_1, \ldots, x_n)^T$, um dies zu betonen. Z.B. könnte $f(t; x)$ durch

$$f(t; x) = x_1 + x_2 t \qquad \text{oder} \qquad f(t; x) = x_1 + x_2 t + x_3 t^2$$

gegeben sein. Im ersten Fall beschreibt die gesuchte Funktion eine Gerade, im zweiten eine (verallgemeinerte) Parabel. Wenn wir annehmen, dass die Funktion f das Experiment wirklich exakt beschreibt und keine Messfehler vorliegen, so könnten wir die Parameter x_i durch Lösen des (im Allgemeinen nichtlinearen) Gleichungssystems

$$\begin{aligned} f(t_1; x) &= z_1 \\ &\vdots \\ f(t_k; x) &= z_k \end{aligned} \tag{2.5}$$

nach x bestimmen. In vielen praktischen Fällen ist dieses Gleichungssystem linear, so z.B. in den zwei obigen Beispielen, in denen sich (2.5) zu $\widetilde{A}x = z$ mit

$$\widetilde{A} = \begin{pmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{pmatrix} \quad \text{bzw.} \quad \widetilde{A} = \begin{pmatrix} 1 & t_1 & t_1^2 \\ \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 \end{pmatrix} \qquad \text{und} \qquad z = \begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix}$$

ergibt. Diese linearen Gleichungssysteme haben m Gleichungen – eine für jedes Wertepaar (t_i, z_i) – und n Unbekannte (nämlich gerade die unbekannten Parameter x_i), wobei m üblicherweise sehr viel größer als n ist. Man sagt, dass das Gleichungssystem *überbestimmt* ist.

Da Messwerte eines Versuchs praktisch immer mit Fehlern behaftet sind, ist es sicherlich zu optimistisch, anzunehmen, dass das Gleichungssystem $\widetilde{A}x = z$ lösbar ist – überbestimmte Gleichungssysteme haben oft keine Lösung!

Statt also den (vermutlich vergeblichen) Versuch zu machen, eine exakte Lösung x dieses Systems zu finden, wollen wir probieren, eine möglichst gute Näherungslösung zu finden, das heißt wenn $\widetilde{A}x = z$ nicht lösbar ist, wollen wir zumindest ein x finden, so dass $\widetilde{A}x$ möglichst nahe bei z liegt.[3] Dazu müssen wir ein Kriterium für „möglichst nahe“ wählen, das sowohl sinnvoll ist als auch eine einfache Lösung zulässt. Hier bietet sich das sogenannte *Ausgleichsproblem* (auch *Methode der kleinsten Quadrate* genannt) an:

$$\text{Finde } x = (x_1, \ldots, x_n)^T, \text{ so dass } \quad \varphi(x) := \|\widetilde{A}x - z\|^2 \quad \text{minimal wird.}$$

Hierbei bezeichnet $\|y\|$ die euklidische Norm eines Vektors $y \in \mathbb{R}^m$, also

$$\|y\| = \sqrt{\sum_{i=1}^{m} y_i^2}.$$

Um die Funktion φ zu minimieren, setzten wir den Gradienten $\nabla\varphi(x)$ gleich Null. Beachte dazu, dass der Gradient der Funktion $g(x) := \|f(x)\|^2$ für beliebiges $f : \mathbb{R}^n \to \mathbb{R}^m$ durch $\nabla g(x) = 2Df(x)^T f(x)$ gegeben ist. Wir erhalten also

$$\nabla\varphi(x) = 2\widetilde{A}^T\widetilde{A}x - 2\widetilde{A}^T z.$$

Falls $\widetilde{A}$ vollen Spaltenrang besitzt, ist die zweite Ableitung $D^2\varphi(x) = 2\widetilde{A}^T\widetilde{A}$ positiv definit, womit jede Nullstelle des Gradienten $\nabla\varphi$ ein Minimum von φ ist. Folglich minimiert ein Vektor x die Funktion φ genau dann, wenn die sogenannten *Normalengleichungen* $\widetilde{A}^T\widetilde{A}x = \widetilde{A}^T z$ erfüllt sind. Das Ausgleichsproblem wird also wie folgt gelöst:

$$\text{löse } \quad Ax = b \quad \text{mit } \quad A = \widetilde{A}^T\widetilde{A} \text{ und } b = \widetilde{A}^T z.$$

Das zunächst recht kompliziert scheinende Minimierungsproblem für φ wird also auf die Lösung eines linearen Gleichungssystems zurückgeführt.

Neben der Ausgleichsrechnung gibt es viele weitere Anwendungen in der Numerik, die auf die Lösung eines linearen Gleichungssystems führen. Einige davon werden uns im weiteren Verlauf dieser Vorlesung noch begegnen, z.B. die *Interpolation* von Punkten mittels *Splines*.

2.2 Das Gauß'sche Eliminationsverfahren

Wir werden nun ein erstes Verfahren zur Lösung linearer Gleichungssysteme kennen lernen. Das *Gauß'sche Eliminationsverfahren* ist ein sehr anschauliches Verfahren, das zudem recht leicht implementiert werden kann. Es beruht auf der einfachen Tatsache, dass ein lineares

[3] In der Notation der neuronalen Netze heißt das: Lösung ω für $\widetilde{A}\omega = y$. Im Folgenden verwenden wir die für lineare Gleichungssysteme üblichere Notation mit x und z.

Gleichungssystem $Ax = b$ leicht lösbar ist, falls die Matrix A in *oberer Dreiecksform* vorliegt, das heißt

$$A = \begin{pmatrix} a_{1\,1} & a_{1\,2} & \dots & a_{1\,n} \\ 0 & a_{2\,2} & \dots & a_{2\,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{n\,n} \end{pmatrix}.$$

In diesem Fall kann man $Ax = b$ leicht mittels der rekursiven Vorschrift

$$x_n = \frac{b_n}{a_{n\,n}}, \qquad x_{n-1} = \frac{b_{n-1} - a_{n-1\,n}x_n}{a_{n-1\,n-1}}, \qquad \dots, \qquad x_1 = \frac{b_1 - a_{1\,2}x_2 - \dots - a_{1\,n}x_n}{a_{1\,1}}$$

oder, kompakt geschrieben,

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{i\,j}x_j}{a_{i\,i}}, \qquad i = n,\, n-1, \dots,\, 1 \tag{2.6}$$

(mit der Konvention $\sum_{j=n+1}^{n} a_{i\,j}x_j = 0$) lösen. Dieses Verfahren wird als *Rückwärtseinsetzen* bezeichnet.

Die Idee des Gauß'schen Eliminationsverfahrens liegt nun darin, das Gleichungssystem $Ax = b$ in ein Gleichungssystem $\widetilde{A}x = \tilde{b}$ umzuformen, so dass die Matrix $\widetilde{A}$ in oberer Dreiecksform vorliegt. Wir wollen dieses Verfahren zunächst an einem Beispiel veranschaulichen.

Beispiel 2.1 Gegeben sei das lineare Gleichungssystem (2.2) mit

$$A = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 9 & 6 \\ 2 & 3 & 4 \end{pmatrix} \qquad \text{und} \qquad b = \begin{pmatrix} 29 \\ 43 \\ 20 \end{pmatrix}.$$

Um die Matrix A auf obere Dreiecksgestalt zu bringen, müssen wir die drei Einträge 7, 2 und 3 unterhalb der Diagonalen auf 0 bringen, also *eliminieren*. Wir beginnen mit $a_{3\,3} = 2$. Hierzu subtrahieren wir 2-mal die erste Zeile von der letzten und erhalten so

$$A_1 = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 9 & 6 \\ 0 & -7 & -8 \end{pmatrix}$$

Das Gleiche machen wir mit dem Vektor b. Dies liefert

$$b_1 = \begin{pmatrix} 29 \\ 43 \\ -38 \end{pmatrix}.$$

Nun fahren wir fort mit der 7: Wir subtrahieren 7-mal die erste Zeile von der zweiten, sowohl in A_1 als auch in b_1, und erhalten

$$A_2 = \begin{pmatrix} 1 & 5 & 6 \\ 0 & -26 & -36 \\ 0 & -7 & -8 \end{pmatrix} \qquad \text{und} \qquad b_2 = \begin{pmatrix} 29 \\ -160 \\ -38 \end{pmatrix}.$$

Im dritten Schritt *eliminieren* wir die -7, die jetzt an der Stelle der 3 steht, indem wir $7/26$-mal die zweite Zeile von der dritten subtrahieren:

$$A_3 = \begin{pmatrix} 1 & 5 & 6 \\ 0 & -26 & -36 \\ 0 & 0 & \frac{22}{13} \end{pmatrix} \qquad \text{und} \qquad b_3 = \begin{pmatrix} 29 \\ -160 \\ \frac{66}{13} \end{pmatrix}.$$

Hiermit sind wir fertig und setzen $\widetilde{A} = A_3$ und $\tilde{b} = b_3$. Rückwärtseinsetzen gemäß (2.6) liefert dann

$$x_3 = \frac{\frac{66}{13}}{\frac{22}{13}} = 3, \qquad x_2 = \frac{-160 - 3 \cdot (-36)}{-26} = 2 \qquad \text{und} \qquad x_1 = \frac{29 - 2 \cdot 5 - 3 \cdot 6}{1} = 1.$$

□

Wir formulieren den Algorithmus nun für allgemeine quadratische Matrizen A.

Algorithmus 2.2 (Gauß-Elimination, Grundversion)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n \times n}$ und ein Vektor $b \in \mathbb{R}^n$.

Für j von 1 bis $n-1$ (j: Spaltenindex des zu eliminierenden Eintrags)

Für i von n bis $j+1$ (rückwärts zählend)
(i: Zeilenindex des zu eliminierenden Eintrags)
Subtrahiere a_{ij}/a_{jj}-mal die j-te Zeile von der i-ten Zeile:

Setze $\alpha := a_{ij}/a_{jj}$ und berechne

$$a_{ik} := a_{ik} - \alpha\, a_{jk} \text{ für } k = j, \ldots, n, \quad b_i := b_i - \alpha\, b_j$$

Ende der i-Schleife

Ende der j-Schleife

□

Es kann passieren, dass dieser Algorithmus zu keinem Ergebnis führt, obwohl das Gleichungssystem lösbar ist. Der Grund dafür liegt in der Division durch das Diagonalelement a_{jj}, siehe Definition von α in Algorithmus 2.2. Hierbei wurde stillschweigend angenommen, dass dieses ungleich Null ist, was aber im Allgemeinen nicht der Fall sein muss. Glücklicherweise gibt es eine Möglichkeit, dieses Problem zu beheben:

Nehmen wir an, dass wir a_{ij} für gegebene Indizes i und j *eliminieren* möchten und $a_{jj} = 0$ ist. Nun gibt es zwei Möglichkeiten: Im ersten Fall ist $a_{ij} = 0$. In diesem Fall brauchen wir nichts zu tun, da das Element a_{ij}, das auf 0 gebracht werden soll, bereits gleich 0 ist. Im zweiten Fall gilt $a_{ij} \neq 0$. In diesem Fall können wir die i-te und die j-te Zeile der Matrix A sowie die entsprechenden Einträge im Vektor b vertauschen, wodurch wir die gewünschte Eigenschaft $a_{ij} = 0$ erreichen, nun allerdings nicht durch Elimination sondern durch Vertauschung. Dieses Verfahren nennt man *Pivotierung* und der folgende Algorithmus bringt nun tatsächlich jedes lineare Gleichungssystem in Dreiecksform.

Algorithmus 2.3 (Gauß-Elimination mit Pivotierung)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

Für j von 1 bis $n-1$ (j: Spaltenindex des zu eliminierenden Eintrags)

Falls $a_{j\,j} = 0$, wähle ein $p \in \{j+1, \ldots, n\}$ mit $a_{p\,j} \neq 0$ und
vertausche $a_{j\,k}$ und $a_{p\,k}$ für $k = j, \ldots, n$ sowie b_j und b_p

Für i von n bis $j+1$ (rückwärts zählend)
(i: Zeilenindex des zu eliminierenden Eintrags)
Subtrahiere $a_{i\,j}/a_{j\,j}$-mal die j-te Zeile von der i-ten Zeile:

Setze $\alpha := a_{i\,j}/a_{j\,j}$ und berechne

$$a_{i\,k} := a_{i\,k} - \alpha\, a_{j\,k} \text{ für } k = j, \ldots, n, \quad b_i := b_i - \alpha\, b_j$$

Ende der i-Schleife

Ende der j-Schleife

□

Das Element $a_{i\,j}$, das vor Beginn der i-Schleife mit $a_{j\,j}$ getauscht wird, nennt man *Pivotelement*. Wir werden später i Abschnitt 2.4 eine Strategie kennen lernen, bei der – auch wenn $a_{j\,j} \neq 0$ ist – gezielt ein Element $a_{k\,j} \neq 0$ als Pivotelement ausgewählt wird, mit dem man ein besseres Robustheitsverhalten des Algorithmus' erhalten kann.

2.3 LR-Faktorisierung

In der obigen Version des Gauß-Verfahrens haben wir die Matrix A auf obere Dreiecksform gebracht und zugleich alle dafür notwendigen Operationen auch auf den Vektor b angewendet. Es gibt alternative Möglichkeiten, lineare Gleichungssysteme zu lösen, bei denen der Vektor b unverändert bleibt. Wir werden nun ein Verfahren kennenlernen, bei dem die Matrix A in ein Produkt von zwei Matrizen L und R zerlegt wird, also $A = LR$ gilt, wobei R in oberer Dreiecksform und L in *unterer Dreiecksform*

$$L = \begin{pmatrix} l_{1\,1} & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ l_{n-1\,1} & \ldots & l_{n-1\,n-1} & 0 \\ l_{n\,1} & l_{n\,2} & \ldots & l_{n\,n} \end{pmatrix}.$$

vorliegt. Die Zerlegung $A = LR$ wird als *LR-Faktorisierung* oder *LR-Zerlegung* bezeichnet.

Um $Ax = b$ zu lösen, kann man dann $LRx = b$ wie folgt in zwei Schritten lösen: Zunächst löst man das Gleichungssystem $Ly = b$. Dies kann, ganz analog zum Rückwärtseinsetzen (2.6) durch textitVorwärtseinsetzen geschehen:

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{i\,j} y_j}{l_{i\,i}}, \qquad i = 1, 2, \ldots, n. \tag{2.7}$$

Im zweiten Schritt löst man dann durch Rückwärtseinsetzen das Gleichungssystem $Rx = y$. Dann gilt

$$Ax = LRx = Ly = b,$$

womit das gewünschte System $Ax = b$ gelöst ist.

Das Gauß'sche Eliminationsverfahren lässt sich so erweitern, dass damit – bis auf die Zeilenvertauschungen in A – eine LR-Zerlegung einer invertierbaren Matrix A möglich ist:

Die Subtraktion „i-te Zeile – α-mal j-te Zeile“ im m-ten Schritt des Algorithmus lässt sich durch Multiplikation von links mit der Matrix

$$F_m = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \ddots & & & \\ & -\alpha & & \ddots & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix} \quad \text{mit} \quad F_m^{-1} = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \ddots & & & \\ & \alpha & & \ddots & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}$$

erzielen, hierbei steht α in der i-ten Zeile und j-ten Spalte.

Wenn sich die Gauß-Elimination ohne Zeilenvertauschung durchführen lässt, so werden nur Operationen mit solchen Matrizen benötigt. Wenn man die dabei benötigten Matrizen F_1, $F_2, \ldots,$ F_k zu einer Matrix $F = F_k \cdot F_{k-1} \cdots F_1$ aufmultipliziert, so erhält man $A = LR$ mit $L = F^{-1} = F_1^{-1} \cdot F_2^{-1} \cdots F_k^{-1}$, also die LR-Faktorisierung (beachte, dass die Matrix F^{-1} als Produkt unterer Dreiecksmatrizen wieder eine untere Dreiecksmatrix ist).

Sind allerdings wegen der Pivotierung Zeilenvertauschungen nötig, so lässt sich i.A. keine LR-Zerlegung erhalten.[4] Die Vertauschung der i-ten und der j-ten Zeile in A im m-ten Schritt des Algorithmus lässt sich durch die Multiplikation von links mit Matrizen der Form

$$P_m = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & 1 & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & 1 & & & 0 & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix}$$

erreichen, wobei die 0-Einträge gerade an der i-ten und j-ten Stelle stehen. Sind solche Zeilenvertauschungen notwendig, so kann man die benötigten Matrizen $P_1, \ldots, P_l$ ebenfalls zu einer Matrix $P = P_l \cdots P_1$ aufmultiplizieren (P nennt man *Permutationsmatrix*).

[4] tatsächlich existiert für allgemeine invertierbare Matrizen im Allgemeinen keine LR-Zerlegung

Diese Permutationen müssen ggf. bei der Konstruktion der Matrix L berücksichtigt werden. Um das entsprechende Vorgehen zu erklären, verwenden wir die folgende Notation:

Es sei A_m die Matrix, die nach dem m-ten Schritt des Algorithmus erzeugt wurde (also $A_0 = A$ und $A_k = R$, wenn k die Gesamtanzahl der Schritte im Algorithmus ist). Es sei $P^{(m)} = P_m \cdots P_1$ die aufmultiplizierte Matrix der bis zum Schritt m angefallenen Permutationen, wobei wir zur einfacheren Zählung $P_l = \mathrm{Id}$ (Einheitsmatrix) setzen, falls keine Permutation durchgeführt wurde. Schließlich sei $L^{(m)}$ die untere Dreiecksmatrix mit

$$P^{(m)} A = L^{(m)} A_m. \tag{2.8}$$

Falls bis zum m-ten Schritt keine Permutation durchgeführt wurde, gilt wie oben $L^{(m)} = F_1^{-1} \cdot F_2^{-1} \cdots F_m^{-1}$.

Nehmen wir an, dass vor der Durchführung des $(m+1)$-ten Schritts eine Pivotierung durchgeführt wird, bei der die i-te mit der j-ten Zeile vertauscht wird. Bezeichnen wir die zugehörige Permutationsmatrix mit P_{m+1}, so wird die Matrix A_m in (2.8) also verändert zu $\widetilde{A}_m = P_{m+1} A_m$. Aus der Konstruktion von $P^{(m)}$ folgt zudem $P^{(m+1)} = P_{m+1} P^{(m)}$. Damit (2.8) nach der Pivotierung weiterhin gültig ist, muss die Matrix $L^{(m)}$ also durch eine Matrix $\widetilde{L}^{(m)}$ ersetzt werden, so dass die Gleichung

$$P^{(m+1)} A = \widetilde{L}^{(m)} \widetilde{A}_m$$

gilt. Aus (2.8) folgt nun

$$P^{(m+1)} A = P_{m+1} P^{(m)} A = P_{m+1} L^{(m)} A_m = P_{m+1} L^{(m)} P_{m+1}^{-1} \widetilde{A}_m$$

und wir erhalten $\widetilde{L}^{(m)} = P_{m+1} L^{(m)} P_{m+1}^{-1}$.

Aus der Form von P_{m+1} folgt $P_{m+1}^{-1} = P_{m+1}$. Die Multiplikation mit P_{m+1} von rechts bewirkt daher gerade die Vertauschung der i-ten und der j-ten Spalte in der Matrix $L^{(m)}$. Die Matrix $\widetilde{L}^{(m)}$ entsteht also, indem wir erst die i-te und j-te Zeile und dann die i-te und j-te Spalte in $L^{(m)}$ vertauschen.

Beachte dabei, dass $L^{(m)}$ auf Grund der Form der Matrizen F_l^{-1} stets 1-Einträge auf der Diagonalen besitzt und darüberhinaus Einträge ungleich Null nur unterhalb der Diagonalen und nur in den Spalten $1, \ldots, j-1$ besitzt. Weil im Algorithmus zudem stets $i > j$ gilt, werden durch die Zeilen- und Spaltenvertauschungen daher gerade die Elemente l_{ik} und l_{jk} für $k = 1, \ldots, j-1$ vertauscht. Insbesondere ist daher auch $\widetilde{L}^{(m)}$ wieder eine untere Dreiecksmatrix.

Details zur Implementierung dieses Algorithmus werden in den Übungen besprochen. Sie finden sich darüberhinaus z.B. in den Büchern von Deuflhard/Hohmann [3], Schwarz/Köckler [7] oder Stoer [8].

2.4 Fehlerabschätzungen und Kondition

Wie bereits im einführenden Kapitel erläutert, können Computer nicht alle reellen Zahlen darstellen, weswegen alle Zahlen intern gerundet werden, damit sie in die endliche Menge der *maschinendarstellbaren Zahlen* passen. Hierdurch entstehen *Rundungsfehler*. Selbst

wenn sowohl die Eingabewerte als auch das Ergebnis eines Algorithmus maschinendarstellbare Zahlen sind, können solche Fehler auftreten, denn auch die (möglicherweise nicht darstellbaren) Zwischenergebnisse eines Algorithmus werden gerundet. Auf Grund dieser Fehler aber auch wegen Eingabe- bzw. Messfehlern in den vorliegenden Daten oder Fehlern aus vorhergehenden numerischen Rechnungen wird durch einen Algorithmus üblicherweise nicht die exakte Lösung x des linearen Gleichungssystems

$$Ax = b$$

berechnet, sondern eine Näherungslösung $\tilde{x}$. Um dies formal zu fassen, führt man ein *benachbartes* oder *gestörtes* Gleichungssystems

$$A\tilde{x} = b + \Delta b$$

ein, für das $\tilde{x}$ gerade die exakte Lösung ist. Der Vektor Δb heißt hierbei das *Residuum* oder der *Defekt* der Näherungslösung $\tilde{x}$. Den Vektor $\Delta x = \tilde{x} - x$ nennen wir den *Fehler* der Näherungslösung $\tilde{x}$. Da Rundung und andere Fehlerquellen i.A. nur kleine Fehler bewirken, ist es gerechtfertigt anzunehmen, dass $\|\Delta b\|$ „klein“ ist. Das Ziel dieses Abschnitts ist es nun, aus der Größe des Residuums $\|\Delta b\|$ auf die Größe des Fehlers $\|\Delta x\|$ zu schließen. Insbesondere wollen wir untersuchen, wie *sensibel* die Größe $\|\Delta x\|$ von $\|\Delta b\|$ abhängt, das heißt ob kleine Residuen $\|\Delta b\|$ große Fehler $\|\Delta x\|$ hervorrufen können. Diese Analyse ist unabhängig von dem verwendeten Lösungsverfahren, da wir hier nur das Gleichungssystem selbst und kein explizites Verfahren betrachten.

Um solch eine Analyse durchzuführen, brauchen wir das Konzept der *Matrixnorm*. Man kann Matrixnormen ganz allgemein definieren; für unsere Zwecke reicht aber der Begriff der *induzierten Matrixnorm* aus. Wir erinnern zunächst an verschiedene Vektornormen für Vektoren $x \in \mathbb{R}^n$. In dieser Vorlesung verwenden wir üblicherweise die *euklidische Norm* oder 2-*Norm*

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2},$$

welche wir meistens einfach mit $\|x\|$ bezeichnen. Weitere Normen sind die 1-*Norm*

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

und die *Maximums-* oder ∞-*Norm*

$$\|x\|_\infty = \max_{i=1,\ldots,n} |x_i|.$$

Wir werden gleich sehen, dass die zwei letzten Normen im Zusammenhang mit linearen Gleichungssystemen gewisse Vorteile haben.

Für alle Normen im $\mathbb{R}^n$ kann man eine zugehörige *induzierte Matrixnorm* definieren.

Definition 2.4 Sei $\mathbb{R}^{n\times n}$ die Menge der $(n \times n)$-Matrizen und sei $\|\cdot\|_p$ eine Vektornorm im $\mathbb{R}^n$. Dann definieren wir für $A \in \mathbb{R}^{n\times n}$ die zu $\|\cdot\|_p$ gehörige *induzierte Matrixnorm* $\|A\|_p$ als

$$\|A\|_p := \sup_{x\in\mathbb{R}^n\setminus\{0\}} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\substack{x\in\mathbb{R}^n \\ \|x\|_p=1}} \|Ax\|_p.$$

□

Die Gleichheit der beiden Ausdrücke auf der rechten Seite folgt dabei aus der Beziehung $\|\alpha x\|_p = |\alpha| \, \|x\|_p$ für $\alpha \in \mathbb{R}$. Dass es sich hierbei tatsächlich um Normen auf dem Vektorraum $\mathbb{R}^{n\times n}$ handelt, wird in den Übungen bewiesen.

Da im Allgemeinen keine Verwechslungsgefahr besteht, bezeichnen wir die Vektornormen und die von ihnen induzierten Matrixnormen mit dem gleichen Symbol.

Satz 2.5 Für die zu den obigen Vektornormen gehörigen induzierten Matrixnormen und $A \in \mathbb{R}^{n\times n}$ gelten die folgenden Gleichungen

$$\begin{aligned}
\|A\|_1 &= \max_{j=1,\ldots,n} \sum_{i=1}^n |a_{i\,j}| && \text{(Spaltensummennorm)}\\
\|A\|_\infty &= \max_{i=1,\ldots,n} \sum_{j=1}^n |a_{i\,j}| && \text{(Zeilensummennorm)}\\
\|A\|_2 &= \sqrt{\rho(A^TA)} && \text{(Spektralnorm),}
\end{aligned}$$

wobei $\rho(A^TA)$ den maximalen Eigenwert der symmetrischen und positiv definiten Matrix A^TA bezeichnet.[5]

Beweis: Wir beweisen die Gleichungen, indem wir die entsprechenden Ungleichungen beweisen.

„$\|A\|_1$“: Für einen beliebigen Vektor $x \in \mathbb{R}^n$ gilt

$$\|Ax\|_1 = \sum_{i=1}^n \left| \sum_{j=1}^n a_{i\,j} x_j \right| \le \sum_{i=1}^n \sum_{j=1}^n |a_{i\,j}| \, |x_j| = \sum_{j=1}^n |x_j| \sum_{i=1}^n |a_{i\,j}|.$$

Sei nun j^* der Index, für den die innere Summe maximal wird, also

$$\sum_{i=1}^n |a_{i\,j^*}| = \max_{j=1,\ldots,n} \sum_{i=1}^n |a_{i\,j}|.$$

Dann gilt für Vektoren mit $\|x\|_1 = 1$

$$\sum_{j=1}^n |x_j| \sum_{i=1}^n |a_{i\,j}| \le \underbrace{\sum_{j=1}^n |x_j|}_{=1} \sum_{i=1}^n |a_{i\,j^*}| = \sum_{i=1}^n |a_{i\,j^*}|,$$

woraus, da x beliebig war, die Ungleichung

$$\|A\|_1 \le \sum_{i=1}^n |a_{i\,j^*}| = \max_{j=1,\ldots,n} \sum_{i=1}^n |a_{i\,j}|$$

folgt. Andererseits gilt für den j^*-ten Einheitsvektor e_{j^*}

$$\|A\|_1 \ge \|Ae_{j^*}\|_1 = \sum_{i=1}^n \underbrace{\left| \sum_{j=1}^n a_{i\,j} [e_{j^*}]_j \right|}_{=|a_{i\,j^*}|} = \sum_{i=1}^n |a_{i\,j^*}| = \max_{j=1,\ldots,n} \sum_{i=1}^n |a_{i\,j}|$$

und damit die behauptete Gleichung.

„$\|A\|_\infty$“: Ähnlich wie für die 1-Norm mit $x^* = (\pm 1, \ldots, \pm 1)^T$ an Stelle von e_{j^*}.

„$\|A\|_2$“: Da A^TA symmetrisch ist, können wir eine Orthonormalbasis von Eigenvektoren $v_1, \ldots, v_n$ wählen, also $\|v_i\|_2 = 1$ und $\langle v_i, v_j \rangle = 0$ für $i \ne j$. Sei nun $x \in \mathbb{R}^n$ ein beliebiger Vektor mit Länge 1, dann lässt sich x als

[5] Sofern A vollen Rang besitzt; ansonsten ist A^TA nur positiv semi-definit.

Linearkombination $x = \sum_{i=1}^n \mu_i v_i$ mit $\sum_{i=1}^n \mu_i^2 = 1$ schreiben. Seien λ_i die zu den v_i gehörigen Eigenwerte von A^TA und sei λ_{i^*} der maximale Eigenwert, also $\lambda_{i^*} = \rho(A^TA)$. Dann gilt

$$\begin{aligned}\|Ax\|_2^2 &= \langle Ax, Ax\rangle = \langle A^TAx, x\rangle = \sum_{i,j=1}^n \mu_i\mu_j \langle \underbrace{A^TAv_i}_{=\lambda_i v_i}, v_j\rangle \\ &= \sum_{\substack{i,j=1\\ i\neq j}}^n \mu_i\mu_j\lambda_i \underbrace{\langle v_i, v_j\rangle}_{=0} + \sum_{i=1}^n \mu_i^2\lambda_i \underbrace{\langle v_i, v_i\rangle}_{=1} = \sum_{i=1}^n \mu_i^2\lambda_i.\end{aligned}$$

Damit folgt

$$\|Ax\|_2^2 = \sum_{i=1}^n \mu_i^2\lambda_i \leq \underbrace{\sum_{i=1}^n \mu_i^2}_{=1}\lambda_{i^*} = \lambda_{i^*},$$

also, da x beliebig war, auch

$$\|A\|_2^2 \leq \lambda_{i^*} = \rho(A^TA).$$

Andererseits gilt die Ungleichung

$$\|A\|_2^2 \geq \|Av_{i^*}\|_2^2 = \langle A^TAv_{i^*}, v_{i^*}\rangle = \lambda_{i^*}\underbrace{\langle v_{i^*}, v_{i^*}\rangle}_{=1} = \lambda_{i^*} = \rho(A^TA).$$

□

Für jede Matrixnorm können wir die zugehörige *Kondition* einer invertierbaren Matrix definieren.

Definition 2.6 Für eine gegebene Matrixnorm $\|\cdot\|_p$ ist die *Kondition* einer invertierbaren Matrix A (bzgl. $\|\cdot\|_p$) definiert durch

$$\mathrm{cond}_p(A) := \|A\|_p\|A^{-1}\|_p.$$

□

Wenn wir das Verhältnis zwischen dem Fehler Δx und dem Residuum Δb betrachten, können wir entweder die *absoluten Größen* dieser Werte, also $\|\Delta x\|_p$ und $\|\Delta b\|_p$, oder, was oft sinnvoller ist, die *relativen Größen* $\|\Delta x\|_p/\|x\|_p$ und $\|\Delta b\|_p/\|b\|_p$ betrachten. Der folgende Satz zeigt, wie man den Fehler durch das Residuum abschätzen kann.

Satz 2.7 Sei $\|\cdot\|_p$ eine Vektornorm mit zugehöriger (und gleich bezeichneter) induzierter Matrixnorm. Sei $A \in \mathbb{R}^{n\times n}$ eine gegebene invertierbare Matrix und b, $\Delta b \in \mathbb{R}^n$ gegebene Vektoren. Seien x, $\tilde{x} \in \mathbb{R}^n$ die Lösungen der linearen Gleichungssysteme

$$Ax = b \qquad \text{und} \qquad A\tilde{x} = b + \Delta b.$$

Dann gelten für den Fehler $\Delta x = \tilde{x} - x$ die Abschätzungen

$$\|\Delta x\|_p \leq \|A^{-1}\|_p\|\Delta b\|_p \tag{2.9}$$

und

$$\frac{\|\Delta x\|_p}{\|x\|_p} \leq \mathrm{cond}_p(A)\frac{\|\Delta b\|_p}{\|b\|_p}. \tag{2.10}$$

Beweis: Seien $C \in \mathbb{R}^{n\times n}$ und $y \in \mathbb{R}^n$ eine beliebige Matrix und ein beliebiger Vektor. Dann gilt gemäß Übung

$$\|Cy\|_p \leq \|C\|_p \|y\|_p. \tag{2.11}$$

Schreiben wir $\tilde{x} = x + \Delta x$ und ziehen die Gleichung

$$Ax = b$$

von der Gleichung

$$A(x + \Delta x) = b + \Delta b$$

ab, so erhalten wir

$$A\Delta x = \Delta b.$$

Weil A invertierbar ist, können wir die Gleichung auf beiden Seiten von links mit A^{-1} multiplizieren und erhalten so

$$\Delta x = A^{-1}\Delta b.$$

Daraus folgt

$$\|\Delta x\|_p = \|A^{-1}\Delta b\|_p \leq \|A^{-1}\|_p \|\Delta b\|_p,$$

wobei wir (2.11) mit $C = A^{-1}$ und $y = \Delta b$ benutzt haben. Dies zeigt (2.9).

Aus (2.11) mit $C = A$ und $y = x$ folgt

$$\|b\|_p = \|Ax\|_p \leq \|A\|_p \|x\|_p,$$

und damit

$$\frac{1}{\|x\|_p} \leq \frac{\|A\|_p}{\|b\|_p}.$$

Wenn wir erst diese Ungleichung und dann (2.9) anwenden, erhalten wir

$$\frac{\|\Delta x\|_p}{\|x\|_p} \leq \frac{\|A\|_p \|\Delta x\|_p}{\|b\|_p} \leq \frac{\|A\|_p \|A^{-1}\|_p \|\Delta b\|_p}{\|b\|_p} = \mathrm{cond}_p(A) \frac{\|\Delta b\|_p}{\|b\|_p},$$

also die Abschätzung (2.10). □

Für Matrizen, deren Kondition $\mathrm{cond}_p(A)$ groß ist, können sich kleine Fehler im Vektor b (bzw. Rundungsfehler im Verfahren) zu großen Fehlern im Ergebnis x verstärken. Man spricht in diesem Fall von *schlecht konditionierten* Matrizen. Das folgende Beispiel illustriert diesen Sachverhalt.

Beispiel 2.8 Betrachte das Gleichungssystem $Ax = b$ mit

$$A = \begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix} \qquad \text{und} \qquad b = \begin{pmatrix} 0.001 \\ 1 \end{pmatrix}.$$

Durch Nachrechnen sieht man leicht, dass die Lösung gegeben ist durch $x = (0.001, 0)^T$. Ebenso überprüft man leicht, dass

$$A^{-1} = \begin{pmatrix} 1 & 0 \\ -1000 & 1 \end{pmatrix}$$

gilt. In der Zeilensummennorm gilt also $\|A\|_\infty = 1001$ und $\|A^{-1}\|_\infty = 1001$ und damit

$$\mathrm{cond}_\infty(A) = 1001 \cdot 1001 \approx 1\,000\,000.$$

Für die gestörte rechte Seite $\tilde{b} = (0.002, 1)^T$, das heißt für $\Delta b = (0.001, 0)^T$ erhält man die Lösung $\tilde{x} = (0.002, -1)^T$. Es gilt also $\Delta x = (0.001, -1)^T$ und damit

$$\frac{\|\Delta x\|_\infty}{\|x\|_\infty} = \frac{1}{0.001} = 1000 \qquad \text{und} \qquad \frac{\|\Delta b\|_\infty}{\|b\|_\infty} = \frac{0.001}{1} = 0.001.$$

Der relative Fehler 0.001 in $\tilde{b}$ wird also um den Faktor 1 000 000 zu dem relativen Fehler 1000 in $\tilde{x}$ vergrößert, was fast exakt der Kondition $\mathrm{cond}_\infty(A)$ entspricht. □

Bei schlecht konditionierten Matrizen können sich Rundungsfehler im Algorithmus ähnlich stark wie Fehler in der rechten Seite b auswirken. Ein wichtiges Kriterium beim Entwurf eines Lösungsverfahrens ist es, dass das Verfahren auch für schlecht konditionierte Matrizen noch zuverlässig funktioniert. Beim Gauß-Verfahren kann man dazu die Auswahl der Pivotelemente so gestalten, dass möglichst wenig Rundungsfehler auftreten.

Hierzu muss man sich überlegen, welche Operationen in der Gauß-Elimination besonders fehleranfällig sind; dies kann man sehr ausführlich und formal durchführen, wir werden uns hier aber auf ein eher heuristisches Kriterium beschränken: Die Rechenoperationen in der Gauß-Elimination sind gegeben durch die Subtraktionen

$$a_{i\,k} - \frac{a_{i\,j}}{a_{j\,j}} a_{j\,k}, k \in \{j, j+1, \ldots, n\}, \qquad \text{und} \qquad b_i - \frac{a_{i\,j}}{a_{j\,j}} b_j.$$

Im Prinzip können wir im Pivotierungsschritt des Algorithmus eine beliebige andere Zeile mit der j-ten Zeile vertauschen, solange diese die gleiche Anzahl von führenden Nulleinträgen besitzt, was gerade für die Zeilen $j, \ldots, n$ gilt. Dies gibt uns die Freiheit, den *Zeilenindex* „j“ durch einen beliebigen Index $p \in \{j, \ldots, n\}$ zu ersetzen, was im Algorithmus durch Tauschen der j-ten mit der p-ten Zeile realisiert wird. Beachte, dass das „j“ in $a_{i\,j}$ der *Spaltenindex* des zu eliminierenden Elements ist, der sich durch die Vertauschung nicht ändert; wir können also durch Zeilenvertauschung nur die Elemente $a_{j\,j}$, $a_{j\,k}$ und b_j oder anders gesagt gerade die Brüche $a_{j\,k}/a_{j\,j}$ und $b_j/a_{j\,j}$ beeinflussen.

Die wesentliche Quelle für Rundungsfehler in einer Subtraktion „$c - d$“ im Computer entsteht, wenn die zu berechnende Differenz betragsmäßig klein ist und die einzelnen Terme c und d im Vergleich dazu betragsmäßig groß sind. Um dies zu veranschaulichen nehmen wir an, dass wir im Dezimalsystem auf 5 Stellen genau rechnen. Wenn wir nun die Zahl $1,234$ von der Zahl $1,235$ subtrahieren, so erhalten wir das richtige Ergebnis $0,001$, wenn wir aber die Zahl $1000,234$ von der Zahl $1000,235$ subtrahieren, so wird nach interner Rundung auf 5 Stellen die Rechnung $1000,2 - 1000,2 = 0$ ausgeführt, was zu einem deutlichen Fehler führt (dieser spezielle Fehler wird auch „Auslöschung“ gennant). Die Strategie, solche Fehler in der Gauß-Elimination so weit wie möglich zu vermeiden, besteht nun darin, den Zeilenindex p bei der Pivotierung so auszuwählen, dass die zu subtrahierenden Ausdrücke betragsmäßig klein sind, ein Verfahren, das man *Pivotsuche* nennt. Da wir nur die Brüche $a_{j\,k}/a_{j\,j}$ $(k = j, \ldots, n)$ und $b_j/a_{j\,j}$ beeinflussen können, sollte man p dazu also so wählen, dass eben diese Brüche im Betrag möglichst klein werden; diese Strategie wird im folgenden Algorithmus 2.9 umgesetzt.

Algorithmus 2.9 (Gauß-Elimination mit Pivotsuche)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

Für j von 1 bis $n-1$ (j: Spaltenindex des zu eliminierenden Eintrags)

Wähle aus den Zeilenindizes $p \in \{j, \ldots, n \,|\, a_{p\,j} \neq 0\}$ denjenigen aus, für den der Ausdruck

$$K(p) = \max\left\{ \max_{k=j,\ldots,n} \frac{|a_{p\,k}|}{|a_{p\,j}|}, \frac{|b_p|}{|a_{p\,j}|} \right\}$$

minimal wird. Falls $p \neq j$ vertausche $a_{j\,k}$ und $a_{p\,k}$ für $k = j, \ldots, n$ sowie b_p und b_j

Für i von n bis $j+1$ (rückwärts zählend)
(i: Zeilenindex des zu eliminierenden Eintrags)
Subtrahiere $a_{i\,j}/a_{j\,j}$-mal die j-te Zeile von der i-ten Zeile:

Setze $\alpha := a_{i\,j}/a_{j\,j}$ und berechne

$$a_{i\,k} := a_{i\,k} - \alpha\, a_{j\,k} \text{ für } k = j, \ldots, n, \quad b_i := b_i - \alpha\, b_j$$

Ende der i-Schleife

Ende der j-Schleife

□

Die hier verwendete Form der Pivotsuche wird *Spaltenpivotsuche* genannt, da in jedem Schritt innerhalb der j-ten Spalte nach dem besten Pivotelement $a_{p\,j}$ gesucht wird. Eine erweiterte Form ist die *vollständige* oder *totale Pivotsuche* bei der auch in den Zeilen gesucht wird und dann gegebenenfalls auch Spaltenvertauschungen durchgeführt werden. Gute Implementierungen der Gauß-Elimination verwenden immer solche Pivotsuchmethoden. Diese bietet eine Verbesserung der Robustheit aber keinen vollständigen Schutz gegen große Fehler bei schlechter Kondition – aus prinzipiellen mathematischen Gründen, wie wir im nächsten Abschnitt näher erläutern werden.

Eine allgemeinere Strategie zur Vermeidung schlechter Kondition ist die *Präkonditionierung*, bei der eine Matrix $P \in \mathbb{R}^{n\times n}$ gesucht wird, für so dass die Kondition von PA kleiner als die von A ist. Es wird dann das besser konditionierte Problem $PAx = Pb$ gelöst. Eine weitere Strategie zur Behandlung schlecht konditionierter Gleichungssysteme, die wir nun genauer untersuchen wollen, ist die QR-Faktorisierung (oder QR-Zerlegung) einer Matrix.

2.5 QR-Faktorisierung

Die LR-Zerlegung, die explizit oder implizit Grundlage der bisher betrachteten Lösungsverfahren war, hat unter Konditions-Gesichtspunkten einen wesentlichen Nachteil: Es kann nämlich passieren, das die einzelnen Matrizen L und R der Zerlegung deutlich größere Kondition haben als die zerlegte Matrix A.

Beispiel 2.10 Betrachte die Matrix

$$A = \begin{pmatrix} 0.001 & 0.001 \\ 1 & 2 \end{pmatrix} \qquad \text{mit} \qquad A^{-1} = \begin{pmatrix} 2000 & -1 \\ -1000 & 1 \end{pmatrix}$$

und LR-Faktorisierung

$$L = \begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} 0.001 & 0.001 \\ 0 & 1 \end{pmatrix}.$$

Wegen

$$L^{-1} = \begin{pmatrix} 1 & 0 \\ -1000 & 1 \end{pmatrix} \qquad \text{und} \qquad R^{-1} = \begin{pmatrix} 1000 & -1 \\ 0 & 1 \end{pmatrix}$$

gilt

$$\text{cond}_\infty(A) \approx 6000, \ \ \text{cond}_\infty(L) \approx 1000000 \ \text{ und } \ \text{cond}_\infty(R) \approx 1000.$$

Die ∞-Kondition von L ist also etwa 166-mal so groß wie die von A. □

Selbst wenn wir eventuelle Fehler in der Berechnung von R und L vernachlässigen oder z.B. durch geschickte Pivotsuche vermindern, kann die schlechte Konditionierung von R und L durch die Verstärkung der beim Rückwärts- und Vorwärtseinsetzen auftretenden Rundungsfehler zu großen Fehlern Δx führen, besonders wenn die Matrix A selbst bereits schlecht konditioniert ist. Bei der LR-Faktorisierung kann es also passieren, dass die Kondition der Teilprobleme, die im Algorithmus auftreten, deutlich schlechter ist als die des Ausgangsproblems. Beachte, dass die Kondition des Ausgangsproblems nur von der Problemstellung abhängt, die der Teilprobleme aber von dem verwendeten Algorithmus, weswegen diese auch als *numerische Kondition* bezeichnet wird.

Um die numerische Kondition zu verringern, wollen wir nun eine andere Form der Zerlegung betrachten, bei der die Kondition der einzelnen Teilprobleme (bzw. der zugehörigen Matrizen) nicht größer ist als die des Ausgangsproblems (also der Ausgangsmatrix A).

Hierzu verwenden wir die folgenden Matrizen.

Definition 2.11 Eine Matrix $Q \in \mathbb{R}^{n\times n}$ heißt *orthogonal*, falls $QQ^T = \text{Id}$ ist.[6] □

Unser Ziel ist nun ein Algorithmus, mit dem eine Matrix A in ein Produkt QR zerlegt wird, wobei Q eine orthogonale Matrix ist und R eine obere Dreiecksmatrix. Offenbar ist ein Gleichungssystem der Form $Qy = b$ leicht zu lösen, indem man die Matrixmultiplikation $y = Q^T b$ durchführt. Deswegen kann man das Gleichungssystem $Ax = b$ wie bei der LR-Faktorisierung leicht durch Lösen der Teilsysteme $Qy = b$ und $Rx = y$ lösen.

Bevor wir den entsprechenden Algorithmus herleiten, wollen wir beweisen, dass bei dieser Form der Zerlegung die Kondition tatsächlich erhalten bleibt – zumindest für die euklidische Norm.

[6] Das komplexe Gegenstück hierzu sind die unitären Matrizen, mit denen sich all das, was wir hier im Reellen machen, auch im Komplexen durchführen lässt

Satz 2.12 Sei $A \in \mathbb{R}^{n\times n}$ eine invertierbare Matrix mit QR-Zerlegung. Dann gilt

$$\mathrm{cond}_2(Q) = 1 \qquad \text{und} \qquad \mathrm{cond}_2(R) = \mathrm{cond}_2(A).$$

Beweis: Da Q orthogonal ist, gilt $Q^{-1} = Q^T$. Daraus folgt für beliebige Vektoren $x \in \mathbb{R}^n$

$$\|Qx\|_2^2 = \langle Qx, Qx\rangle_2 = \langle Q^TQx, x\rangle_2 = \langle x, x\rangle_2 = \|x\|_2^2,$$

also auch $\|Qx\|_2 = \|x\|_2$. Damit folgt für invertierbare Matrizen $B \in \mathbb{R}^{n\times n}$

$$\|QB\|_2 = \max_{\|x\|_2=1} \|QBx\|_2 = \max_{\|x\|_2=1} \|Q(Bx)\|_2 = \max_{\|x\|_2=1} \|Bx\|_2 = \|B\|_2$$

und mit $Qx = y$ auch

$$\|BQ\|_2 = \max_{\|x\|_2=1} \|BQx\|_2 = \max_{\|Q^Ty\|_2=1} \|By\|_2 = \max_{\|y\|_2=1} \|By\|_2 = \|B\|_2,$$

da mit Q auch $Q^T = Q^{-1}$ orthogonal ist. Also folgen

$$\mathrm{cond}_2(Q) = \|Q\|_2\|Q^{-1}\|_2 = \|Q\mathrm{Id}\|_2\|Q^{-1}\mathrm{Id}\|_2 = \|\mathrm{Id}\|_2\|\mathrm{Id}\|_2 = 1$$

und

$$\begin{aligned}\mathrm{cond}_2(R) &= \mathrm{cond}_2(Q^TA) = \|Q^TA\|_2\|(Q^TA)^{-1}\|_2 = \|Q^TA\|_2\|A^{-1}Q\|_2\\ &= \|A\|_2\|A^{-1}\|_2 = \mathrm{cond}_2(A).\end{aligned}$$

□

Zwar gilt dieser Satz für andere Matrixnormen nicht, da für je zwei Vektornormen aber Abschätzungen der Form $\|x\|_p \le C_{p,q}\|x\|_q$ gelten, ist zumindest eine extreme Verschlechterung der numerischen Kondition auch bezüglich anderer induzierter Matrixnormen ausgeschlossen.

Beispiel 2.13 Als Beispiel betrachten wir noch einmal das Gleichungssystem aus Beispiel 2.10. Eine QR-Zerlegung von A ist gegeben durch

$$\begin{aligned}Q &= \frac{1+\sqrt{1000001}}{1000001+\sqrt{1000001}}\begin{pmatrix}-1 & -1000\\ -1000 & 1\end{pmatrix} \approx \begin{pmatrix}-0.001 & -1\\ -1 & 0.001\end{pmatrix},\\ R &= \frac{1+\sqrt{1000001}}{1000001+\sqrt{1000001}}\begin{pmatrix}-1000.001 & -2000.001\\ 0 & 1\end{pmatrix} \approx \begin{pmatrix}-1 & -2\\ 0 & 0.001\end{pmatrix}\end{aligned}$$

mit

$$Q^{-1} \approx \begin{pmatrix}-0.001 & -1\\ -1 & 0.001\end{pmatrix} \qquad \text{und} \qquad R^{-1} \approx \begin{pmatrix}-1 & -2000\\ 0 & 1000\end{pmatrix}.$$

Hier gilt also $\mathrm{cond}_\infty(Q) \approx 1$ und $\mathrm{cond}_\infty(R) \approx 6000$, das heißt es tritt auch bzgl. der ∞-Kondition keine wesentliche Verschlechterung der Kondition auf. □

Die Idee der Algorithmen zur QR-Zerlegung liegt nun darin, die Spalten der Matrix A als Vektoren aufzufassen und durch orthogonale Transformationen auf die gewünschte Form zu bringen. Orthogonale Transformationen sind dabei gerade die linearen Transformationen, die sich durch orthogonale Matrizen darstellen lassen. Geometrisch sind dies die Transformationen, die die (euklidische) Länge des transformierten Vektors sowie den Winkel zwischen zwei Vektoren unverändert lassen – nichts anderes haben wir im Beweis von Satz 2.12 ausgenutzt.

Zur Realisierung eines solchen Algorithmus bieten sich zwei mögliche Transformationen an: Drehungen und Spiegelungen. Wir wollen hier den nach seinem Erfinder benannten *Householder-Algorithmus* herleiten, der auf Basis von Spiegelungen funktioniert.[7] Wir veranschaulichen die Idee zunächst geometrisch: Sei $a_{\star j} \in \mathbb{R}^n$ die j-te Spalte der Matrix A. Wir wollen eine Spiegelung $H^{(j)}$ finden, die $a_{\star j}$ auf einen Vektor der Form

$$a^{(j)}_{\star j} = (\underbrace{*, *, \ldots, *}_{j \text{ Stellen}}, 0, \ldots, 0)^T$$

bringt, ihn also in den linearen Unterraum $E_j = \operatorname{span}(e_1, \ldots, e_j)$ spiegelt.

Um diese Spiegelung zu konstruieren, betrachten wir allgemeine Spiegelmatrizen der Form

$$H = H(v) = \mathrm{Id} - \frac{2vv^T}{v^T v}$$

wobei $v \in \mathbb{R}^n$ ein beliebiger Vektor ist (beachte, dass dann $vv^T \in \mathbb{R}^{n\times n}$ und $v^T v \in \mathbb{R}$ sind). Diese Matrizen heißen nach dem Erfinder des zugehörigen Algorithmus *Householder-Matrizen*. Offenbar ist H symmetrisch und es gilt

$$HH^T = H^2 = \mathrm{Id} - \frac{4vv^T}{v^T v} + \frac{2vv^T}{v^T v}\frac{2vv^T}{v^T v} = \mathrm{Id},$$

also Orthogonalität. Geometrisch entspricht die Multiplikation mit H der Spiegelung an der Ebene mit Normalenvektor $n = v/\|v\|$.

Um nun die gewünschte Spiegelung in die Ebene E_j zu realisieren, muss man v geeignet wählen. Dabei hilft das folgende Lemma.

Lemma 2.14 Betrachte einen Vektor $w = (w_1, \ldots, w_n)^T \in \mathbb{R}^n$. Für einen gegebenen Index $j \in \{1, \ldots, n\}$ betrachte

$$\begin{aligned} c &= \operatorname{sgn}(w_j)\sqrt{w_j^2 + w_{j+1}^2 + \cdots + w_n^2} \in \mathbb{R} \\ v &= (0, \ldots, 0, c + w_j, w_{j+1}, \ldots, w_n)^T \\ H &= \mathrm{Id} - \frac{2vv^T}{v^T v} \end{aligned}$$

mit den Konventionen $\operatorname{sgn}(a) = 1$, falls $a \geq 0$, und $\operatorname{sgn}(a) = -1$, falls $a < 0$, sowie $H = \mathrm{Id}$ für $v = 0$. Dann gilt

$$Hw = (w_1, w_2, \ldots, w_{j-1}, -c, 0, \ldots, 0)^T.$$

Darüberhinaus gilt für jeden Vektor $z \in \mathbb{R}^n$ der Form $z = (z_1, \ldots, z_{j-1}, 0, \ldots, 0)$ die Gleichung $Hz = z$.

[7] Ein Algorithmus auf Basis von Drehungen ist der sogenannte Givens-Algorithmus.

Beweis: Falls $v \neq 0$ ist, gilt

$$\begin{aligned}\frac{2v^Tw}{v^Tv} &= \frac{2((c+w_j)w_j + w_{j+1}^2 + \cdots + w_n^2)}{c^2 + 2cw_j + w_j^2 + w_{j+1}^2 + \cdots + w_n^2} \\ &= \frac{2(cw_j + w_j^2 + w_{j+1}^2 + \cdots + w_n^2)}{2cw_j + 2w_j^2 + 2w_{j+1}^2 + \cdots + 2w_n^2} = 1.\end{aligned}$$

Daraus folgt

$$Hw = w - v\frac{2v^Tw}{v^Tv} = w - v = \begin{pmatrix} w_1 \\ \vdots \\ w_{j-1} \\ w_j \\ w_{j+1} \\ \vdots \\ w_n \end{pmatrix} - \begin{pmatrix} 0 \\ \vdots \\ 0 \\ c+w_j \\ w_{j+1} \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_{j-1} \\ -c \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Falls $v = 0$ ist, sieht man sofort, dass $w_{j+1} = \ldots = w_n = 0$ gelten muss. Damit folgt $c = w_j$, also $w_j + c = 2w_j$, weswegen auch $w_j = 0$ sein muss. In diesem Fall gilt also bereits $w_j = w_{j+1} = \ldots = w_n = 0$, so dass die Behauptung mit $H = \mathrm{Id}$ folgt.

Für die zweite Behauptung verwenden wir, dass für Vektoren z der angegebenen Form die Gleichung $v^Tz = 0$ gilt, woraus sofort

$$Hz = z - v\frac{2v^Tz}{v^Tv} = z,$$

also die Behauptung, folgt. □

Die Idee des Algorithmus liegt nun nahe:

Wir setzen $A^{(1)} = A$ und konstruieren im ersten Schritt $H^{(1)}$ gemäß Lemma 2.14 mit $j = 1$ und $w = a^{(1)}_{\star 1}$, der ersten Spalte der Matrix $A^{(1)}$. Damit ist dann $A^{(2)} = H^{(1)}A^{(1)}$ von der Form

$$A^{(2)} = H^{(1)}A^{(1)} = \begin{pmatrix} a^{(2)}_{11} & a^{(2)}_{12} & \cdots & a^{(2)}_{1n} \\ 0 & a^{(2)}_{22} & \cdots & a^{(2)}_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & a^{(2)}_{n2} & \cdots & a^{(2)}_{nn} \end{pmatrix}.$$

Im zweiten Schritt konstruieren wir $H^{(2)}$ gemäß Lemma 2.14 mit $j = 2$ und $w = a^{(2)}_{\star 2}$, der zweiten Spalte der Matrix $A^{(2)}$. Da die erste Spalte der Matrix $A^{(2)}$ die Voraussetzungen an den Vektor z in Lemma 2.14 erfüllt, folgt die Form

$$A^{(3)} = H^{(2)}A^{(2)} = \begin{pmatrix} a^{(3)}_{11} & a^{(3)}_{12} & a^{(3)}_{13} & \cdots & a^{(3)}_{1n} \\ 0 & a^{(3)}_{22} & a^{(3)}_{23} & \cdots & a^{(3)}_{2n} \\ 0 & 0 & a^{(3)}_{33} & \cdots & a^{(3)}_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a^{(3)}_{n3} & \cdots & a^{(3)}_{nn} \end{pmatrix}.$$

Wenn wir sukzessive fortfahren, erhalten wir nach $n-1$ Schritten die gewünschte QR-Zerlegung mit

$$Q^T = H^{(n-1)} \cdots H^{(1)} \qquad \text{und} \qquad R = A^{(n)},$$

denn es gilt

$$\begin{aligned} QR &= {H^{(1)}}^T \cdots {H^{(n-2)}}^T {H^{(n-1)}}^T A^{(n)} \\ &= {H^{(1)}}^T \cdots {H^{(n-2)}}^T A^{(n-1)} \\ &\vdots \\ &= {H^{(1)}}^T A^{(2)} \\ &= A^{(1)} = A \end{aligned}$$

Insgesamt ergibt sich der folgende Ablaufplan für die QR-Zerlegung.[8]

Algorithmus 2.15 [QR-Zerlegung]
Gegeben: Matrix $A \in \mathbb{R}^{n \times n}$

Setze $A^{(1)} = A$ und $Q^T = Id$.
Für j von 1 bis $n-1$ (j: Spaltenindex)

(1) Konstruiere $H^{(j)}$ gemäß Lemma 2.14 mit $w = a^{(j)}_{\star j}$ (j-te Spalte von $A^{(j)}$)

(2) Berechne $A^{(j+1)} = H^{(j)} A^{(j)}$

(3) Setze $Q^T = H^{(j)} Q^T$

Ende der j-Schleife

Ausgabe: Matrix $R = A^{(n)}$ in oberer Dreiecksform und Matrix $Q^T \in \mathbb{R}^{n \times n}$ □

Beachte, dass die QR-Faktorisierung *immer* funktioniert, auch wenn A nicht invertierbar ist, denn die obigen Überlegungen liefern einen konstruktiven Beweis für die Existenz. Die resultierende Matrix Q ist immer invertierbar, die Matrix R ist invertierbar genau dann, wenn A invertierbar ist.

[8] Für eine **Implementierung** sei auf [4, Algorithmus 2.15] verwiesen.

Bei der Lösung eines linearen Gleichungssystems sollte die Invertierbarkeit von R vor dem Rückwärtseinsetzen getestet werden (eine obere Dreiecksmatrix ist genau dann invertierbar, wenn alle Diagonalelemente ungleich Null sind). Dies kann schon im obigen Algorithmus geschehen, indem überprüft wird, ob die c-Werte (die gerade die Diagonalelemente von R bilden) ungleich Null sind.

Die QR-Zerlegung kann tatsächlich mehr als nur lineare Gleichungssysteme lösen: Wir haben in Abschnitt 2.1 das lineare Ausgleichsproblem kennengelernt, bei dem $x \in \mathbb{R}^n$ gesucht ist, so dass der Vektor

$$r = \widetilde{A}x - z$$

minimale 2-Norm $\|r\|_2$ hat, und haben gesehen, dass dieses Problem durch Lösen der Normalengleichungen $\widetilde{A}^T\widetilde{A}x = \widetilde{A}^T z$ gelöst werden kann. Jedoch kann gerade die Matrix $\widetilde{A}^T\widetilde{A}$ (bedingt durch ihre Struktur, vgl. Übung) eine sehr große Kondition haben, so dass es hier erstrebenswert ist,

(a) ein robustes Verfahren zu verwenden und

(b) die explizite Lösung der Normalengleichungen zu vermeiden.

Mit dem QR-Algorithmus kann man beides erreichen, man kann nämlich das Ausgleichsproblem *direkt* lösen.

Die QR-Zerlegung (und auch der obige Algorithmus) kann auf die nichtquadratische Matrix $\widetilde{A}$ mit n Spalten und $m > n$ Zeilen angewendet werden. Das Resultat ist dann eine Faktorisierung $\widetilde{A} = QR$ mit

$$R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$$

wobei $R_1 \in \mathbb{R}^{n\times n}$ eine obere Dreiecksmatrix ist. Beachte, dass die Normalengleichungen genau dann lösbar sind, wenn $\widetilde{A}$ vollen Spaltenrang besitzt, was wir annehmen. In diesem Fall ist auch die Matrix R_1 invertierbar.

Wenn man dann x so wählt, dass der Vektor

$$s = Q^T r = Q^T\widetilde{A}x - Q^T z$$

minimale 2-Norm besitzt, dann hat auch r minimale 2-Norm, da aus der Orthogonalität von Q^T die Gleichung $\|r\|_2 = \|s\|_2$ folgt. Wir zerlegen s in $s^1 = (s_1, \ldots, s_n)^T$ und $s^2 = (s_{n+1}, \ldots, s_m)^T$. Wegen der Form von $R = Q^T\widetilde{A}$ ist der Vektor s^2 unabhängig von x und wegen

$$\|s\|_2^2 = \sum_{i=1}^{m} s_i^2 = \sum_{i=1}^{n} s_i^2 + \sum_{i=n+1}^{m} s_i^2 = \|s^1\|_2^2 + \|s^2\|_2^2$$

wird die 2-Norm des Vektors s genau dann miminal, wenn $\|s^1\|_2^2$ minimal wird. Wir suchen also ein $x \in \mathbb{R}^n$, so dass

$$\|s^1\|_2 = \|R_1 x - y^1\|_2$$

minimal wird, wobei y^1 die ersten n Komponenten des Vektors $y = Q^T z$ bezeichnet. Da R_1 eine invertierbare obere Dreiecksmatrix ist, kann man durch Rückwärtseinsetzen eine Lösung x des Gleichungssystems $R_1 x = y^1$ finden, für die dann

$$\|s^1\|_2 = \|R_1 x - y^1\|_2 = 0$$

gilt, womit offenbar das Minimum erreicht wird. Zusammenfassend kann man also das Ausgleichsproblem wie folgt mit der QR-Faktorisierung direkt lösen:

Algorithmus 2.16 (Lösung des Ausgleichsproblems mit QR-Zerlegung)
Eingabe: Matrix $\widetilde{A} \in \mathbb{R}^{m\times n}$ mit $m > n$ und (maximalem) Spaltenrang n, Vektor $z \in \mathbb{R}^m$

(1) Berechne Zerlegung $\widetilde{A} = QR$ mit $R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$ und oberer Dreiecksmatrix $R_1 \in \mathbb{R}^{n\times n}$

(2) Löse das Gleichungssystem $R_1 x = y^1$ durch Rückwärtseinsetzen, wobei y^1 die ersten n Komponenten des Vektors $y = Q^T z \in \mathbb{R}^m$ bezeichnet

Ausgabe: Vektor $x \in \mathbb{R}^n$, für den $\|\widetilde{A}x - z\|_2$ minimal wird. □

Geometrisch passiert hier das Folgende: Das Bild von $\widetilde{A}$ wird durch die orthogonale Transformation Q^T längentreu auf den Unterraum $\mathrm{span}(e_1, \ldots, e_n)$ abgebildet, in dem wir dann das entstehende Gleichungssystem lösen können, vgl. Abbildung 2.1.

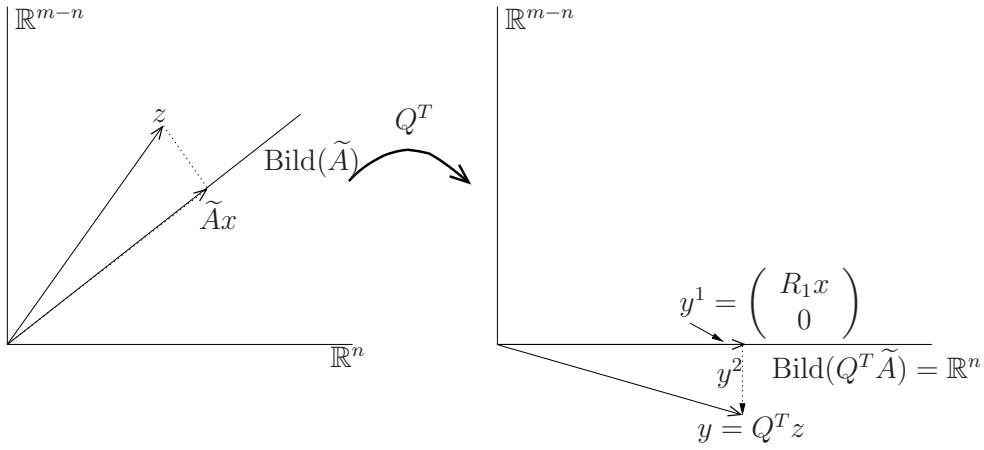


Abbildung 2.1: Veranschaulichung von Algorithmus 2.16

2.6 Aufwandsabschätzungen

Ein wichtiger Aspekt bei der Analyse numerischer Verfahren ist es zu untersuchen, wie lange diese Verfahren in der Regel benötigen, um zu dem gewünschten Ergebnis zu kommen. Da dies natürlich entscheidend von der Leistungsfähigkeit des verwendeten Computers abhängt, schätzt man nicht direkt die Zeit ab, sondern die Anzahl der Rechenoperationen, die ein Algorithmus benötigt. Da hierbei die *Gleitkommaoperationen*, also Addition, Multiplikation etc. von reellen Zahlen, die mit Abstand zeitintensivsten Operationen sind, beschränkt man sich in der Analyse üblicherweise auf diese.[9]

[9] Tatsächlich sind Multiplikation, Division und die Berechnung von Wurzeln etwas aufwändiger als Addition und Subtraktion, was wir hier aber vernachlässigen werden.

Die Verfahren, die wir bisher betrachtet haben, liefern nach endlich vielen Schritten ein Ergebnis (man spricht von *direkten Verfahren*), wobei die Anzahl der Operationen von der Dimension n der Matrix abhängt. Zur *Aufwandsabschätzung* genügt es also, die Anzahl der Gleitkommaoperationen (in Abhängigkeit von n) „abzuzählen“. Wie man dies am geschicktesten macht, hängt dabei von der Struktur des Algorithmus ab. Zudem muss man einige Rechenregeln aus der elementaren Analysis ausnutzen, um die entstehenden Ausdrücke zu vereinfachen. Speziell benötigen wir hier die Gleichungen

$$\sum_{i=1}^{n} i = \frac{(n+1)n}{2} \qquad \text{und} \qquad \sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n.$$

Wir beginnen mit dem **Rückwärtseinsetzen**, und betrachten zunächst die Multiplikationen und Divisionen: Für $i = n$ muss eine Division durchgeführt werden, für $i = n-1$ muss eine Multiplikation und eine Division durchgeführt werden, für $i = n-2$ müssen zwei Multiplikationen und eine Division durchgeführt werden, usw. So ergibt sich die Anzahl dieser Operationen als

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{(n+1)n}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

Für die Anzahl der Additionen und Subtraktionen zählt man ab

$$0 + 1 + 2 + \cdots + n - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

Insgesamt kommt man also auf

$$\frac{n^2}{2} + \frac{n}{2} + \frac{n^2}{2} - \frac{n}{2} = n^2$$

Gleitkommaoperationen. Da das **Vorwärtseinsetzen** völlig analog funktioniert, gilt dafür die gleiche Abschätzung.

Bei der **Gauß-Elimination** betrachten wir hier die Version aus Algorithmus 2.3 ohne Pivotsuche und nehmen den schlechtesten Fall an, nämlich dass $\alpha = a_{ij}/a_{jj} \neq 0$ für alle $i \in \{j+1, j+2, \ldots, n\}$ gilt und somit für jeden Zeilenindex i eine Elimination durchzuführen ist. Wir gehen spaltenweise vor und betrachten die Elemente, die für jedes j eliminiert werden müssen. Für jedes zu eliminierende Element in der j-ten Spalte benötigt man je $n - (j-1) + 1$ Additionen und Multiplikationen (die „+1“ ergibt sich aus der Operation für b) sowie eine Division zur Berechnung von α, das heißt $2(n+2-j)+1 = 2n+5-2j$ Operationen. In der j-ten Spalte müssen dabei $n-j$ Einträge eliminiert werden, nämlich für $i = n, \ldots, j+1$. Also ergeben sich für die j-te Spalte

$$(n-j)(2n+5-2j) = 2n^2 + 5n - 2nj - 2jn - 5j + 2j^2 = 2j^2 - (4n+5)j + 5n + 2n^2$$

Operationen. Dies muss für die Spalten $j = 1, \ldots, n-1$ durchgeführt werden, womit wir

auf

$$\begin{aligned}
&\sum_{j=1}^{n-1} \left(2j^2 - (4n+5)j + 5n + 2n^2\right) \\
&= 2\sum_{j=1}^{n-1} j^2 - (4n+5)\sum_{j=1}^{n-1} j + \left(5n + 2n^2\right)\sum_{j=1}^{n-1} 1 \\
&= \frac{2}{3}(n-1)^3 + (n-1)^2 + \frac{1}{3}(n-1) - (4n+5)\frac{(n-1)n}{2} + (n-1)(5n+2n^2) \\
&= \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{13}{6}n
\end{aligned}$$

Operationen kommen.

Zur vollständigen **Lösung eines linearen Gleichungssystems** müssen wir nun einfach die Operationen der Teilalgorithmen aufaddieren.

Für den Gauß-Algorithmus kommt man so auf

$$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{13}{6}n + n^2 = \frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{13}{6}n$$

Operationen. Analoge Überlegungen führen für die QR-Zerlegung [4, Algorithmus 2.15] auf

$$\frac{4}{3}n^3 + 3n^2 + \frac{14}{3}n - 9 + n^2 = \frac{4}{3}n^3 + 4n^2 + \frac{14}{3}n - 9$$

Operationen. Berücksichtigt man, dass für große n die führenden "n^3-Terme" dominant werden, so ergibt sich, dass die Gauß-Elimination etwa doppelt so schnell ist wie die QR-Faktorisierung.

Um einen Eindruck von den tatsächlichen Rechenzeiten zu bekommen, nehmen wir an, dass wir einen PC verwenden, der mit einem Core i7 Prozessor etwa eine Leistung von 33 GFLOPS (FLOPS = floating point operations per second; GFLOPS = GigaFLOPS = 10^9 Flops) schafft. Nehmen wir weiterhin (sehr optimistisch) an, dass wir Implementierungen der obigen Algorithmen haben, die diese Leistung optimal ausnutzen. Dann ergeben sich für $n \times n$ Gleichungssysteme die folgenden Rechenzeiten

n	Gauß	QR
1000	20.25 ms	40.49 ms
10000	20.21 s	40.41 s
100000	5.61 h	11.22 h
500000	29.23 d	58.46 d

Spätestens im Fall $n = 500\,000$ sind die Zeiten kaum mehr akzeptabel: Wer will schon mehr als vier Wochen auf ein Ergebnis warten?

Zum Abschluss dieses Abschnitts wollen wir noch ein gröberes Konzept der Aufwandsabschätzung einführen, das für praktische Zwecke oft ausreicht. Oft ist man nämlich nicht an der exakten Zahl der Operationen für ein gegebenes n interessiert, sondern nur an einer Abschätzung für große Dimensionen. Genauer möchte man wissen, wie schnell der Aufwand in Abhängigkeit von n wächst, das heißt wie er sich *asymptotisch* für $n \to \infty$ verhält. Man spricht dann von der *Ordnung* eines Algorithmus.

Definition 2.17 Ein Algorithmus hat die *Ordnung* $\mathcal{O}(n^q)$, wenn $q > 0$ die minimale Zahl ist, für die es eine Konstante $C > 0$ gibt, so dass der Algorithmus für alle $n \in \mathbb{N}$ weniger als Cn^q Operationen benötigt. □

Diese Zahl q ist aus den obigen Aufwandsberechnungen leicht abzulesen: Es ist gerade die höchste auftretende Potenz von n. Somit haben Vorwärts- und Rückwärtseinsetzen die Ordnung $\mathcal{O}(n^2)$, während Gauß- und QR-Verfahren die Ordnung $\mathcal{O}(n^3)$ besitzen.

Iterative Verfahren: Wir haben im letzten Abschnitt gesehen, dass die bisher betrachteten Verfahren – die sogenannten *direkten Verfahren* – die Ordnung $\mathcal{O}(n^3)$ besitzen: Wenn sich also n verzehnfacht, so vertausendfacht sich die Anzahl der Operationen und damit die Rechenzeit. Für große Gleichungssysteme mit mehreren 100 000 Unbekannten, die in der Praxis durchaus auftreten, führt dies wie oben gesehen zu unakzeptabelen Rechenzeiten.

Eine Klasse von Verfahren, die eine niedrigere Ordnung hat, sind die *iterativen Verfahren*. Allerdings zahlt man für den geringeren Aufwand einen Preis: Man kann bei diesen Verfahren nicht mehr erwarten, eine (bis auf Rundungsfehler) exakte Lösung zu erhalten, sondern muss von vornherein eine gewisse Ungenauigkeit im Ergebnis in Kauf nehmen.

Das Grundprinzip iterativer Verfahren funktioniert dabei wie folgt: Ausgehend von einem Startvektor $x^{(0)}$ berechnet man mittels einer Rechenvorschrift $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ iterativ eine Folge von Vektoren

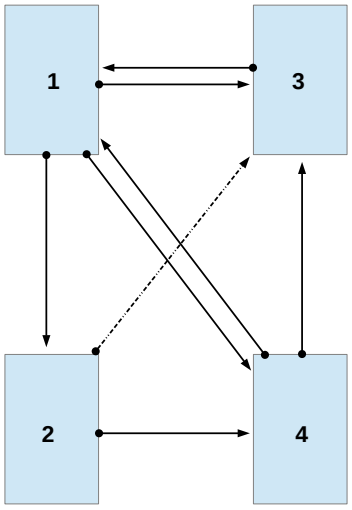
$$x^{(i+1)} = \Phi(x^{(i)}), \qquad i = 0, 1, 2, \ldots,$$

die für $i \to \infty$ gegen die Lösung x^* des Gleichungssystems $Ax = b$ konvergiert. Es gilt also $\lim_{i\to\infty} \|x^{(i)} - x^*\|_p = 0$. Wenn die gewünschte Genauigkeit erreicht ist, wird die Iteration abgebrochen und der letzte Wert $x^{(i)}$ als Näherung des Ergebnisses verwendet.

Kapitel 3

Eigenwerte und Eigenvektoren

Gemäß einer Aussage von Kurt Bryan muss ein Suchmaschinenanbieter, zum Beispiel *Google*, das Netz (*World Wide Web*) durchforsten, Information auf Webseiten sammeln und diese (in einem geeignetem Format) speichern.[1] Anschließend muss er in der Lage sein, auf Anfrage, die relevaten Links zu finden und sie in einer geeigneten Reihenfolge zu listen. Aber was ist eine *geeignete Reihenfolge*? Ein erster Ansatz die Relevanz (Wichtigkeit / *importance*) einer Seite in Bezug auf eine Suchanfrage einzuschätzen besteht darin, die Anzahl ihrer Rückverweise (*backlinks*) zu zählen; dies ist auch Teil des *Google PageRank* Algorithmus.



Bezeichne x_k die Anzahl der Seite k zeigenden Rückverweise. Dann ergibt sich für das links abgebildete *Web*:

- $x_1 = 2$
- $x_2 = 1$ (nahezu irrelevant)
- $x_3 = 3$ (Listenplatz 1)
- $x_4 = 2$

Aber sollte nicht ein Rückverweis von der Internetseite `www.spiegel.de` mehr Wert sein als einer von meiner Institutshomepage `www.tu-ilmenau.de/de/analysis/team/karl-worthmann`?

Diese Frage regt die folgende Verbesserung an: der Beitrag eines Rückverweises wird durch die Wichtigkeit x_j der verweisenden Seite dividiert durch die Anzahl n_j ihrer Rückverweise bestimmt, also mit dem Faktor x_j/n_j multipliziert. Dieser Ansatz liefert für das betrachtete Beispiel

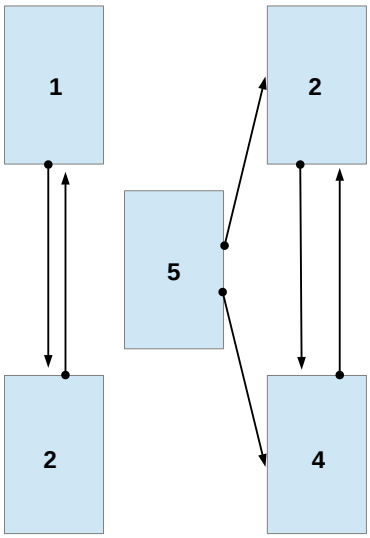
$$\begin{aligned} x_1 &= & & x_3/1 + x_4/2 \\ x_2 &= x_1/3 & & \\ x_3 &= x_1/3 + x_2/2 & & + x_4/2 \\ x_4 &= x_1/3 + x_2/2 & & \end{aligned}$$

[1] `http://www.rose-hulman.edu/~bryan/invprobs/usafagoogle.pdf`

beziehungsweise in Matrixschreibweise

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 0 & 1 & 1/2 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}}_{=:x}.$$

Es ist also das lineare Gleichungssystem $x = Ax$ zu lösen oder, etwas formaler ausgedrückt, ein Eigenvektor x der Matrix A zum Eigenwert 1 zu bestimmen. Eine Lösung ist beispielsweise $x \approx (0.387\ 0.129\ 0.290\ 0.194)^\top$ bzw. beliebige Vielfache dieses Eigenvektors; solch ein Eigenvektor existiert immer, falls A eine *stochastische Matrix* (quadratische Matrix, deren Spaltensummen Eins betragen und deren Elemente nichtnegativ sind) ohne sogenannte *dangling nodes* (Knoten ohne ausgehende Kante) ist. Gewünschte Eigenschaften dieses Eigenvektors wären gemäß Bryan, dass er nicht negativ (was ist negative Relevanz?), eindeutig bis auf Skalierung (eindimensionaler Eigenraum) sowie "leicht" zu berechnen ist – auch für sehr große Matrizen).



Dazu betrachten wir das Beispiel mit Verweismatrix

$$A = \left(\begin{array}{cc|ccc} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1/2 \\ 0 & 0 & 1 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right).$$

Die Matrix $A \in \mathbb{R}^{n\times n}$, $n = 5$, besitzt zwei linear unabhängige Eigenvektoren zum Eigenwert 1:

$x_1 = (0.5\ 0.5\ 0\ 0\ 0)^\top$ und $x_2 = (0\ 0\ 0.5\ 0.5\ 0)^\top$

Theorem: Besitzt ein Netz r Zusammenhangskomponenten, dann hat der von Eigenvektoren zum Eigenwert 1 aufgespannte Unterraum mindenstens Dimension r.

Als Ausweg kann man jeder Seite eine *schwache Verknüpfung* mit jeder anderen geben; A wird dann durch die gewichtete Kombination $M = (1-m)A + mS$ mit $m \in (0,1)$ und $S = (s_{ij})$, $s_{ij} = 1/n$ für alle $(i,j) \in \{1,2,\ldots,n\} \times \{1,2,\ldots,n\}$, ersetzt. Diese Matrix besitzt den dominanten Eigenwert 1, dessen Eigenraum Dimension 1 hat. Zudem existiert ein eindeutiger nichtnegativer Eigenvektor x mit $\sum_{i=1}^{n} x_i = 1$.[2] Folglich wird eine Methode benötigt, die es erlaubt einen Eigenvektor einer $(8.000.000.000 \times 8.000.000.000)$-Matrix zu finden. Für Details (auch bzgl. weiterer Anmerkungen zu diesem Beispiel) sei auf den sehr gut lesbaren Zeitschriftenbeitrag [2] verwiesen.

Tatsächlich spielen Eigenwerte und Eigenvektoren von Matrizen in vielen Anwendungen eine Rolle; so geben die Eigenwerte bei der Betrachtung iterativer Verfahren bspw. Auskunft über die Konvergenzgeschwindigkeit. Dies ist ein generelles Prinzip linearer Iterationen

[2] Der Suchmaschinenanbieter *Google* nutzt angeblich $m = 0.15$.

(ähnlich ist dies bei linearen Differentialgleichungen) und ein wichtiges Beispiel für eine Problemklasse, bei der Kenntnisse über die Eigenwerte einer Matrix von Interesse sind. Eine weitere Anwendung in der Bildverarbeitung wird in der Übung behandelt.

3.1 Eigenwertproblem & Kondition

Gesucht sind bei einem Eigenwertproblem diejenigen $\lambda \in \mathbb{C}\backslash\{0\}$, welche die Gleichung

$$Av = \lambda v$$

für einen *Eigenvektor* $v \in \mathbb{C}^n$ erfüllen. In vielen Anwendungen, zum Beispiel im oben beschriebenen Seitenranking von Google, ist man darüberhinaus an den zugehörigen Eigenvektoren v interessiert.

Wir werden in diesem relativ kurzen Kapitel einige Algorithmen zur Berechnung von Eigenwerten und zugehörigen Eigenvektoren für spezielle Matrizen, (z.B. symmetrische Matrizen) kennenlernen. Bevor wir zu numerischen Verfahren zur Berechnung von Eigenwerten kommen, wollen wir kurz die vielleicht naheliegendste Methode untersuchen, nämlich die Berechnung der Eigenwerte λ über die Nullstellen des charakteristischen Polynoms. Diese Methode ist numerisch äußerst schlecht konditioniert (unabhängig von der Kondition der Eigenwertberechnung selbst) und bereits kleinste Rundungsfehler können sehr große Fehler im Ergebnis nach sich ziehen. Als Beispiel betrachte man das Polynom

$$P(\lambda) = (\lambda - 1)(\lambda - 2)\cdots(\lambda - 20)$$

mit den Nullstellen $\lambda_i = i$ für $i = 1, \ldots, 20$.[3] Wenn dieses Polynom als charakteristisches Polynom einer Matrix berechnet wird, z.B. ist es gerade das charakteristische Polynom $\chi_A(\lambda)$ der Matrix $A = \mathrm{diag}(1, 2, \ldots, 20)$, liegt es üblicherweise nicht in der obigen "Nullstellen"-Darstellung sondern in anderer Form vor, z.B. ausmultipliziert. Wenn man das obige Polynom $P(\lambda)$ ausmultipliziert, ergeben sich Koeffizienten zwischen 1 (für λ^{20}) und $20! \approx 10^{20}$ (der konstante Term). Stört man nun den Koeffizienten vor λ^{19} (der den Wert 210 hat) mit dem sehr kleinen Wert $\varepsilon = 2^{-23} \approx 10^{-7}$, so erhält man die folgenden Nullstellen für das gestörte Polynom $\widetilde{P}(\lambda) = P(\lambda) - \varepsilon\lambda^{19}$:

$$\begin{aligned}
\lambda_1 &= 1.000\,000\,000 & \lambda_{10/11} &= 10.095\,266\,145 \pm 0.643\,500\,904\,i \\
\lambda_2 &= 2.000\,000\,000 & \lambda_{12/13} &= 11.793\,633\,881 \pm 1.652\,329\,728\,i \\
\lambda_3 &= 3.000\,000\,000 & \lambda_{14/15} &= 13.992\,358\,137 \pm 2.518\,830\,070\,i \\
\lambda_4 &= 4.000\,000\,000 & \lambda_{16/17} &= 16.730\,737\,466 \pm 2.812\,624\,894\,i \\
\lambda_5 &= 4.999\,999\,928 & \lambda_{18/19} &= 19.502\,439\,400 \pm 1.940\,330\,347\,i \\
\lambda_6 &= 6.000\,006\,944 & \lambda_{20} &= 20.846\,908\,101 \\
\lambda_7 &= 6.999\,697\,234 & & \\
\lambda_8 &= 8.007\,267\,603 & & \\
\lambda_9 &= 8.917\,250\,249 & &
\end{aligned}$$

[3] Das Beispiel stammt von dem englischen Numeriker James H. Wilkinson (1919–1986), der die Entdeckung dieses Polynoms angeblich als "the most traumatic experience in my career as a numerical analyst" bezeichnet hat.

Die winzige Störung bewirkt also beachtliche Fehler, insbesondere sind 10 Nullstellen durch die Störung komplex geworden.

Bevor wir mit konkreten Algorithmen beginnen, wollen wir vorab ein paar Überlegungen zur Kondition des Eigenwertproblems anstellen. Bei den linearen Gleichungssystemen haben wir den Fehler analysiert, indem wir das Residuum betrachtet haben, womit wir alle möglichen Fehlerquellen in die gestörte rechte Seite $b + \Delta b$ *verschoben* haben. Damit hängt der Fehler Δx linear vom Residuum Δb ab, was eine Fehlerbetrachtung allein über die Matrixnorm ermöglicht hat. Bei den Eigenwertproblemen ist dies nicht möglich, da ja nur die Einträge der Matrix als Problemdaten vorhanden sind. Wir müssten also explizit untersuchen, wie sich ein Eigenwert $\lambda_0(A)$ in Abhängigkeit von Änderungen in A verändert, also die Größe

$$\frac{|\lambda_0(A) - \lambda_0(A + \Delta A)|}{\|\Delta A\|}$$

berechnen. Da die Eigenwerte nichtlinear von der Matrix A abhängen, ist dieser Ausdruck im Allgemeinen schwer zu berechnen. Entsprechend sei für eine weitergehende Analyse auf [4, Kapitel 3] verwiesen.

3.2 Vektoriteration

Die einfachste Möglichkeit der Berechnung von Eigenwerten ist die Vektoriteration, die sich entweder als *direkte Iteration* (auch bekannt als *von Mises-Iteration* oder *power iteration*) oder als *inverse Iteration* (auch *inverse power iteration*) durchführen lässt.

Gegeben sei eine reelle Matrix $A \in \mathbb{R}^{n\times n}$. Die Idee der direkten Iteration beruht darauf, für einen beliebigen Startwert $x^{(0)} \in \mathbb{R}^n$ die Iteration

$$x^{(i+1)} = Ax^{(i)} \tag{3.1}$$

durchzuführen. Dass dieses einfache Verfahren tatsächlich unter gewissen Bedingungen einen Eigenwert liefert, zeigt der folgende Satz. Wir erinnern hierbei daran, dass symmetrische reelle Matrizen nur reelle Eigenwerte besitzen. Ferner bezeichne $\langle x, y\rangle = x^\top y = \sum_{i=1} x_i y_i$ das euklidische Skalarprodukt für Vektoren $x, y \in \mathbb{R}^n$, $n \in \mathbb{N}$.

Satz 3.1 Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix und $\lambda_1 = \lambda_1(A)$ ein einfacher Eigenwert, für den die Ungleichung

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$$

für alle anderen Eigenwerte $\lambda_j = \lambda_j(A)$ gilt. Sei weiterhin $x^{(0)} \in \mathbb{R}^n$ ein Vektor, für den $\langle x^{(0)}, v_1\rangle \neq 0$ für den zu $\lambda_1(A)$ gehörigen (normierten) Eigenvektor v_1 gilt. Dann konvergiert die Folge $y^{(i)} := x^{(i)}/\|x^{(i)}\|$ für $x^{(i)}$ aus (3.1) gegen $\pm v_1$, also gegen einen normierten Eigenvektor zum Eigenwert λ_1. Insbesondere konvergiert damit der *Rayleigh'sche Quotient*

$$\lambda^{(i)} := \frac{\langle Ax^{(i)}, x^{(i)}\rangle}{\langle x^{(i)}, x^{(i)}\rangle} = \langle Ay^{(i)}, y^{(i)}\rangle$$

gegen den Eigenwert λ_1.

Beweis: Wegen der Symmetrie von A existiert eine Orthonormalbasis von Eigenvektoren $v_1, \ldots, v_n$ von A. Damit gilt

$$x^{(0)} = \sum_{j=1}^{n} \alpha_j v_j \quad \text{mit } \alpha_j = \langle x^{(0)}, v_j \rangle,$$

insbesondere gilt also $\alpha_1 \neq 0$. Daraus folgt

$$x^{(i)} = A^i x^{(0)} = \sum_{j=1}^{n} \alpha_j \lambda_j^i v_j = \alpha_1 \lambda_1^i \underbrace{\left(v_1 + \sum_{j=2}^{n} \frac{\alpha_j}{\alpha_1} \left(\frac{\lambda_j}{\lambda_1} \right)^i v_j \right)}_{=:z^{(i)}}.$$

Da $|\lambda_j| < |\lambda_1|$ ist für $j = 2, \ldots, n$, gilt $\lim_{i\to\infty} z^{(i)} = v_1$ und damit

$$y^{(i)} = \frac{x^{(i)}}{\|x^{(i)}\|} = \pm \frac{z^{(i)}}{\|z^{(i)}\|} \to \pm v_1.$$

Die Konvergenz $\lambda^{(i)} \to \lambda_1$ folgt, indem wir $y^{(i)} = \pm v_1 + r^{(i)}$ mit $r^{(i)} \to 0$ schreiben. Dann gilt

$$\begin{aligned} \langle Ay^{(i)}, y^{(i)} \rangle &= \langle A(\pm v_1 + r^{(i)}), \pm v_1 + r^{(i)} \rangle \\ &= \langle Av_1, v_1 \rangle + \underbrace{\langle Ar^{(i)}, \pm v_1 \rangle + \langle \pm Av_1, r^{(i)} \rangle + \langle Ar^{(i)}, r^{(i)} \rangle}_{\to 0} \\ &\to \langle Av_1, v_1 \rangle = \langle \lambda_1 v_1, v_1 \rangle = \lambda_1 \|v_1\| = \lambda_1. \end{aligned}$$

□

Beachte, dass die Symmetrie der Matrix A hier nur eine hinreichende aber keine notwendige Bedingung für die Konvergenz des Verfahrens ist. So ist z.B. in [2] bewiesen, dass das Verfahren auch für die (nicht symmetrische) Matrix aus dem Seitenranking funktioniert. Hier kann überdies gezeigt werden, dass $\lambda_2 = 1 - m$ gilt; dies führt auf $\lambda_i/\lambda_1 \leq 0.85$ für $i = 2, 3, \ldots, n$. Rechnentechnisch ergibt sich aber scheinbar das Problem, dass die Matrix M dicht besetzt (keine Nulleinträge; *dense*) ist. Dies kann basierend auf der Beobachtung, dass sich $M = (1 - m)A + mS$ aus der dünn besetzten (*sparse*) Matrix A (Links auf andere Webseiten im Verhältnis zur Gesamtzahl aller Webseiten) und der Matrix S, deren Einträge alle gleich $1/n$ sind, zusammensetzt wie folgt gelöst werden:

- Für einen Vektor v mit $\sum_{i=1}^{n} v_i = 1$ gilt $Sv = (1/n\ 1/n\ \ldots 1/n)^\top$.
- Av kann effizient berechnet werden (so sind zum Beispiel unter der Annahme, dass jede Seite im Durchschnitt 10 Links enthält nur $1.6 \cdot 10^{11}$ Operationen nötig. Für eine direkte Berechnung von Mv wären es ungefähr $1.3 \cdot 10^{20}$ gewesen; $n = 8 \cdot 10^9$).

Folglich kann mittels $(1 - m)Av + (m/n\ m/n\ \ldots m/n)^\top$ die Matrix-Vektor-Multiplikation Mv mit vertretbarer Rechenzeit ausgeführt werden.

Das Vorgehen von *Google* hat weitere Anwendungen gefunden, zum Beispiel in der Spieltheorie oder der Medizin. Zudem zog K. Bryan zwei weitere Schlußfolgerungen:

- "Sophisticated linear algebra is at the core of much scientific computation, and pops up when you least expect it."
- "If you pay attention to your math professors, you might become a billionaire."

Die direkte Vektoriteration ist zwar aufgrund ihrer Einfachheit attraktiv, birgt aber mehrere Nachteile: Erstens erhalten wir nur den (betragsmäßig) größten Eigenwert $|\lambda_1|$ und den zugehörigen Eigenvektor, zweitens hängt die Konvergenzgeschwindigkeit davon ab, wie schnell die Terme $|\lambda_j/\lambda_1|^i$, also insbesondere $|\lambda_2/\lambda_1|^i$ gegen Null konvergieren.[4] Falls also $|\lambda_1| \approx |\lambda_2|$ und damit $|\lambda_2/\lambda_1| \approx 1$ gilt, ist nur sehr langsame Konvergenz zu erwarten.

Die *inverse Vektoriteration* vermeidet diese Nachteile. Sei A wiederum eine reelle symmetrische Matrix. Wir setzen voraus, dass wir einen Schätzwert $\tilde{\lambda} \in \mathbb{R}$ für einen Eigenwert $\lambda_j = \lambda_j(A)$ kennen, für den die Ungleichung

$$|\tilde{\lambda} - \lambda_j| < |\tilde{\lambda} - \lambda_k| \qquad \text{für alle } k \in \{1, 2, \ldots, n\} \backslash \{j\}$$

mit $\lambda_k = \lambda_k(A)$ gilt. Dann betrachten wir die Matrix $\widetilde{A} = (A - \tilde{\lambda}\mathrm{Id})^{-1}$. Diese besitzt die Eigenwerte $1/(\lambda_k - \tilde{\lambda})$ für $k = 1, \ldots, n$, also ist $1/(\lambda_j - \tilde{\lambda})$ der betragsmäßig größte Eigenwert.

Die inverse Vektoriteration ist nun gegeben durch

$$x^{(i+1)} = (A - \tilde{\lambda}\mathrm{Id})^{-1} x^{(i)}. \tag{3.2}$$

Aus Satz 3.1 (angewendet auf $(A - \tilde{\lambda}\mathrm{Id})^{-1}$ an Stelle von A) folgt, dass diese Iteration gegen einen normierten Eigenvektor v_j von $(A - \tilde{\lambda}\mathrm{Id})^{-1}$ zum Eigenwert $1/(\lambda_j - \tilde{\lambda})$ konvergiert. Wegen

$$\begin{aligned} & (A - \tilde{\lambda}\mathrm{Id})^{-1} v_j &&= 1/(\lambda_j - \tilde{\lambda}) v_j \\ \Leftrightarrow \quad & (\lambda_j - \tilde{\lambda}) v_j &&= (A - \tilde{\lambda}\mathrm{Id}) v_j \\ \Leftrightarrow \quad & \lambda_j v_j &&= A v_j \end{aligned}$$

ist dies gerade ein Eigenvektor von A zum Eigenwert λ_j. Die Konvergenzgeschwindigkeit ist bestimmt durch den Term

$$\max_{\substack{k=1,\ldots,n \\ k \neq j}} \frac{|\lambda_j - \tilde{\lambda}|}{|\lambda_k - \tilde{\lambda}|}.$$

Je kleiner dieser Term ist, das heißt je besser der Schätzwert ist, desto schneller wird die Konvergenz.

Die tatsächliche Implementierung der Iteration (3.2) ist hier etwas komplizierter als bei der direkten Iteration (3.1). Während dort in jedem Schritt eine Matrix-Vektor-Multiplikation mit Aufwand $\mathcal{O}(n^2)$ durchgeführt werden muss, geht hier die Inverse $(A - \tilde{\lambda}\mathrm{Id})^{-1}$ ein.[5]

[4] Falls $\lambda_1, \lambda_2, \ldots, \lambda_{n-1}, \lambda_n$ die Eigenwerte von A sind mit $\lambda_1 > \lambda_2 \geq \ldots \geq \lambda_{n-1} > \lambda_n$ und $|\lambda_1| \neq |\lambda_n|$ gilt, können sowohl λ_1 (der größte Eigenwert) als auch λ_n (der kleinste Eigenwert) mittels der Vektoriteration und einer Polverschiebung bestimmt werden, siehe Übung.

[5] Zudem ist die Matrix-Vektor-Multiplikation besonderes einfach zu parallelisieren; Hier sei auf Abschnitt 2.3 der Dissertation von Thomas Jahn zum SIMD-Prinzip (*Single Instruction Multiple Data*) verwiesen, siehe `http://num.math.uni-bayreuth.de/en/thesis/2010/Jahn_Thomas/Jahn_Thomas.pdf`.

In der Praxis berechnet man nicht die Inverse (weil dies numerisch sehr aufwändig ist), sondern löst das lineare Gleichungssystem

$$(A - \tilde{\lambda}\mathrm{Id})x^{(i+1)} = x^{(i)}, \tag{3.3}$$

wobei bei der Verwendung eines direkten Verfahrens die Matrix $(A - \tilde{\lambda}\mathrm{Id})$ nur einmal am Anfang der Iteration faktorisiert werden muss und dann in jedem Iterationsschritt einmal Vorwärts- bzw. Rückwärtseinsetzen durchzuführen ist. Der Aufwand $\mathcal{O}(n^3)$ der Zerlegung kommt also hier zum Aufwand des Verfahrens dazu, die einzelnen Iterationsschritte haben bei diesem Vorgehen allerdings keinen höheren Rechenaufwand als bei der direkten Iteration, da das Vorwärts- bzw. Rückwärtseinsetzen wie die Matrix-Vektor Multiplikation den Aufwand $\mathcal{O}(n^2)$ besitzen.

Für sehr gute Schätzwerte $\tilde{\lambda} \approx \lambda_j$ wird die Matrix $(A - \tilde{\lambda}\mathrm{Id})$ *fast* singulär (für $\tilde{\lambda} = \lambda_j$ wäre sie singulär), weswegen die Kondition von $(A - \tilde{\lambda}\mathrm{Id})$ sehr groß wird. Wegen der besonderen Struktur des Algorithmus führt dies hier aber nicht auf numerische Probleme, da zwar die Lösung $x^{(i+1)}$ des Gleichungssystems (3.3) mit großen Fehlern behaftet sein kann, sich diese Fehler aber in der hier eigentlich wichtigen *normierten Lösung* $x^{(i+1)}/\|x^{(i+1)}\|$ nicht auswirken. Wir wollen dies an einem Beispiel illustrieren.

Beispiel 3.2 Betrachte

$$A = \begin{pmatrix} -1 & 3 \\ -2 & 4 \end{pmatrix}$$

mit den Eigenwerten $\lambda_1(A) = 2$ und $\lambda_2(A) = 1$. Wir wählen $\tilde{\lambda} = 1 - \varepsilon$ für ein sehr kleines $\varepsilon > 0$. Dann ist die Matrix

$$(A - \tilde{\lambda}\mathrm{Id}) = \begin{pmatrix} -2+\varepsilon & 3 \\ -2 & 3+\varepsilon \end{pmatrix} \quad \text{mit} \quad (A - \tilde{\lambda}\mathrm{Id})^{-1} = \frac{1}{\varepsilon(\varepsilon+1)} \underbrace{\begin{pmatrix} 3+\varepsilon & -3 \\ 2 & -2+\varepsilon \end{pmatrix}}_{=:B}$$

fast singulär und man sieht leicht, dass für die Kondition z.B. in der Zeilensummennorm die Abschätzung $\mathrm{cond}_\infty(A - \tilde{\lambda}\mathrm{Id}) > 1/\varepsilon$ gilt. Die Inverse besitzt aber eine typische spezielle Struktur: die großen Einträge, die der Grund für die große Kondition sind, entstehen lediglich durch einen skalaren Vorfaktor. Daher ist die Berechnung von $y^{(i+1)} = x^{(i+1)}/\|x^{(i+1)}\|$ mittels (3.3) nicht sehr anfällig für Rundungsfehler, denn es gilt

$$y^{(i+1)} = \frac{(A - \tilde{\lambda}\mathrm{Id})^{-1}x^{(i)}}{\|(A - \tilde{\lambda}\mathrm{Id})^{-1}x^{(i)}\|} = \frac{Bx^{(i)}}{\|Bx^{(i)}\|_2},$$

das heißt der ungünstige große Faktor $1/(\varepsilon(\varepsilon+1))$ kürzt sich heraus. Ein Zahlenbeispiel illustriert dies noch einmal: Betrachten wir beispielsweise $x^{(i)} = (1, 0)^\top$ und $\varepsilon = 1/1000$, so erhält man

$$x^{(i+1)} = (A - \tilde{\lambda}\mathrm{Id})^{-1}x^{(i)} \approx \begin{pmatrix} 2998.001998 \\ 1998.001998 \end{pmatrix} \text{ und } y^{(i+1)} = \frac{x^{(i+1)}}{\|x^{(i+1)}\|_2} \approx \begin{pmatrix} 0.832135603 \\ 0.554572211 \end{pmatrix}$$

während man für die gestörte Lösung mit $\tilde{x}^{(i)} = (1.1, -0.1)^\top$

$$\tilde{x}^{(i+1)} = (A - \tilde{\lambda}\mathrm{Id})^{-1}\tilde{x}^{(i)} \approx \begin{pmatrix} 3597.502498 \\ 2397.502498 \end{pmatrix} \text{ und } \tilde{y}^{(i+1)} = \frac{\tilde{x}^{(i+1)}}{\|\tilde{x}^{(i+1)}\|_2} \approx \begin{pmatrix} 0.832117832 \\ 0.554598875 \end{pmatrix}$$

erhält. Die Störung in der ersten Nachkommastelle der rechten Seite bewirkt in $\tilde{x}^{(i+1)}$ zwar einen Fehler von etwa 600 und verstärkt sich daher sichtlich. In dem für den Algorithmus eigentlich wichtigen Vektor $\tilde{y}^{(i+1)}$ ist der Effekt der Störung hingegen erst in der fünften Nachkommastelle der Lösung ersichtlich. □

Bemerkung 3.3 [QR-Algorithmus] Zwar kann man mit der inversen Vektoriteration im Prinzip alle Eigenwerte berechnen, benötigt dafür aber geeignete Schätzwerte. Eine Alternative ist der QR-Algorithmus [4, Algorithmus 3.5], der in einer einzigen Rechnung alle Eigenwerte einer Matrix approximiert. Wie bereits bei linearen Gleichungssystemen spielt auch hier eine Faktorisierung mittels orthogonaler Matrizen eine wichtige Rolle, was auch die Nomenklatur erklärt.

Eine Herleitung für reelle symmetrische Matrizen findet sich in [4]; dort wird ebenfalls kurz auf eine Verallgemeinerung auf allgemeine Matrizen eingegangen. Die Grundidee für reelle symmetrische Matrizen besteht darin, dass für solche Matrizen eine Orthonormalbasis aus Eigenvektoren v_1, $v_2, \ldots,$ v_n besteht, so dass für die orthogonale Matrix $Q = (v_1, \ldots, v_n) \in \mathbb{R}^{n\times n}$ die Gleichung

$$Q^\top AQ = \Lambda = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$$

gilt, wobei die λ_i gerade die Eigenwerte von A sind. Dann können die Eigenwerte direkt aus der resultierenden Diagonalmatrix Λ und die zugehörigen Eigenvektoren aus Q abgelesen werden.

Um den Rechenaufwand zu reduzieren wird die Matrix A in einem vorbereitenden Schritt zunächst auf eine möglichst einfache Form gebracht. Dies sorgt dafür, dass die Anzahl der Rechenoperationen je Iteration in der Größenordnung $\mathcal{O}(n^2)$ bewegt, siehe [4, Bemerkung 3.9 (i)]. Hierbei wählt man die *Tridiagonalgestalt* und nutzt aus, dass man Householder-Matrizen dazu verwenden kann, Matrizen auf Tridiagonalgestalt zu konjugieren. Grundlage dafür ist [4, Lemma 3.3], das – für eine reelle symmetrische Matrix A – (konstruktiv) die Existenz einer orthogonalen Matrix P zeigt, so dass $P^\top AP$ symmetrisch und in Tridiagonalgestalt ist.[6] □

[6] Für nicht symmetrische Matrizen wählt man die sogenannte obere Hessenberg-Gestalt, siehe [1] für detaillierte Ausführungen.

Kapitel 4

Interpolation

Die Interpolation von Funktionen oder Daten ist ein häufig auftretendes Problem sowohl in der Mathematik als auch in vielen Anwendungen.

Das allgemeine Problem, die sogenannte *Dateninterpolation*, entsteht, wenn wir eine Menge von Daten (x_i, f_i) für $i = 0, \ldots, n$ gegeben haben (z.B. Messwerte eines Experiments). Die Problemstellung ist nun wie folgt: Gesucht ist eine Funktion F, für welche die Gleichung

$$F(x_i) = f_i \quad \text{für } i = 0, 1, \ldots, n \tag{4.1}$$

gilt.

Ein wichtiger Spezialfall dieses Problems ist die *Funktionsinterpolation*: Nehmen wir an, dass wir eine reellwertige Funktion $f : \mathbb{R} \to \mathbb{R}$ gegeben haben, die aber (z.B. weil keine explizite Formel bekannt ist) sehr kompliziert auszuwerten ist. Ein Beispiel einer solchen Funktion ist die in der Stochastik oft benötigte Gauß-Verteilungsfunktion

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{x} e^{-y^2/2} dy,$$

für die keine geschlossene Formel existiert.

Das Ziel der Interpolation liegt nun darin, eine Funktion $F(x)$ zu bestimmen, die leicht auszuwerten ist, und für die für vorgegebene *Stützstellen* x_0, $x_1, \ldots, x_n$ die Gleichung

$$F(x_i) = f(x_i) \quad \text{für } i = 0, 1, \ldots, n \tag{4.2}$$

gilt. Mit der Schreibweise

$$f_i = f(x_i),$$

erhalten wir hier wieder die Bedingung (4.1), weswegen (4.2) tatsächlich ein Spezialfall von (4.1) ist.

Wir werden in diesem Kapitel zum einen Verfahren zur Lösung des Interpolationsproblems (4.1) entwickeln, die dann selbstverständlich auch auf den Spezialfall (4.2) anwendbar sind. Die Wichtigkeit dieses Spezialfalls liegt in diesem Zusammenhang darin, dass man bei der Interpolation einer Funktion f in natürlicher Weise einen *Interpolationsfehler* über den Abstand zwischen f und F definieren kann, und daher ein Maß für die Güte

des Verfahrens erhält. Bei der Dateninterpolation ergibt dies keinen rechten Sinn, da es ja keine Funktion f gibt, bezüglich der man einen Fehler messen könnte.

Zum anderen werden wir Verfahren betrachten, die speziell auf die Funktionsapproximation (4.2) zugeschnitten sind, da sich bei diesen die Wahl der Stützstellen x_i aus dem Verfahren ergibt, also nicht beliebig vorgegeben werden kann.

4.1 Polynominterpolation

Eine einfache aber oft sehr effektive Methode zur Interpolation ist die Wahl von F als Polynom, also als Funktion der Form

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m. \tag{4.3}$$

Hierbei werden die Werte a_i, $i = 0, \ldots, m$, die *Koeffizienten* des Polynoms genannt. Die höchste auftretende Potenz (hier also m, falls $a_m \neq 0$) heißt der *Grad* des Polynoms. Um zu betonen, dass wir hier Polynome verwenden, schreiben wir in diesem Abschnitt „P“ statt „F“ für die Interpolationsfunktion. Den Raum der Polynome vom Grad $\leq m$ bezeichnen wir mit $\mathcal{P}_m$. Dieser Funktionenraum ist ein $(m+1)$-dimensionaler Vektorraum über $\mathbb{R}$ bzw. $\mathbb{C}$ mit Basis $\mathcal{B} = \{1, x, \ldots, x^m\}$, da Addition von Polynomen und Multiplikation mit Skalaren wieder ein Polynom des selben Grads ergeben. Andere Basen dieses Vektorraums werden in den Übungen behandelt.

Das Problem der Polynominterpolation liegt nun darin, ein Polynom P zu bestimmen, das die Interpolationsbedingung (4.1) erfüllt. Zunächst einmal müssen wir uns dazu überlegen, welchen Grad das gesuchte Polynom haben soll. Hier hilft uns der folgende Satz.

Satz 4.1 Sei $n \in \mathbb{N}$ und seien Daten (x_i, f_i) für $i = 0, \ldots, n$ gegeben, so dass die Stützstellen paarweise verschieden sind, das heißt $x_i \neq x_j$ für alle $i \neq j$. Dann gibt es genau ein Polynom $P \in \mathcal{P}_n$, also vom Grad $\leq n$, das die Bedingung

$$P(x_i) = f_i \qquad \text{für alle } i \in \{0, 1, \ldots, n\}$$

erfüllt.

Beweis: Die Koeffizienten a_i des interpolierenden Polynoms erfüllen das lineare Gleichungssystem

$$\begin{pmatrix} 1 & x_0 & \ldots & x_0^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix}.$$

Die Determinante dieser Matrix ist

$$\prod_{i=0}^{n} \left(\prod_{j=i+1}^{n} (x_j - x_i) \right)$$

und ist damit ungleich Null, falls die x_i paarweise verschieden sind. Also ist die Matrix invertierbar und das Gleichungssystem besitzt eine eindeutige Lösung. □

Für $n+1$ gegebene Datenpunkte (x_i, f_i) *passt* also gerade ein Polynom vom Grad n.

Nun ist es aus verschiedenen Gründen nicht besonders effizient, dieses lineare Gleichungssystem tatsächlich zu lösen, um die a_i zu bestimmen (wir erinnern daran, dass die direkte Lösung des linearen Gleichungssystems den Aufwand der Ordnung $\mathcal{O}(n^3)$ hat). Wir betrachten daher eine andere Technik zur Berechnung des Polynoms P. Beachte, dass diese das gleiche Polynom liefert, auch wenn es auf andere Art dargestellt wird.

4.1.1 Lagrange-Polynome und baryzentrische Koordinaten

Die Idee der Lagrange-Polynome beruht auf einer geschickten Darstellung für Polynome. Für die vorgegebenen Stützstellen x_0, $x_1 \ldots, x_n$ definieren wir für $i = 0, \ldots, n$ die *Lagrange-Polynome* L_i als

$$L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

Man rechnet leicht nach, dass diese Polynome alle vom Grad n sind, und darüberhinaus die Gleichung

$$L_i(x_k) = \begin{cases} 1 & \text{für } i = k \\ 0 & \text{für } i \neq k \end{cases}$$

erfüllen. Mit Hilfe der L_i kann man das Interpolationspolynom einfach explizit berechnen.

Satz 4.2 Seien Daten (x_i, f_i) für $i = 0, \ldots, n$ mit paarweise verschiedenen Stützstellen x_i gegeben. Dann ist das eindeutige Interpolationspolynom $P(x)$ mit $P(x_i) = f_i$ gegeben durch

$$P(x) = \sum_{i=0}^{n} f_i L_i(x).$$

Beweis: Offensichtlich ist die angegebene Funktion ein Polynom vom Grad $\leq n$. Darüberhinaus gilt

$$P(x_k) = \sum_{i=0}^{n} \underbrace{f_i L_i(x_k)}_{\substack{=0 \text{ falls } i \neq k \\ = f_k \text{ falls } i = k}} = f_k,$$

also gerade die gewünschte Bedingung (4.1). □

Beispiel 4.3 Betrachte die Daten (3, 68), (2, 16), (5, 352). Die zugehörigen Lagrange-Polynome sind gegeben durch

$$\begin{aligned} L_0(x) &= \frac{x-2}{3-2}\frac{x-5}{3-5} = -\frac{1}{2}(x-2)(x-5), \\ L_1(x) &= \frac{x-3}{2-3}\frac{x-5}{2-5} = \frac{1}{3}(x-3)(x-5), \\ L_2(x) &= \frac{x-2}{5-2}\frac{x-3}{5-3} = \frac{1}{6}(x-2)(x-3). \end{aligned}$$

Damit erhalten wir

$$P(x) = -68\,\frac{1}{2}(x-2)(x-5) + 16\,\frac{1}{3}(x-3)(x-5) + 352\frac{1}{6}(x-2)(x-3).$$

Für $x = 3$ ergibt sich $P(3) = -68\frac{1}{2}(3-2)(3-5) = 68$, für $x = 2$ berechnet man $P(2) = 16\frac{1}{3}(2-3)(2-5) = 16$ und für $x = 5$ erhalten wir $P(5) = 352\frac{1}{6}(5-2)(5-3) = 352$. □

Durch Abzählen der notwendigen Operationen sieht man, dass die direkte Auswertung des Polynoms P in dieser Form den Aufwand $\mathcal{O}(n^2)$ besitzt, also deutlich effizienter als die Lösung eines linearen Gleichungssystems ist. Für eine effiziente direkte Auswertung sollte man die Nenner der Lagrange-Polynome vorab berechnen und speichern, damit diese nicht bei jeder Auswertung von P erneut berechnet werden müssen.

Es geht aber noch effizienter, wenn wir die Auswertung der Lagrange-Polynome geschickt umformulieren. Dazu schreiben wir den Zähler von

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

als

$$\frac{\ell(x)}{x - x_i} \quad \text{mit} \quad \ell(x) := \prod_{j=0}^{n} x - x_j.$$

Den Nenner schreiben wir mittels der sogenannten *baryzentrischen Koordinaten*

$$w_i := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{1}{x_i - x_j}.$$

Dann gilt

$$L_i(x) = \ell(x)\frac{w_i}{x - x_i}$$

und damit

$$P(x) = \sum_{i=0}^{n} L_i(x) f_i = \sum_{i=0}^{n} \ell(x)\frac{w_i}{x - x_i} f_i = \ell(x) \sum_{i=0}^{n} \frac{w_i}{x - x_i} f_i.$$

Beispiel 4.4 Betrachte wiederum die Daten (3; 68), (2; 16), (5; 352). Das zugehörige ℓ ist gegeben durch

$$\ell(x) = (x-2)(x-3)(x-5)$$

und die w_i berechnen sich zu

$$\begin{aligned} w_0 &= \frac{1}{3-2}\,\frac{1}{3-5} = -\frac{1}{2}, \\ w_1 &= \frac{1}{2-3}\,\frac{1}{2-5} = \frac{1}{3}, \\ w_3 &= \frac{1}{5-2}\,\frac{1}{5-3} = \frac{1}{6}. \end{aligned}$$

Damit erhalten wir

$$\begin{aligned} P(x) &= \ell(x)\left(\frac{-\frac{1}{2}}{x-3}\,68 + \frac{\frac{1}{3}}{x-2}\,16 + \frac{\frac{1}{6}}{x-5}\,352\right) \\ &= -\frac{1}{2}(x-2)(x-5)\,68 + \frac{1}{3}(x-3)(x-5)\,16 + \frac{1}{6}(x-2)(x-3)\,352, \end{aligned}$$

also – wie zu erwarten – das gleiche Polynom wie oben. □

Um dieses Verfahren effizient zu implementieren, teilen wir die Berechnung in zwei Algorithmen auf. Des Weiteren wird der Aufwand um ein Viertel reduziert, indem die Differenz $(x_i - x_j)$ (und $x_j - x_i$; zusätzliches Vorzeichen beachten) nur einmal ausgewertet wird.

Algorithmus 4.5 (Berechnung der baryzentrischen Koordinaten)
Eingabe: Stützstellen $x_0,\ldots,x_n$.

Setze $w_i := 1$ für $i = 0, 1, \ldots, n$
Für i von 0 bis $n-1$:
  Für j von $i+1$ bis n:
    Setze $y := x_i - x_j$
    Setze $w_i := w_i / y$
    Setze $w_j := -w_j / y$
  Ende der j-Schleife
Ende der i-Schleife

Ausgabe: baryzentrische Koordinaten $w_0,\ldots,w_n$ □

Durch Abzählen der Operationen sieht man leicht, dass die Berechnung der w_i gerade $3\sum_{i=1}^{n} i = 3n(n+1)/2 = \mathcal{O}(n^2)$ Operationen benötigt. Dies entspricht der Ordnung des Aufwandes der direkten Auswertung von P. Der Trick liegt nun aber darin, die w_i einmal vorab zu berechnen und die gespeicherten Werte in der Auswertung von P zu verwenden.

Algorithmus 4.6 (Auswertung des Interpolationspolynoms)
Eingabe: Stützstellen $x_0,\ldots,x_n$, Werte $f_0,\ldots,f_n$, baryzentrische Koordinaten $w_0,\ldots,w_n$, Auswertungsstelle x

Setze $l := 1$ und $s := 0$ (Variablen für ℓ und $\sum_{i=0}^{n} \frac{w_i}{x-x_i} f_i$)
Für i von 0 bis n
  Setze $y := x - x_i$
  Falls $y = 0$ ist, setze $P := f_i$ und beende den Algorithmus
  Setze $l := l * y$
  Setze $s := s + w_i * f_i / y$
Ende der i-Schleife
Setze $P := l * s$

Ausgabe: Polynomwert $P = P(x)$ □

Durch Abzählen der Operationen sieht man: die Auswertung benötigt gerade $5(n+1)+1 = 5n+6 = \mathcal{O}(n)$ Operationen. Sind also die w_i einmal berechnet, so ist die Auswertung für ein gegebenes x deutlich weniger aufwändig als die direkte Auswertung von P. Dies ist z.B. bei der grafischen Darstellung des Polynoms ein wichtiger Vorteil, da das Polynom dabei für viele verschiedene x ausgewertet werden muss.

4.1.2 Fehlerabschätzungen

Wir betrachten in diesem Abschnitt das Problem der Funktionsinterpolation (4.2) für eine gegebene Funktion $f : \mathbb{R} \to \mathbb{R}$. Wir wollen abschätzen, wie groß der Abstand des interpolierenden Polynoms P von der Funktion f ist. Hierbei bezeichnen wir mit $[a, b]$ ein Interpolationsintervall mit der Eigenschaft, dass alle Stützstellen $x_i \in [a, b]$ erfüllen und wir verwenden wieder die Maximumsnorm

$$\|f\|_\infty := \sup_{x\in[a,b]} |f(x)|.$$

Der folgende Satz gibt eine Abschätzung für den Abstand in Abhängigkeit von den Stützstellen an. Hierbei bezeichnet $f^{(k)}$ die k-te Ableitung der Funktion f.

Satz 4.7 Sei f $(n+1)$-mal stetig differenzierbar und sei P das Interpolationspolynom zu den paarweise verschiedenen Stützstellen $x_0, \ldots, x_n$. Dann gelten die folgenden Ausagen.

(i) Für alle $x \in [a, b]$ gibt es ein $\xi \in [a, b]$, so dass die Gleichung

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n)$$

gilt.

(ii) Für alle $x \in [a, b]$ gilt die Abschätzung

$$|f(x) - P(x)| \le \|f^{(n+1)}\|_\infty \frac{|(x - x_0)(x - x_1)\cdots(x - x_n)|}{(n+1)!}.$$

(iii) Es gilt die Abschätzung

$$\|f - P\|_\infty \le \|f^{(n+1)}\|_\infty \frac{(b-a)^{n+1}}{(n+1)!}.$$

Beweis: (i) Wähle ein $x \in [a, b]$ mit $x \neq x_i$ für ein $i \in \{0, 1, \ldots, n\}$ und setze

$$c = \frac{f(x) - P(x)}{(x - x_0)\cdots(x - x_n)}.$$

Mit diesem c definieren wir die Funktion

$$\Delta(y) = f(y) - P(y) - c(y - x_0)\cdots(y - x_n).$$

Dann ist $\Delta(y)$ wieder $(n+1)$-mal stetig differenzierbar und hat (mindestens) die $n+2$ Nullstellen $y = x_0, \ldots, x_n$ und $y = x$. Wir nummerieren diese Nullstellen aufsteigend mit der Bezeichnung $y_0^{(0)} < y_1^{(0)} < \ldots < y_{n+1}^{(0)}$. Nach dem Satz von Rolle gilt: Zwischen je zwei Nullstellen der Funktion $\Delta(y)$ liegt (mindestens) eine Nullstelle ihrer Ableitung $\Delta'(y)$. Also liegt für jedes $i = 0, \ldots, n$ zwischen den Werten $y_i^{(0)}$ und $y_{i+1}^{(0)}$ ein Wert $y_i^{(1)}$ mit $\Delta'(y_i^{(1)}) = 0$, und wir erhalten $n+1$ Nullstellen $y_0^{(1)} < y_1^{(1)} < \ldots < y_n^{(1)}$ für die Ableitung $\Delta'(y) = \Delta^{(1)}(y)$. Indem wir induktiv fortfahren, erhalten wir $n+2-k$ Nullstellen $y_0^{(k)} < y_1^{(k)} < \ldots < y_{n+1-k}^{(k)}$ der Ableitung $\Delta^{(k)}$ und damit für $k = n+1$ eine Nullstelle $\xi = y_0^{(n+1)}$ der Funktion $\Delta^{(n+1)}$. Da P ein Polynom vom Grad $\leq n$ ist, folgt $P^{(n+1)}(y) \equiv 0$. Außerdem gilt

$$[(y - x_0) \cdots (y - x_n)]^{(n+1)} = (n+1)! \qquad \text{für alle } y \in \mathbb{R}.$$

Damit erhalten wir

$$0 = \Delta^{(n+1)}(\xi) = f^{(n+1)}(\xi) - c(n+1)! = f^{(n+1)}(\xi) - \frac{f(x) - P(x)}{(x - x_0) \cdots (x - x_n)}(n+1)!,$$

also

$$f^{(n+1)}(\xi) = \frac{f(x) - P(x)}{(x - x_0) \cdots (x - x_n)}(n+1)!.$$

Auflösen nach $f(x) - P(x)$ liefert die Gleichung in (i).

(ii) Diese Abschätzung folgt aus (i) wegen

$$\begin{aligned} |f(x) - P(x)| &= \left| \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1) \cdots (x - x_n) \right| \\ &\leq \max_{y \in [a,b]} \left| \frac{f^{(n+1)}(y)}{(n+1)!}(x - x_0)(x - x_1) \cdots (x - x_n) \right|, \end{aligned}$$

da $\xi \in [a, b]$ ist.

(iii) Für alle x und x_i aus $[a, b]$ gilt die Abschätzung

$$|x - x_i| \leq b - a.$$

Damit erhalten wir aus (ii)

$$\|f - P\|_\infty \leq \|f^{(n+1)}\|_\infty \frac{(b-a)^{n+1}}{(n+1)!},$$

also gerade die Behauptung. □

Wir illustrieren diese Abschätzung an einem Beispiel.

Beispiel 4.8 Betrachte die Funktion $f(x) = \sin(x)$ auf dem Intervall $[0, 2\pi]$. Die Ableitungen von f sind

$$f^{(1)}(x) = \cos(x), \;\; f^{(2)}(x) = -\sin(x), \;\; f^{(3)}(x) = -\cos(x), \;\; f^{(4)}(x) = \sin(x), \;\; \ldots$$

Für alle diese Funktionen gilt $|f^{(k)}(x)| \leq 1$ für alle $x \in \mathbb{R}$. Mit äquidistanten Stützstellen $x_i = 2\pi i/n$ ergibt sich damit die Abschätzung

$$|f(x) - P(x)| \leq \max_{y \in [a,b]} \left| f^{(n+1)}(y) \right| \frac{(b-a)^{n+1}}{(n+1)!} \leq \frac{(2\pi)^{n+1}}{(n+1)!}.$$

Dieser Term konvergiert für wachsende n sehr schnell gegen Null, weswegen man schon für kleine n eine sehr gute Übereinstimmung der Funktionen erwarten kann, siehe Abbildung 4.1. □

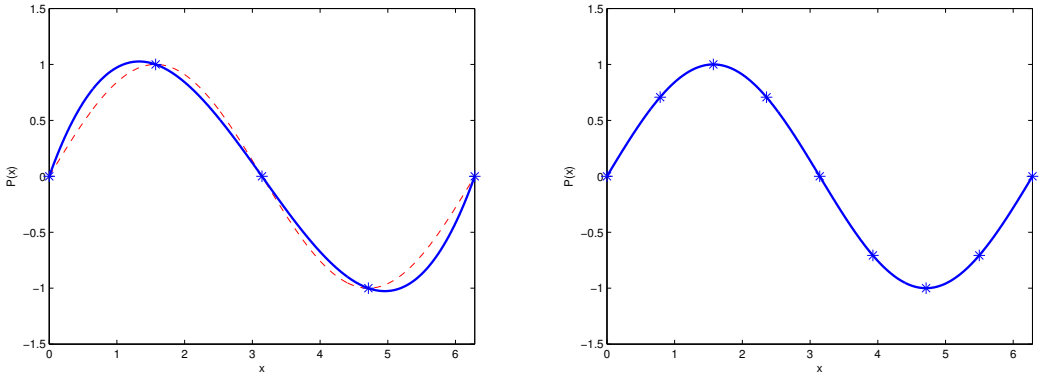


Abbildung 4.1: Links: Interpolationspolynom für $n = 4$ (links) und $n = 8$ (rechts) äquidistante Stützstellen für die Funktion $f(x) = \sin(x)$ auf $[0, 2\pi]$.

Fügt man jedoch in Beispiel 4.8 weitere Stützstellen ein, beobachtet man Oszillationen am Rand des Intervalls $[0, 2\pi]$ (siehe Abbildung 4.2) — trotz unser Fehlerabschätzung aus Satz 4.7. Dies muss ein numerischer Effekt aus, was eine Untersuchung der Kondition des Interpolationsproblems motiviert.

4.1.3 Kondition

In diesem Abschnitt wollen wir die Kondition der Polynominterpolation betrachten, wobei wir das Polynominterpolationsproblem für fest vorgegebene Stützstellen betrachten. In diesem Fall ist die Abbildung

$$\phi : (f_0, \ldots, f_n) \mapsto \sum_{i=0}^{n} f_i L_i$$

des Datenvektors $(f_0, \ldots, f_n)$ auf das interpolierende Polynom $P \in \mathcal{P}_n$ eine lineare Abbildung $\phi : \mathbb{R}^{n+1} \to \mathcal{P}_n$, weshalb wir die (absolute) Kondition κ_{abs} als induzierte Operatornorm

$$\kappa_{abs} := \|\phi\|_\infty = \sup_{\substack{f \in \mathbb{R}^{n+1} \\ f \neq 0}} \frac{\|\phi(f)\|_\infty}{\|f\|_\infty}$$

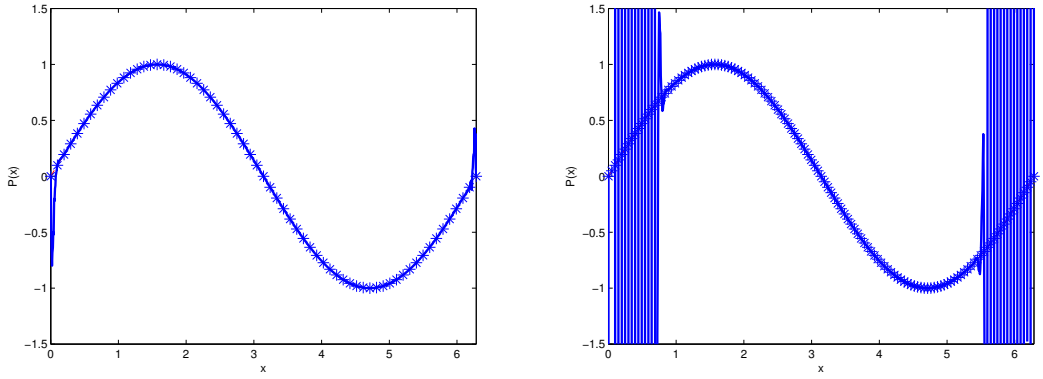


Abbildung 4.2: Links: Interpolationspolynom für $n = 64$ (links) und $n = 128$ (rechts) äquidistante Stützstellen für die Funktion $f(x) = \sin(x)$ auf $[0, 2\pi]$.

dieser linearen Abbildung berechnen können. Diese induzierte Operatornorm ist die Erweiterung der induzierten Matrixnorm auf lineare Abbildungen, die nicht notwendigerweise durch eine Matrix definiert sind. Im Gegensatz zu den linearen Gleichungssystemen verwenden wir hier die absolute Kondition, weil eine relative Definition hier keine anschauliche Interpretation besitzt. Auf dem Polynomraum $\mathcal{P}_n$ verwenden wir dabei die Maximumsnorm

$$\|P\|_\infty := \max_{x\in[a,b]} |P(x)|,$$

des Raums der stetigen reellwertigen Funktionen $\mathcal{C}([a, b], \mathbb{R})$, wobei wir $a = \min_{i=0,\dots,n}\{x_i\}$ und $b = \max_{i=0,\dots,n}\{x_i\}$ wählen (beachte, dass wir keine Ordnung der Stützstellen x_i vorausgesetzt haben).

Satz 4.9 Seien x_0, $x_1, \dots, x_n$ paarweise verschiedene Stützstellen und L_i die zugehörigen Lagrange-Polynome. Dann ist die absolute Kondition des Interpolationsproblems mit diesen Stützstellen gegeben durch

$$\kappa_{\text{abs}} = \Lambda_n := \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty ,$$

wobei Λ_n als *Lebesgue-Konstante* bezeichnet wird.

Beweis: Es gilt

$$\begin{aligned} |\phi(f)(x)| &= \left| \sum_{i=0}^{n} f_i L_i(x) \right| \le \sum_{i=0}^{n} |f_i|\,|L_i(x)| \\ &\le \|f\|_\infty \max_{x\in[a,b]} \sum_{i=0}^{n} |L_i(x)| = \|f\|_\infty \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty = \|f\|_\infty \Lambda_n, \end{aligned}$$

für alle $x \in [a, b]$, woraus $\|\phi(f)\|_\infty \le \|f\|_\infty \Lambda_n$ für alle $f \in \mathbb{R}^{n+1}$ und damit $\|\phi\| \le \Lambda_n$ folgt. Für die umgekehrte Richtung konstruieren wir ein $g \in \mathbb{R}^{n+1}$ so, dass

$$|\phi(g)(x^*)| = \|g\|_\infty \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty$$

für ein $x^* \in [a,b]$ gilt. Sei dazu $x^* \in [a,b]$ die Stelle, an der die Funktion $x \mapsto \sum_{i=0}^n |L_i(x)|$ ihr Maximum annimmt, also

$$\sum_{i=0}^n |L_i(x^*)| = \left\| \sum_{i=0}^n |L_i| \right\|_\infty = \Lambda_n.$$

Wir wählen $g \in \mathbb{R}^{n+1}$ als $g_i = \operatorname{sgn}(L_i(x^*))$. Dann gilt $\|g\|_\infty = 1$ und $g_i L_i(x^*) = |L_i(x^*)|$, also

$$\|\phi(g)\|_\infty \geq |\phi(g)(x^*)| = \left| \sum_{i=0}^n g_i L_i(x^*) \right| = \left| \sum_{i=0}^n |L_i(x^*)| \right| = \|g\|_\infty \left| \sum_{i=0}^n |L_i(x^*)| \right| = \|g\|_\infty \Lambda_n,$$

weswegen $\|\phi\| \geq \Lambda_n$ ist. Zusammen erhalten wir die Behauptung $\kappa_{\text{abs}} = \|\phi\|_\infty = \Lambda_n$. □

Die Zahl Λ_n hängt natürlich von der Anzahl und Lage der Stützstellen ab. In der folgenden Tabelle 4.1 sind die Konditionen für das Intervall $[-1,1]$ und verschiedene Anzahlen äquidistanter Stützstellen $x_i = -1 + 2i/n$ sowie für die sogenannten Tschebyscheff-Stützstellen $x_i = \cos[(2i+1)\pi/(2n+2)]$, die wir in Abschnitt 4.2 näher kennen lernen werden, dargestellt.

n	κ_{abs} für äquidistante Stützstellen	κ_{abs} für Tschebyscheff-Stützstellen
5	3.11	2.10
10	29.89	2.49
15	512.05	2.73
20	10986.53	2.90
60	$2.97 \cdot 10^{15}$	3.58
100	$1.76 \cdot 10^{27}$	3.90

Tabelle 4.1: Kondition κ_{abs} für verschiedene Stützstellen

Man sieht, dass das Problem für äquidistante Stützstellen und große n sehr schlecht konditioniert ist. Dies erklärt die starken Oszillationen bei numerisch erzeugten Interpolationspolynomen bei großer Stützstellenanzahl, selbst wenn die zu interpolierende Funktion gutartig ist (vergleiche Abbildung 4.2). Wir werden daher in den nächsten Abschnitten weitere Möglichkeiten zur Interpolation betrachten, die diese Probleme umgehen, indem sie entweder besser positionierte Stützstellen verwenden oder ohne Polynome hohen Grades auskommen.

4.2 Funktionsinterpolation und Stützstellenwahl

In diesem Abschnitt beschäftigen wir uns speziell mit der Frage der Funktionsinterpolation (4.2) durch Polynome. Wie bereits erwähnt, unterscheidet sich diese aus algorithmischer Sicht von der Dateninterpolation (4.1) dadurch, dass man die Stützstellen x_i frei wählen kann. Dies führt auf die Frage, wie man diese Stützstellen für ein gegebenenes Interpolationsintervall $[a,b]$ optimal wählen kann. Wir wollen dieses Problem lösen *ohne* die Kenntnis der zu interpolierenden Funktion f vorauszusetzen.

Dazu betrachten wir die Fehlerabschätzung aus Satz 4.7(ii) für die Funktionsinterpolation. Dort haben wir die Ungleichung

$$|f(x) - P(x)| \leq \|f^{(n+1)}\|_\infty \frac{|(x-x_0)(x-x_1)\cdots(x-x_n)|}{(n+1)!}. \tag{4.4}$$

für $x \in [a,b]$ bewiesen. Wir wollen nun untersuchen, wie man die Stützstellen x_i wählen muss, so dass diese Fehlerschranke minimal wird. Da wir hierbei kein spezielles x vorgeben wollen (die Abschätzung soll für alle x optimal sein, also für $\|f-P\|_\infty$), besteht die Aufgabe also darin, Stützstellen $x_0, \ldots, x_n$ zu finden, so dass der Ausdruck

$$\max_{x\in[a,b]} |(x-x_0)(x-x_1)\cdots(x-x_n)| \tag{4.5}$$

minimal wird.

O.B.d.A. betrachten wir nun das Intervall $[a,b] = [-1,1]$, denn wenn wir auf $[-1,1]$ die optimalen Stützstellen x_i gefunden haben, so sind die mittels $\tilde{x}_i = a + (x_i+1)(b-a)/2$ definierten Stützstellen auf $[a,b]$ ebenfalls optimal und es gilt

$$\max_{x\in[a,b]} |(x-\tilde{x}_0)(x-\tilde{x}_1)\cdots(x-\tilde{x}_n)| = \left(\frac{b-a}{2}\right)^{n+1} \max_{x\in[-1,1]} |(x-x_0)(x-x_1)\cdots(x-x_n)|.$$

Definieren wir nun das Polynom (führender Koeffizient ist $a_{n+1} = 1$)

$$R_{n+1}(x) = (x-x_0)(x-x_1)\cdots(x-x_n)$$

für beliebige Stützstellen $x_0, \ldots, x_n$. Dann sind die Stützstellen x_i gerade die Nullstellen von R_{n+1} und der Ausdruck (4.5) ist gerade die Maximumsnorm $\|R_{n+1}\|_\infty$. Die Minimierung von (4.5) ist also äquivalent zur folgenden Problemstellung: Unter allen Polynomen R_{n+1} vom Grad $n+1$ mit führendem Koeffizienten $a_{n+1} = 1$ finde dasjenige mit kleinster Maximumsnorm auf $[-1,1]$.

Der folgende Satz zeigt, dass die bereits in Tabelle 4.1 aufgeführten Tschebyscheff-Knoten (Nullstellen des zugehörigen normierten Tschebyscheff-Polynoms) gerade das Minimum dieses Optimierungsproblems liefern.[1]

Satz 4.10 Die Tschebyscheff-Knoten

$$x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right), \quad i = 0, \ldots, n,$$

minimieren den Ausdruck (4.5) und damit die Fehlerabschätzung (4.4).

Für die Tschebyscheff-Stützstellen x_i erhält man damit in (4.5)

$$\max_{x\in[-1,1]} |(x-x_0)(x-x_1)\cdots(x-x_n)| = \frac{1}{2^n}.$$

[1] Ein Beweis findet sich im bereits erwähnten Skript von Prof. Grüne "Einführung in die numerische Mathematik".

Für allgemeine Intervalle $[a, b]$ ergibt sich daraus für die durch

$$\tilde{x}_i = a + (x_i + 1)\,(b - a)/2$$

gegebenen transformierten Stützstellen

$$\max_{x\in[a,b]} |(x - \tilde{x}_0)(x - \tilde{x}_1)\cdots(x - \tilde{x}_n)| = \frac{(b-a)^{n+1}}{2^{2n+1}} = 2\left(\frac{b-a}{4}\right)^{n+1}.$$

Beachte, dass die Randpunkte -1 und 1 des Interpolationsintervalls keine Tschebyscheff-Knoten und damit keine Stützstellen sind. Wir interpolieren mit dieser Methode also auch außerhalb des durch die Stützstellen definierten Intervalls.

Zusammenfassend kann man feststellen, dass die Tschebyscheff-Stützstellen sowohl dafür sorgen, dass das Interpolationsproblem gut konditioniert ist als auch unsere Abschätzung an den Interpolationsfehler minimieren.

4.3 Splineinterpolation

Wir haben gesehen, dass die Polynominterpolation aus Konditionsgründen problematisch ist, wenn wir viele Stützstellen gegeben haben und diese nicht — wie die Tschebyscheff–Stützstellen — optimal gewählt sind. Dies kann insbesondere bei der Dateninterpolation (4.1) auftreten, wenn die Stützstellen fest vorgegeben sind und nicht frei gewählt werden können. Wir behandeln daher in diesem Abschnitt eine alternative Interpolationstechnik, die auch bei einer großen Anzahl von Stützstellen problemlos funktioniert. Wir betrachten dazu paarweise verschiedene Stützstellen und nehmen an, dass diese aufsteigend angeordnet sind, also $x_0 < x_1 < \ldots < x_n$ gilt.

Die Grundidee der *Splineinterpolation* liegt darin, die interpolierende Funktion (die wir hier mit "S" für "Spline" bezeichnen) nicht global, sondern nur auf jedem Teilintervall $[x_i, x_{i+1}]$ als Polynom zu wählen. Diese Teilpolynome sollten dabei an den Intervallgrenzen nicht beliebig sondern möglichst glatt zusammenlaufen. Eine solche Funktion, die aus glatt zusammengefügten stückweisen Polynomen besteht, nennt man *Spline*.

Formal wird dies durch die folgende Definition präzisiert.

Definition 4.11 Seien $x_0 < x_1 < \ldots < x_n$ Stützstellen und $k \in \mathbb{N}$. Eine stetige und $(k-1)$-mal stetig differenzierbare Funktion $S : [x_0, x_n] \to \mathbb{R}$ heißt *Spline vom Grad k*, falls S auf jedem Intervall $I_i = [x_{i-1}, x_i]$ mit $i = 1, \ldots, n$ durch ein Polynom P_i vom Grad $\leq k$ gegeben ist, d.h. für $x \in I_i$ gilt

$$S(x) = P_i(x) = a_{i\,0} + a_{i\,1}(x - x_{i-1}) + \ldots + a_{i\,k}(x - x_{i-1})^k = \sum_{j=0}^{k} a_{i\,j}(x - x_{i-1})^j. \quad (4.6)$$

Den Raum der Splines vom Grad k zur Stützstellenmenge $\Delta = \{x_0, x_1, \ldots, x_n\}$ bezeichnen wir mit $S_{\Delta,k}$. □

Ein solcher Spline aus Definition 4.11 löst dann das Interpolationsproblem, falls zusätzlich die Bedingung (4.1) erfüllt ist, also $S(x_i) = f_i$ für alle $i = 0, \ldots, n$ gilt.

Bevor wir an die Berechnung von Splines gehen, zeigen wir eine Eigenschaft des Funktionenraums $S_{\Delta,k}$.

Satz 4.12 Sei $\Delta = \{x_0, x_1, \ldots, x_n\}$ mit $x_0 < x_1 < \ldots < x_n$ und $k \in \mathbb{N}$ gegeben. Dann ist der Raum der Splines $S_{\Delta,k}$ ein $k+n$-dimensionaler Vektorraum über $\mathbb{R}$.

Beweis: Sicherlich ist mit S_1 und S_2 auch $aS_1 + bS_2$ für a, $b \in \mathbb{R}$ wieder ein Spline, also ist $S_{\Delta,k}$ ein Vektorraum. Da die Splines linear von den Koeffizienten a_{ij} abhängen, genügt es zur Berechnung der Dimension die Anzahl der freien Parameter a_{ij} zu bestimmen. Auf dem ersten Intervall I_1 haben wir freie Wahl für P_1, also gibt es genau $k+1$ freie Parameter. Auf jedem weiteren Intervall I_i für $i \geq 2$ sind die Werte der j-ten Ableitung $P_i^{(j)}(x_{i-1}) = j!a_{ij}$ für $j = 0, \ldots, k-1$ bereits festgelegt, da die zusammengesetzte Funktion S in x_{i-1} ja stetig und $k-1$-mal stetig differenzierbar ist, es muss also

$$a_{ij} = P_{i-1}^{(j)}(x_{i-1})/j! \quad \text{für } j = 0, \ldots, k-1$$

gelten. Daher ist hier nur noch der Koeffizient a_{ik} frei wählbar, was auf den $n-1$ verbleibenden Intervallen gerade $n-1$ weitere freie Parameter ergibt, also insgesamt $k+n$. □

Tatsächlich kann man beweisen, dass die Spline-Koeffizienten a_{ij} linear voneinander abhängen, statt der im Beweis des Satzes verwendeten Koeffizienten $a_{10}, \ldots, a_{1k}, a_{2k}, \ldots, a_{nk}$ kann man also auch andere $n+k$ Koeffizienten vorgeben und die übrigen daraus berechnen.

Von besonderer praktischer Bedeutung in vielen Anwendungen sind die *kubischen Splines*, also Splines vom Grad $k = 3$; den Grund dafür werden wir etwas später besprechen. Wegen der Darstellung (4.6) von $P_i(x)$ und der Interpolationsbedingung (4.1) sind für die Splineinterpolation die Koeffizienten a_{i0}, $i = 1, 2, \ldots, n$, bereits festgelegt:

$$a_{i0} = f_{i-1} \qquad \text{für} \qquad i = 1, \ldots, n.$$

Entsprechend können wir (4.1) für $x = x_n$ sicherstellen, indem wir

$$a_{n1} = \frac{f_n - a_{n0} - a_{n2}(x_n - x_{n-1})^2 - a_{n3}(x_n - x_{n-1})^3}{x_n - x_{n-1}}.$$

setzen. Dann haben wir insgesamt genau $n+1$ Koeffizienten festgelegt und sichergestellt, dass die Interpolationsbedingung (4.1) für gegebene Daten (x_i, f_i) mit $x_0 < x_1 < \ldots < x_n$ erfüllt ist. Da der Vektorraum $S_{\Delta,3}$ Dimension $n+3$ hat, verbleiben also 2 weitere Koeffizienten, die durch geeignete Bedingungen festgelegt werden müssen, um einen eindeutigen interpolierenden Spline zu erhalten. Diese werden typischerweise in Form von Randbedingungen, also Bedingungen an S oder an Ableitungen von S in den Punkten x_0 und x_n formuliert:

(a) $S''(x_0) = S''(x_n) = 0$ (“natürliche Randbedingungen”)

(b) $S'(x_0) = S'(x_n)$ und $S''(x_0) = S''(x_n)$ (“periodische Randbedingungen”)

(c) $S'(x_0) = f'(x_0)$ und $S'(x_n) = f'(x_n)$ ("hermite'sche Randbedingungen", nur sinnvoll bei der Funktionsinterpolation)

Dann gibt es genau einen kubischen Spline $S \in S_{\Delta\,3}$, der das Interpolationsproblem (4.1) löst und die entsprechende Randbedingung (a), (b) oder (c) erfüllt.

Kubische Splines werden in Anwendungen wie z.B. der Computergrafik bevorzugt verwendet, und wir wollen als nächstes den Grund dafür erläutern. Ein Kriterium zur Wahl der Ordnung eines Splines — speziell bei grafischen Anwendungen, aber auch bei "klassischen" Interpolationsproblemen — ist, dass die Krümmung der interpolierenden Kurve möglichst klein sein soll. Die Krümmung einer Kurve $y(x)$ in einem Punkt x ist gerade gegeben durch die zweite Ableitung $y''(x)$. Die Gesamtkrümmung für alle $x \in [x_0, x_n]$ kann nun auf verschiedene Arten gemessen werden, hier verwenden wir die L_2-Norm $\|\cdot\|_2$ für quadratisch integrierbare Funktionen, die für $g : [x_0, x_n] \to \mathbb{R}$ durch

$$\|g\|_2 := \left(\int_{x_0}^{x_n} g^2(x) dx \right)^{\frac{1}{2}}$$

gegeben ist. Die Krümmung einer zweimal stetig differenzierbaren Funktion $y : [x_0, x_n] \to \mathbb{R}$ über dem gesamten Intervall kann also mittels $\|y''\|_2$ gemessen werden. Hierfür gilt der folgende Satz.

Satz 4.13 Sei $S : [x_0, x_n] \to \mathbb{R}$ ein die Daten (x_i, f_i), $i = 0, \ldots, n$ interpolierender kubischer Spline, der eine der Randbedingungen (a)–(c) erfüllt. Sei $y : [x_0, x_n] \to \mathbb{R}$ eine zweimal stetig differenzierbare Funktion, die ebenfalls das Interpolationsproblem löst und die gleichen Randbedingungen wie S erfüllt. Dann gilt

$$\|S''\|_2 \leq \|y''\|_2.$$

Beweis: Setzen wir die offensichtliche Gleichung $y'' = S'' + (y'' - S'')$ in die quadrierte Norm $\|y''\|_2^2$ ein, so folgt

$$\begin{aligned} \|y''\|_2^2 &= \int_{x_0}^{x_n} (y''(x))^2 dx \\ &= \underbrace{\int_{x_0}^{x_n} (S''(x))^2 dx}_{=\|S''\|_2^2} + \underbrace{2 \int_{x_0}^{x_n} S''(x)(y''(x) - S''(x)) dx}_{=:J} + \underbrace{\int_{x_0}^{x_n} (y''(x) - S''(x))^2 dx}_{\geq 0} \\ &\geq \|S''\|_2^2 + J. \end{aligned}$$

Wir betrachten nun den Term J genauer. Aus jeder der drei Randbedingungen folgt die Gleichung

$$\Big[S''(x)(y'(x) - S'(x)) \Big]_{x=x_0}^{x_n} = S''(x_n)(y'(x_n) - S'(x_n)) - S''(x_0)(y'(x_0) - S'(x_0)) = 0. \quad (4.7)$$

Mit partieller Integration gilt

$$\int_{x_0}^{x_n} S''(x)(y''(x) - S''(x)) dx = \Big[S''(x)(y'(x) - S'(x)) \Big]_{x=x_0}^{x_n} - \int_{x_0}^{x_n} S'''(x)(y'(x) - S'(x)) dx$$

Hierbei ist der erste Summand wegen (4.7) gleich Null. Auf jedem Intervall $I_i = [x_{i-1}, x_i]$ ist $S(x) = P_i(x)$ ein kubisches Polynom, weswegen $S'''(x) \equiv d_i$ konstant für $x \in I_i$ ist. Also folgt für den zweiten Summanden

$$\begin{aligned}\int_{x_0}^{x_n} S'''(x)(y'(x) - S'(x))dx &= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} d_i(y'(x) - S'(x))dx \\ &= \sum_{i=1}^{n} d_i \int_{x_{i-1}}^{x_i} y'(x) - S'(x)dx \\ &= \sum_{i=1}^{n} d_i \underbrace{[(y(x_i) - y(x_{i-1}) - S(x_i) + S(x_{i-1})]}_{=0,\, \text{da } y(x_i)=S(x_i) \text{ und } y(x_{i-1})=S(x_{i-1})} = 0\end{aligned}$$

Damit erhalten wir $J = 0$ und folglich die Behauptung. □

Diese Eigenschaft erklärt auch den Namen Spline: Ein "Spline" ist im Englischen eine dünne Holzlatte. Wenn man diese so verbiegt, dass sie vorgegebenen Punkten folgt (diese also "interpoliert"), so ist auch bei dieser Latte die Krümmung, die hier näherungsweise die notwendige "Biegeenergie" beschreibt, minimal — zumindest für kleine Auslenkungen der Latte.

Wir kommen nun zur praktischen Berechnung der Spline-Koeffizienten für kubische Splines. Hierbei gibt es verschiedene Vorgehensweisen: man kann z.B. direkt ein lineares Gleichungssystem für die $4n$ Koeffizienten a_{ij} für $i = 1, \ldots, n$ aufstellen, was aber wenig effizient ist. Alternativ kann man geschickt gewählte Basis-Funktionen für den Vektorraum $S_{\Delta,3}$ verwenden (sogenannte B-Splines), und S in dieser Basis berechnen; dieser Ansatz wird im Buch von Deuflhard/Hohmann beschrieben. Dies führt auf ein n-dimensionales lineares Gleichungssystem mit tridiagonaler Matrix A. Es werden bei diesem Verfahren allerdings nicht die Koeffizienten a_{ij} berechnet, sondern die Koeffizienten bezüglich der B-Spline-Basis, die Auswertung von S muss demnach ebenfalls über diese Basisfunktionen erfolgen.

Hier stellen wir eine weitere Variante vor, mit der direkt die Koeffizienten a_{ij} berechnet werden, so dass S über die Darstellung in Definition 4.11 ausgewertet werden kann, was z.B. der Vorgehensweise in MATLAB entspricht. Wir kommen hierbei ebenfalls auf ein n-dimensionales lineares Gleichungssystem mit tridiagonaler Matrix A, so dass der Aufwand der Berechnung von der Ordnung $O(n)$ ist. Wir betrachten zuerst die natürlichen Randbedingungen.

Hierzu definieren wir zunächst die Werte

$$f_i'' := S''(x_i) \quad \text{und} \quad h_i := x_i - x_{i-1}$$

für $i = 0, \ldots, n$ bzw. $i = 1, \ldots, n$. Aus der Interpolationsbedingung und der geforderten Stetigkeit der zweiten Ableitung erhält man 4 Gleichungen für die Teilpolynome P_i:

$$P_i(x_{i-1}) = f_{i-1}, \;\; P_i(x_i) = f_i, \;\; P_i''(x_{i-1}) = f_{i-1}'', \;\; P_i''(x_i) = f_i''. \tag{4.8}$$

Löst man diese — unter Ausnutzung der Ableitungsregeln für Polynome — nach den a_{ij}

auf, so erhält man

$$\begin{aligned} a_{i\,0} &= f_{i-1} \\ a_{i\,1} &= \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(f_i'' + 2f_{i-1}'') \\ a_{i\,2} &= \frac{f_{i-1}''}{2} \\ a_{i\,3} &= \frac{f_i'' - f_{i-1}''}{6h_i}. \end{aligned}$$

Da die Werte h_i und f_i ja direkt aus den Daten verfügbar sind, müssen lediglich die Werte f_i'' berechnet werden. Da aus den natürlichen Randbedingungen sofort $f_0'' = 0$ und $f_n'' = 0$ folgt, brauchen nur die Werte $f_1'', \ldots, f_{n-1}''$ berechnet werden.

Beachte, dass wir in (4.8) bereits die Bedingungen an P_i und P_i'' in den Stützstellen verwendet haben. Aus den noch nicht benutzten Gleichungen für die ersten Ableitungen erhält man nun die Gleichungen für die f_i'': Aus $P_i'(x_i) = P_{i+1}'(x_i)$ erhält man

$$a_{i\,1} + 2a_{i\,2}(x_i - x_{i-1}) + 3a_{i\,3}(x_i - x_{i-1})^2 = a_{i+1\,1}$$

für $i = 1, \ldots, n-1$. Indem man hier die Werte f_i'' und h_i gemäß den obigen Gleichungen bzw. Definitionen einsetzt, erhält man

$$h_i f_{i-1}'' + 2(h_i + h_{i+1}) f_i'' + h_{i+1} f_{i+1}'' = 6\frac{f_{i+1} - f_i}{h_{i+1}} - 6\frac{f_i - f_{i-1}}{h_i} =: \delta_i$$

für $i = 1, \ldots, n-1$. Dies liefert genau $n-1$ Gleichungen für die $n-1$ Unbekannten $f_1'', \ldots, f_{n-1}''$. In Matrixform geschrieben erhalten wir so das Gleichungssystem

$$\begin{pmatrix} 2(h_1+h_2) & h_2 & 0 & \cdots & \cdots & 0 \\ h_2 & 2(h_2+h_3) & h_3 & \ddots & & \vdots \\ 0 & h_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & h_{n-1} \\ 0 & \cdots & \cdots & 0 & h_{n-1} & 2(h_{n-1}+h_n) \end{pmatrix} \begin{pmatrix} f_1'' \\ f_2'' \\ \vdots \\ \vdots \\ f_{n-1}'' \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \vdots \\ \delta_{n-1} \end{pmatrix}$$

Zur Berechnung des Interpolationssplines löst man also zunächst dieses Gleichungssystem und berechnet dann gemäß der obigen Formel die Koeffizienten $a_{i\,j}$ aus den f_k''.

Für äquidistante Stützstellen, also $x_k - x_{k-1} = h_k = h$ für alle $k = 1, \ldots, n$, kann man beide Seiten durch h teilen, und erhält so das Gleichungssystem

$$\begin{pmatrix} 4 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & \vdots \\ 0 & 1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 \\ \vdots & & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} f_1'' \\ f_2'' \\ \vdots \\ \vdots \\ f_{n-1}'' \end{pmatrix} = \begin{pmatrix} \tilde{\delta}_1 \\ \tilde{\delta}_2 \\ \vdots \\ \vdots \\ \tilde{\delta}_{n-1} \end{pmatrix}$$

mit $\tilde{\delta_k} = \delta_k/h$, welches ein Beispiel für ein lineares Gleichungssystem mit (offensichtlich) diagonaldominanter Matrix ist.

Für andere Randbedingungen verändert sich dieses Gleichungssystem entsprechend.

Zum Abschluss wollen wir noch kurz auf den Interpolationsfehler bei der Splineinterpolation eingehen. Die Analyse dieses Fehlers ist recht langwierig, das Ergebnis, das wir hier ohne Beweis geben, allerdings sehr leicht darzustellen. Der folgende Satz wurde von C.A. Hall und W.W. Meyer [5] bewiesen.

Satz 4.14 Sei $S \in S_{\Delta,3}$ der interpolierende Spline einer 4 mal stetig differenzierbaren Funktion f mit hermite'schen Randbedingungen und Stützstellen $\Delta = \{x_0, \ldots, x_n\}$. Dann gilt für $h = \max_k(x_k - x_{k-1})$ die Abschätzung

$$\|f - S\|_\infty \leq \frac{5}{384} h^4 \|f^{(4)}\|_\infty$$

Literaturverzeichnis

[1] Bollhöfer, Matthias ; Mehrmann, Volker: *Numerische Mathematik: Eine projektorientierte Einführung für Ingenieure, Mathematiker und Naturwissenschaftler.* 2013

[2] Bryan, Kurt ; Leise, Tanya: The $25,000,000,000 eigenvector: The linear algebra behind Google. In: *Siam Review* 48 (2006), Nr. 3, S. 569–581

[3] Deuflhard, P. ; Hohmann, A.: *Numerische Mathematik. I: Eine algorithmisch orientierte Einführung.* 3. Auflage. Berlin : de Gruyter, 2002

[4] Grüne, L.: *Einführung in die Numerische Mathematik.* Vorlesungsskript; 5.Auflage. `http://num.math.uni-bayreuth.de/de/team/Gruene\_Lars/lecture\_notes`

[5] Hall, C. A. ; Meyer, W. W.: Optimal Error Bounds for Cubic Spline Interpolation. In: *J. Approx. Theory* 16 (1976), S. 105–122

[6] Huckle, T. ; Schneider, S.: *Numerische Methoden - Eine Einführung für Informatiker, Naturwissenschaftler, Ingenieure und Mathematiker.* 2.Auflage. Springer, 2006

[7] Schwarz, H. R. ; Köckler, N.: *Numerische Mathematik.* 5. Auflage. Stuttgart : B. G. Teubner, 2004

[8] Stoer, J.: *Numerische Mathematik I.* 9. Auflage. Springer Verlag, Heidelberg, 2005