Freie Universität Berlin

Institut für Mathematik

Bachelorarbeit

Verfahren zur Berechnung und Approximation von Konditionszahlen einer Matrix

Name: Fabian Weber
Matrikelnummer: 5551854
Betreuer: Prof. Dr. Volker John
Zweitgutachter: Dr. Alfonso Caiazzo

Berlin, den 1. Dezember 2022

Selbstständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.
Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.
Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1 Motivation

Da man bei numerischen Verfahren oftmals mit ungenauen Daten arbeitet, ist es von Bedeutung, sich die Auswirkung von Fehlern in den Daten anzuschauen. Die Empfindlichkeit der Lösung eines Problems gegenüber kleinen Änderungen in den Daten heißt Kondition des Problems. Wobei das Lösen eines Problems dem Auswerten einer Funktion $F : D \to W$ für Datenwerte $x \in D$ entspricht.

1.1 Kondition eines linearen Gleichungssystems

Nun wollen wir die Kondition des Problems der Lösung eines lineares Gleichungssystem $Ax = b$ mit $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$ bestimmen. Hierzu ändern wir die die Daten minimal und erhalten ein gestörtes System [1]:

$$(A + \epsilon F)x(\epsilon) = b + \epsilon f, x(0) = x, F \in \mathbb{R}^{n,n}, f \in \mathbb{R}^n,$$

wobei $\epsilon > 0$ hinreichend klein sein soll. Um die Kondition zu bestimmen, wollen wir nun den relativen Fehler $\frac{|x(\epsilon)-x|}{|x|}$ berechnen. Wenn A nichtsingulär ist, ist $A + \epsilon F$ invertierbar [2] und somit $x(\epsilon)$ in einer Umgebung von 0 differenzierbar. Die Taylor-Entwicklung von $x(\epsilon)$ um $\epsilon = 0$ liefert

$$x(\epsilon) = x + \epsilon x'(0) + \mathcal{O}(\epsilon^2).$$

Mit der Produktregel folgt

$$Fx(\epsilon) + (A + \epsilon F)x'(\epsilon) = f$$

und somit für $\epsilon = 0$

$$x'(0) = A^{-1}(f - Fx).$$

Also folgt für jede Vektornorm und zugehörige Matrixnorm

$$\|x(\epsilon) - x\| = \|\epsilon A^{-1}(f - Fx) + \mathcal{O}(\epsilon^2)\| \leq \epsilon\|A^{-1}\|(\|f\| + \|F\|\|x\|) + \mathcal{O}(\epsilon^2),$$

womit sich

$$\frac{\|x(\epsilon) - x\|}{\|x\|} \leq \epsilon\|A^{-1}\|\left(\frac{\|f\|}{\|x\|} + \|F\|\right) + \mathcal{O}(\epsilon^2) = \epsilon\|A^{-1}\|\left(\frac{\|f\|\|b\|}{\|x\|\|b\|} + \frac{\|F\|\|A\|}{\|A\|}\right) + \mathcal{O}(\epsilon^2)$$

ergibt. Mit der Konsistenzungleichung $\|b\| \leq \|A\|\|x\|$ folgt nun

$$\frac{\|x(\epsilon) - x\|}{\|x\|} \leq \epsilon\|A^{-1}\|\|A\|\left(\frac{\|f\|}{\|b\|} + \frac{\|F\|}{\|A\|}\right) + \mathcal{O}(\epsilon^2) = \|A\|\|A^{-1}\|(r_A + r_b) + \mathcal{O}(\epsilon^2),$$

wobei

$$r_A = \frac{\epsilon\|F\|}{\|A\|}, r_b = \frac{\epsilon\|f\|}{\|b\|}$$

die relativen Fehler in A und b sind.
Wir sehen also, dass $\|A\|\|A^{-1}\|$ ein Verstärkungsfaktor für den relativen Fehler in den Daten ist und somit die Kondition des linearen Gleichungssystems beschreibt.

1.1.1 Definition der Konditionszahl

Wir definieren also entsprechend die Konditionszahl einer Matrix A bezüglich der Norm $\|\cdot\|_p$ als

$$\kappa_p(A) = \|A\|_p\|A^{-1}\|_p$$

und wir setzen $\kappa_p(A) = \infty$ für A singulär.

1.1.2 Submultiplikativität der Konditionszahl

Seien $A, B \in \mathbb{R}^{n\times n}$ und $\|\cdot\|_p$ eine submultiplikative Matrixnorm. Dann gilt

$$\begin{aligned}\kappa_p(AB) &= \|AB\|_p\|(AB)^{-1}\|_p \\ &= \|AB\|_p\|B^{-1}A^{-1}\|_p \\ &\leq \|A\|_p\|B\|_p\|B^{-1}\|_p\|A^{-1}\|_p \\ &= \kappa_p(A)\kappa_p(B).\end{aligned}$$

1.2 Auswirkung der Konditionszahl auf die Stabilität des Gauß-Verfahrens

Wir betrachten das Gleichungssystem $Ax = b$ mit $A \in \mathbb{R}^{n\times n}$ regulär und formulieren den Gaußschen Algorithmus wie folgt [2]

$$A^{(k)} = (I - G_k)A^{(k-1)}, \quad A^{(0)} = A, \quad b^{(k)} = (I - G_k)b^{(k-1)}, \quad b^{(0)} = b,$$
$$x = R^{-1}z, \quad R = A^{(n-1)}, \quad z = b^{(n-1)}.$$

Wobei $I \in \mathbb{R}^n$ die Einheitsmatrix ist und G_k wie folgt definiert wird

$$(G_k)_{ij} = \begin{cases} l_{ik} = \frac{a_{ik}^{(k-1}}{a_{kk}^{(k-1)}}, & \text{falls } i = k+1, \ldots, n,\ j = k, \\ 0, & \text{sonst.}\end{cases}$$

Wir betrachten nun die Rundungsfehler, die bei der praktischen Durchführung auftreten und erhalten folgenden Algorithmus

$$\tilde{A}^{(k)} = \mathrm{rd}\left((I - G_k)\tilde{A}^{(k-1)}\right), \quad \tilde{A}^{(0)} = A, \quad \tilde{b}^{(k)} = \mathrm{rd}\left((I - G_k)b^{(k-1)}\right), \quad \tilde{b}^{(0)} = b,$$
$$\tilde{x} = \tilde{R}^{-1}z, \quad \tilde{R} = \tilde{A}^{(n-1)}, \quad \tilde{z} = \tilde{b}^{(n-1)}.$$

Mit dem Ergebnis aus Kapitel 1.1 und der Submultiplikativität der Konditionszahl gilt

$$\begin{aligned}\frac{\|x-\tilde{x}\|}{\|x\|} &\leq \kappa(R)\left(\frac{\|R-\tilde{R}\|}{\|R\|}+\frac{\|z-\tilde{z}\|}{\|z\|}\right)+\mathcal{O}(\|R-\tilde{R}\|^2)\\ &\leq \left(\kappa(A)\prod_{k=1}^{n-1}\kappa(I-G_k)\right)\left(\frac{\|R-\tilde{R}\|}{\|R\|}+\frac{\|z-\tilde{z}\|}{\|z\|}\right)+\mathcal{O}(\|R-\tilde{R}\|^2),\end{aligned}$$

wobei $\|\cdot\|$ eine beliebige Vektornorm beziehungsweise die von der Vektornorm induzierte Matrixnorm ist und $\kappa(\cdot)$ die Konditionszahl zu dieser Norm bezeichnet. Wir können $\frac{\|R-\tilde{R}\|}{\|R\|}$ und $\frac{\|z-\tilde{z}\|}{\|z\|}$ rekursiv berechnen. Hierzu bemerken wir zunächst

$$\begin{aligned}&\|A^{(k-1)}\| = \|(I-G_k)^{-1}A^{(k)}\| \leq \|(I-G_k)^{-1}\|\|A^{(k)}\|\\ \iff\quad &\|A^{(k)}\|^{-1} \leq \|(I-G_k)^{-1}\|\|A^{(k-1)}\|^{-1}.\end{aligned}$$

Wenn wir im weiterem die Maximums-Norm $\|\cdot\|_\infty$ betrachten, dann gilt

$$\frac{\|B-\mathrm{rd}(B)\|_\infty}{\|B\|_\infty} \leq eps, \quad \forall B\in\mathbb{R}^{n\times n}, \quad \frac{\|b-\mathrm{rd}(b)\|_\infty}{\|b\|_\infty} \leq eps, \quad \forall b\in\mathbb{R}^n,$$

wobei *eps* die Maschinengenauigkeit bezeichnet. Für $k=1,\ldots,n-1$ folgt

$$\begin{aligned}\frac{\|A^{(k)}-\tilde{A}^{(k)}\|_\infty}{\|A^{(k)}\|_\infty} &= \frac{\|A^{(k)}-rd\left((I-G_k)\tilde{A}^{(k-1)}\right)\|_\infty}{\|A^{(k)}\|_\infty}\\ &\leq \frac{\|A^{(k)}-(I-G_k)\tilde{A}^{(k-1)}\|_\infty}{\|A^{(k)}\|_\infty}+eps\frac{\|(I-G_k)\tilde{A}^{(k-1)}\|_\infty}{\|A^{(k)}\|_\infty}\\ &\leq \frac{\|A^{(k)}-(I-G_k)\tilde{A}^{(k-1)}\|_\infty}{\|A^{(k)}\|_\infty}+eps\frac{\|(I-G_k)(-A^{(k-1)}+\tilde{A}^{(k-1)})+(I-G_k)A^{(k-1)}\|_\infty}{\|A^{(k)}\|_\infty}\\ &\leq \frac{\|(I-G_k)(A^{(k-1)}-\tilde{A}^{(k-1)})\|_\infty}{\|A^{(k)}\|_\infty}+eps\frac{\|(I-G_k)(-A^{(k-1)}+\tilde{A}^{(k-1)})\|_\infty+\|A^{(k)}\|_\infty}{\|A^{(k)}\|_\infty}\\ &\leq \frac{\|(I-G_k)\|_\infty\|A^{(k-1)}-\tilde{A}^{(k-1)}\|_\infty}{\|A^{(k)}\|_\infty}+eps\frac{\|(I-G_k)(A^{(k-1)}-\tilde{A}^{(k-1)})\|_\infty}{\|A^{(k)}\|_\infty}+eps\\ &\leq \kappa(I-G_k)(1+eps)\frac{\|A^{(k-1)}-\tilde{A}^{(k-1)}\|_\infty}{\|A^{(k-1)}\|_\infty}+eps.\end{aligned}$$

Somit folgt also nun

$$\begin{aligned}\frac{\|R-\tilde{R}\|_\infty}{\|R\|_\infty} = \frac{\|A^{(n-1)}-\tilde{A}^{(n-1)}\|_\infty}{\|A^{(n-1)}\|_\infty} &\leq eps\sum_{j=1}^{n-1}\prod_{k=j+1}^{n-1}\kappa(I-G_k)(1+eps)+\mathcal{O}(eps)\\ &\leq eps\sum_{j=1}^{n-1}\prod_{k=j+1}^{n-1}\kappa(I-G_k)+\mathcal{O}(eps)\end{aligned}$$

und analog

$$\frac{\|z-\tilde{z}\|_\infty}{\|z\|_\infty} \leq eps \sum_{j=1}^{n-1} \prod_{k=j+1}^{n-1} \kappa(I-G_k) + \mathcal{O}(eps).$$

Somit hängt der Fehler der Lösung mittels des Gauß-Verfahrens von $\kappa(A)$ und von $\kappa(I-G_k)$ ab.
Um $\kappa(I-G_k)$ möglichst klein zu haben, können wir Spaltenpivoting anwenden, wodurch nach [2] gilt

$$|l_{ik}| = \left| \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} \right| \leq 1, \quad i = k+1, \ldots, n, \quad k = 1, \ldots, n-1.$$

Mit $\kappa(I-G_k) = \|I-G_k\|_\infty \cdot \|(I-G_k)^{-1}\|_\infty = (1+\max_{i=k+1,\ldots,n} |l_{ik}|)^2$ folgt nun

$$\begin{aligned}
\frac{\|x-\tilde{x}\|_\infty}{\|x\|_\infty} &\leq \left(\kappa(A) \prod_{k=1}^{n-1} \kappa(I-G_k) \right) \left(\frac{\|R-\tilde{R}\|}{\|R\|} + \frac{\|z-\tilde{z}\|}{\|z\|} \right) + \mathcal{O}(\|R-\tilde{R}\|^2) \\
&\leq \left(\kappa(A) \prod_{k=1}^{n-1} \kappa(I-G_k) \right) \cdot 2eps \left(\sum_{j=1}^{n-1} \prod_{k=j+1}^{n-1} \kappa(I-G_k) \right) + \mathcal{O}(eps) \\
&\leq 2\kappa(A) 2^{n-1} (2^{n-1}-1) eps + \mathcal{O}(eps).
\end{aligned}$$

2 Berechnung der Konditionszahl mittels der Inversen

Sei $A \in \mathbb{R}^{n\times n}$, wir nehmen an die LR-Zerlegung $A = LR$ sei bekannt, dann gilt $A^{-1} = (LR)^{-1} = R^{-1}L^{-1}$. Um L^{-1} beziehungsweise R^{-1} zu berechnen, müssen wir jeweils n Dreieckssysteme lösen. Das Lösen von Dreieckssystemen hat eine Komplexität von $\mathcal{O}(n^2)$. Somit hat das Berechnen von L^{-1} beziehungsweise R^{-1} Kosten von $\mathcal{O}(n^3)$. Die Berechnung des Produkts $R^{-1}L^{-1}$ hat wieder eine Komplexität von $\mathcal{O}(n^3)$. Hinzu kommen noch die Kosten für die Berechnung der Normen $\|A\|$ und $\|A^{-1}\|$.
Die direkte Berechnung der Konditionszahl ist somit ziemlich teuer und es lohnt sich, Verfahren anzuschauen mit denen sich die Berechnung oder Abschätzung unter bestimmten Bedingungen effizienter gestaltet.
Zunächst stellen wir fest, dass sich die Inverse in folgenden Fällen effizienter berechnen lässt.

2.1 Sonderfall bidiagonale Matrix

Sei $A \in \mathbb{R}^{n\times n}$ bidiagonal, dann gilt $A = \operatorname{diag}(a_{11}, a_{22}, \dots, a_{nn}) \cdot C$ mit

$$C = \begin{bmatrix} 1 & \frac{a_{12}}{a_{11}} & & \\ & 1 & \frac{a_{23}}{a_{22}} & \\ & & \ddots & \ddots \\ & & & 1 \end{bmatrix}.$$

Weiter gilt $A^{-1} = C^{-1}\operatorname{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \dots, \frac{1}{a_{nn}}\right)$. Setzten wir nun $c_i = -\frac{a_{i(i+1)}}{a_{ii}}$, dann erhalten wir

$$C = \begin{bmatrix} 1 & -c_1 & & \\ & 1 & -c_2 & \\ & & \ddots & \ddots \\ & & & 1 \end{bmatrix} \quad \text{und} \quad C^{-1} = \begin{bmatrix} 1 & c_1 & c_1c_2 & \dots & (c_1 \dots c_{n-1}) \\ & 1 & c_2 & \dots & (c_2 \dots c_{n-1}) \\ & & 1 & \dots & (c_3 \dots c_{n-1}) \\ & & & \ddots & \vdots \\ & & & & 1 \end{bmatrix}.$$

Somit können wir die Inverse einer bidiagonalen Matrix mit einer Komplexität von $\mathcal{O}(n^2)$ wie folgt berechnen.

```python
import numpy as np

def biDiagInv(A):
    n = np.shape(A)[0]
    A_inv = np.eye(n)
    for i in range(1, n):
        A_inv[:, i] = A_inv[:, i] + (A_inv[:, i-1] * (-A[i-1][i] / A[i-1][i-1]))
        A_inv[:, i-1] /= A[i-1][i-1]
    A_inv[n-1][n-1] /= A[n-1][n-1]
    return A_inv
```

Bei der 1-Norm $\|A\|_1 = \max_{\|x\|_1=1} \|Ax\|_1 = \max_{j=1,\dots,n} \sum_{i=1}^{m} |a_{ij}|$ können wir sogar die Konditionszahl unter Ausnutzung des Distributivgesetzes mit einer Komplexität von $\mathcal{O}(n)$ berechnen, indem wir, statt die Inverse explizit auszurechnen, sofort die Spalten aufsummieren. Womit wir folgenden Python Code für die Berechnung der Konditionszahl erhalten:

```python
import numpy as np

def directCondBiDiag(A):
    n = np.shape(A)[0]
    v = np.ones(n)
    for i in range(1, n):
        v[i] = (np.linalg.norm(A[i-1][i])/np.linalg.norm(A[i-1][i-1])) * v[i-1] + 1
        v[i-1] /= np.linalg.norm(A[i-1][i-1])
    v[n-1] /= np.linalg.norm(A[n-1][n-1])
    return np.max(v) * np.linalg.norm(A, 1)
```

Verglichen mit folgenden zwei Funktionen

```python
import numpy as np
import scipy as sc

def directCondBiDiagSc(A):
    A_inv = sc.linalg.solve_triangular(A, np.eye(A.shape[0]), check_finite=False)
    return np.linalg.norm(A_inv, 1) * np.linalg.norm(A, 1)

def directCondBiDiagInv(A):
    A_inv = biDiagInv(A)
    return np.linalg.norm(A_inv, 1) * np.linalg.norm(A, 1)
```

sehen wir folgende Laufzeit Unterschiede, wenn wir die Funktionen 500 mal auf zufälligen Matrizen ausführen.

Tabelle 1: Vergleich der benötigten Laufzeit für die Berechnung der Konditionszahl zur Norm 1 bei zufälligen bidiagonalen Matrizen

	$n = 100$	$n = 250$	$n = 500$
directCondBiDiag	$0,125$	$0,25$	$0,7188$
directCondBiDiagSc	$0,1406$	$0,6094$	$5,1719$
directCondBiDiagInv	$0,1563$	$0,8281$	$2,3438$

2.2 Sonderfall tridiagonale Matrix

Wir betrachten die nichtsinguläre tridiagonale Matrix

$$A = \begin{bmatrix} d_1 & e_1 & & & \\ c_2 & d_2 & e_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & e_{n-1} \\ & & & c_n & d_n \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

Dann gilt für die LR-Zerlegung $A = LR$

$$L = \begin{bmatrix} 1 & & & \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ & & l_n & 1 \end{bmatrix} \quad \text{und} \quad R = \begin{bmatrix} u_1 & e_1 & & \\ & \ddots & \ddots & \\ & & u_{n-1} & e_{n-1} \\ & & & u_{n-1} \end{bmatrix},$$

wobei $u_1 = d_1$, $l_i = \frac{c_i}{u_{i-1}}$, $u_i = d_i - l_i e_{i-1}$ für $i = 2, \dots n$ [3]. Also sind L und R bidiagonal, womit sich die Komplexität der Berechnung von R^{-1} und L^{-1} auf $\mathcal{O}(n^2)$ reduziert. Wir können die Berechnung von L^{-1} jedoch noch etwas beschleunigen, indem wir ausnutzen, dass die Diagonalelemente gleich Eins sind, womit bei der Funktion biDiagInv die Brüche durch die Diagonalelemente wegfallen.
Wir können weiterhin in $\mathcal{O}(n)$ eine obere Schranke für die Konditionszahl zur 1-Norm angeben. Denn mit der Submultiplikativität gilt

$$\kappa(A) = \|A\|_1 \|R^{-1}L^{-1}\|_1 \leq \|A\|_1 \|R^{-1}\|_1 \|L^{-1}\|_1.$$

Aus Kapitel 2.1 wissen wir, wie wir $\|R^{-1}\|_1$ und $\|L^{-1}\|_1$ in $\mathcal{O}(n)$ berechnen können. Diese obere Schranke ist jedoch im Allgemeinen sehr schlecht und um ein vielfaches größer als das tatsächliche Ergebnis.

3 Berechnung der Konditionszahl mittels Singulärwertzerlegung

Wir betrachten die wie folgt definierte 2-Norm $\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2$. Ein für die Berechnung der Konditionszahl zur 2-Norm wichtiges Verfahren ist die im folgenden Satz nach [4] definierte Singulärwertzerlegung.

3.1 Satz über die Existenz der Singulärwertzerlegung

Sei $A \in \mathbb{R}^{m\times n}$. Dann gibt es orthogonale Matrizen $U \in \mathbb{R}^{m\times m}$ und $V \in \mathbb{R}^{n\times n}$, sodass

$$U^T AV = \Sigma = \text{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n},$$

wobei $p = \min(m, n)$ und $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$. Wir nennen die Zerlegung $U^T AV = \Sigma$ die Singulärwertzerlegung von A und die σ_i sind die Singulärwerte von A.

Beweis:
Es gilt, dass $A^T A \in \mathbb{R}^{n\times n}$ eine symmetrische und positiv semidefinite Matrix ist. Damit existiert eine Eigenwertzerlegung von $A^T A$, also gibt es $D = \text{diag}(d_1, \ldots d_n) \in \mathbb{R}^{n\times n}$ mit $d_1 \geq d_2 \geq \cdots \geq d_n$ und eine orthogonale Matrix $V \in \mathbb{R}^{n\times n}$, sodass

$$D = V^T A^T AV = (AV)^T AV$$

gilt. Sei

$$AV = UR$$

eine QR-Zerlegung von AV mit einer orthogonalen Matrix $U \in \mathbb{R}^{m\times m}$ und einer rechten oberen Dreiecksmatrix $R \in \mathbb{R}^{m\times n}$, wobei wir U so bestimmen können, dass alle Diagonalelemente von R nichtnegativ sind. Dann erhalten wir

$$D = (AV)^T AV = (UR)^T UR = R^T U^T UR = R^T R,$$

da U orthogonal ist. Da D eine Diagonalmatrix ist und R eine obere Dreiecksmatrix ist, folgt, dass R nur auf der Diagonalen von Null verschiedene Einträge haben kann. Außerdem müssen alle Diagonalelemente von R der Größe nach geordnet sein, da die Diagonalelemente von D der Größe nach geordnet sind. Es folgt also, dass mit

$$A = URV^T$$

eine Singulärwertzerlegung von A existiert.

□

3.2 Satz zur Berechnung der Konditionszahl mittels Singulärwertzerlegung

Sei $A \in \mathbb{R}^{n\times n}$ invertierbar, dann gilt für die 2-Norm $\kappa(A) = \frac{\sigma_1(A)}{\sigma_n(A)}$, wobei $\sigma_1(A)$ der größte und $\sigma_n(A)$ der kleinste Singulärwert von A ist.

Beweis:
Betrachten wir $\|A\|_2^2$, dann gilt $\|A\|_2^2 = \max_{\|x\|_2=1} \|Ax\|_2^2 = \max_{\|x\|_2=1} \langle Ax, Ax\rangle = \max_{\|x\|_2=1} \langle A^T Ax, x\rangle$, wobei $\langle ., .\rangle$ das Standartskalarprodukt ist. Analog zum Beweis von Satz 3.1 gibt es eine Eigenwertzerlegung $D = V^T A^T AV$. Setzten wir nun $x = Vy$, folgt

$$\|A\|_2^2 = \max_{\|Vy\|_2=1} \langle A^T AVy, Vy\rangle = \max_{\|Vy\|_2=1} \langle V^T A^T AVy, y\rangle = \max_{\|Vy\|_2=1} \langle Dy, y\rangle.$$

Aufgrund der Orthogonalität von V gilt weiter $\|Vy\|_2 = \|y\|_2$ für alle $y \in \mathbb{R}^n$ und somit

$$\|A\|_2^2 = \max_{\|y\|_2=1} \langle Dy, y\rangle = \max_{\|y\|_2=1} (d_1|y_1|^2 + \cdots + d_n|y_n|^2) = d_1.$$

Denn das Maximum wird genau dann angenommen, wenn y der Einheitsvektor e_1 ist. Aus Satz 3.1 folgt $\sigma_1(A) = \sqrt{d_1}$ und somit $\|A\|_2 = \sigma_1(A)$. Analog gilt $\|A^{-1}\|_2 = \frac{1}{\sigma_n(A)}$ und somit $\kappa(A) = \frac{\sigma_1(A)}{\sigma_n(A)}$.

□

3.3 Verfahren, um die Singulärwertzerlegung zu berechnen

Wir können den Beweis von Satz 3.1 nutzen, um ein Verfahren zu Berechnung der Singulärwertzerlegung zu bekommen, jedoch hat die Berechnung von $A^T A$ eine Komplexität von $\mathcal{O}(n^3)$ und kann, dadurch dass die Singulärwerte quadriert werden, zu größeren Ungenauigkeiten führen.
Die Idee ist daher, A wie folgt zu zerlegen

$$A = U_A R V_A,$$

wobei U_A und V_A orthogonale Matrizen sind und R eine Matrix ist, für die wir die Eigenwertzerlegung

$$R^T R = B = V_B D V_B^T$$

mit orthogonaler Matrix $V_B \in \mathbb{R}^{n\times n}$ und Diagonalmatrix $D \in \mathbb{R}^{n\times n}$ mit nach der Größe sortierten Einträgen einfacher und besser berechnen können.
Denn nehmen wir eine QR-Zerlegung $RV_B = U_B S$ mit einer orthogonalen Matrix U_B und einer oberen Dreiecksmatrix S, dann gilt zunächst

$$D = V_B^T R^T R V_B = (RV_B)^T (RV) = S^T U_B^T U_B S = S^T S,$$

wodurch analog zum Beweis von Satz 3.1 S eine Diagonalmatrix mit nach der Größe geordneten Einträgen seien muss. Weiter gilt

$$A = U_A U_B S V_B^T V_A,$$

wobei $U = U_A U_B$ und $V^T = V_B^T V_A$ orthogonale Matrizen sind und wir somit unsere Singulärwertzerlegung $A = USV^T$ haben.

3.3.1 Bidiagonalisierung

Jede Matrix $A \in \mathbb{R}^{m\times n}$ besitzt eine Bidiagonalisierung [5], also eine Zerlegung

$$A = URV,$$

mit orthogonalen Matrizen $U \in \mathbb{R}^{m\times m}$ und $V \in \mathbb{R}^{n\times n}$ und einer Bidiagonalmatrix $R \in \mathbb{R}^{m\times n}$.
Sei $a_k = (0, \ldots, 0, a_{k,k}, a_{k+1,k}, \ldots, a_{m,k})$ der k-te Spaltenvektor, wobei alle Elemente oberhalb der Diagonalen durch Null ersetzt wurden. Weiter definieren wir

$$v_k = a_k + \|a_k\|_2 \cdot e^k \quad \text{und} \quad H_k = I_m - \frac{2}{v_k^T \cdot v_k} \cdot v_k \cdot v_k^T \in \mathbb{R}^{m\times m},$$

wobei e^k der k-te Einheitsvektor ist und H_k die Householder-Matrix zum Vektor v_k ist. Dann hat die Matrix $B = H_k A$ unterhalb der Diagonalen in Spalte k nur Nulleinträge. In [5] finden wir nun folgenden Algorithmus um die Bidiagonalisierung zu berechnen:
Seien $A \in \mathbb{R}^{m\times n}$, $B_1 = (b_{ij}^1) = A$ und $z = \min\{n, m\}$. Für $k = 1, \ldots, z$ führen wir nun folgende Schritte aus:

(1) Setze $a_k^L = (0, \ldots, 0, b_{k,k}^k, b_{k+1,k}^k, \ldots, b_{m,k}^k) \in \mathbb{R}^m$ und entsprechend $v_k^L = a_k^L + \|a_k^L\|_2 \cdot e^k \in \mathbb{R}^m$.

(2) Berechne die Householder-Matrix

$$H_k^L = I_m - \frac{2}{(v_k^L)^T \cdot v_k^L} \cdot v_k^L \cdot (v_k^L)^T \in \mathbb{R}^{m\times m}.$$

(3) Berechne $C_k = H_k^L \cdot B_k$ mit den Matrixeinträgen $c_{i,j}^k$.

(4) Setze $a_k^R = (0, \ldots, 0, c_{k,k+1}^k, c_{k,k+2}^k, \ldots, c_{k,n}^k) \in \mathbb{R}^n$ und entsprechend $v_k^R = a_k^R + \|a_k^R\|_2 \cdot e^{k+1} \in \mathbb{R}^n$.

(5) Berechne die Householder-Matrix

$$H_k^R = I_n - \frac{2}{(v_k^R)^T \cdot v_k^R} \cdot v_k^R \cdot (v_k^R)^T \in \mathbb{R}^{n\times n}.$$

(6) Berechne $B_{k+1} = C_k \cdot H_k^R$ mit den Matrixeinträgen $b_{i,j}^{k+1}$.

Unsere Bidiagonalisierung wird dann durch $A = URV$ mit $U = H_1^L \cdot \ldots \cdot H_z^L \in \mathbb{R}^{m\times m}$, $V = H_z^R \cdot \ldots \cdot H_1^R \in \mathbb{R}^{n\times n}$ und $R = B_z$ gegeben.

3.3.2 Komplexität der Singulärwertzerlegung

Sowohl die Bidiagonalisierung als auch die QR-Zerlegung hat ein Komplexität von $\mathcal{O}(n^3)$. Weiter kann die Eigenwertzerlegung mit einer Komplexität von $\mathcal{O}(n^3)$ ermittelt werden, womit der Algorithmus zur Berechnung der Singulärwerte eine Komplexität von $\mathcal{O}(n^3)$ hat.

4 Hagers 1-Norm Abschätzung

Als letztes Verfahren wollen wir uns die 1-Norm Abschätzung nach Hager [6] anschauen, auf dem beispielsweise der 1-Norm Abschätzer *condest* in Matlab beruht. Das Verfahren rechnet hierbei nicht die Inverse aus, sondern schätzt lediglich die 1-Norm der Inversen ab.

4.1 Benötigte Definitionen und Sätze

Zuerst notieren wir ein paar benötigte Definitionen und Sätze nach [7].

4.1.1 Definition konvexe Menge

Eine Teilmenge $S \subseteq \mathbb{R}^n$ heißt konvex, wenn für alle $x, y \in S$ und $\lambda \in [0, 1]$ gilt

$$\lambda x + (1 - \lambda)y \in S.$$

4.1.2 Definition Extremalpunkte

Eine konvexe Teilmenge F einer konvexen Menge S heißt extremal, wenn für alle $x, y \in S$ und alle $\alpha \in (0, 1)$ gilt

$$\alpha x + (1 - \alpha)y \in F \Longrightarrow x, y \in F.$$

Eine einelementige extremale Menge heißt Extremalpunkt.
Bemerkung: x ist genau dann ein Extremalpunkt, wenn auch $S \setminus \{x\}$ konvex ist.

4.1.3 Definition konvexe Funktion

Sei S eine konvexe Teilmenge des $\mathbb{R}^n$. Eine Abbildung $f : S \to \mathbb{R}$ heißt konvex auf S, wenn für alle $x, y \in S$ und $\lambda \in [0, 1]$ gilt

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y).$$

4.1.4 Satz über das Maximum einer konvexen Funktion

Seien S eine kompakte konvexe Teilmenge des $\mathbb{R}^n$ und $f : \mathbb{R}^n \to \mathbb{R}$ eine konvexe Abbildung. Dann gibt es einen Extremalpunkt von S, der ein globales Maximum für f auf S ist.

4.1.5 Definition Subgradient

Seien S eine konvexe Teilmenge des $\mathbb{R}^n$, $f : S \to \mathbb{R}$ eine konvexe Abbildung und $y \in S$. Ein Vektor s, der für alle $x \in S$ die Ungleichung

$$f(x) \geq f(y) + s^T(x - y)$$

erfüllt, heißt ein Subgradient für f in y.

4.1.6 Satz über den Subgradienten und Gradienten

Seien S eine konvexe Teilmenge des $\mathbb{R}^n$, $f : S \to \mathbb{R}$ eine konvexe Abbildung und $y \in S$. Wenn f differenzierbar in y ist, ist der Gradient von f an y ein Subgradient von f in y.

4.2 Die 1-Norm Abschätzung

Sei $B \in \mathbb{R}^{n\times n}$. Wir betrachten zunächst ein Verfahren um $\|B\|_1$ zu approximieren und passen dieses nachher für $A^{-1} = B$ an. Hierzu definieren wir $F : \mathbb{R}^n \to \mathbb{R}$ durch

$$F(x) = \|Bx\|_1 = \sum_{i=1}^{n} \Big| \sum_{j=1}^{n} b_{ij} x_j \Big|.$$

Wir sehen, es gilt

$$\|B\|_1 = \max\{F(x) : \|x\|_1 \leq 1\}.$$

Es ist klar, dass $K = \{x \in \mathbb{R}^n : \|x\|_1 \leq 1\}$ eine kompakte konvexe Menge ist. Seien $u, v \in K$ und $0 < t < 1$, dann folgt mit

$$\begin{aligned} F((1-t)u + tv) &= \|B((1-t)u + tv)\|_1 \\ &= \|B(1-t)u + Btv\|_1 \\ &\leq \|B(1-t)u\|_1 + \|Btv\|_1 \\ &= F((1-t)u) + F(tv), \end{aligned}$$

dass F eine konvexe Funktion auf K ist und somit ihr Maximum auf K an einem Extremalpunkt annimmt. Die Extremalpunkte von K sind $\{\pm e^i : i = 1, \ldots, n\}$.

Die Idee ist, dass unser Algorithmus an einem Punkt x am Rand von S anfängt und von dort aus prüft, in welche Richtung der größte Anstieg stattfindet und den korrespondierenden Extremalpunkt als nächsten Punkt auswählt. Von dort aus wird wieder geprüft in welche Richtung der größte Anstieg stattfindet. Dies wird solange wiederholt, bis wir ein lokales Maximum gefunden haben, also kein weiterer Anstieg stattfindet. Um die Wahrscheinlichkeit zu erhöhen, dass das lokale Maximum auch unser globales Maximum ist, können wir den Algorithmus mehrmals mit den Extremalpunkten, die wir noch nicht besucht haben, ausführen.

Seien $x, y \in \mathbb{R}^n$ und $\partial F(x)$ ein Subgradient von F an x dann gilt mit der Konvexität von F

$$F(y) \geq F(x) + \partial F(x)(y - x) \text{ [7]}. \tag{1}$$

Weiter gilt, dass F an x differenzierbar ist, wenn $\sum_{j=1}^{n} b_{ij} x_i \neq 0$ für alle i ist und in diesem Fall $\partial F(x)$ also

der Gradient von F an x ist. Wir definieren für $i \in \{1, \ldots, n\}$

$$\xi = \begin{cases} 1, & \text{falls } \sum_{j=1}^{n} b_{ij} x_i \geq 0, \\ -1, & \text{sonst.} \end{cases} \tag{2}$$

Dann erhalten wir mit der Kettenregel

$$\partial F(x) = \xi^T B. \tag{3}$$

Hierbei bekommen wir im Fall $(Bx)_i = 0$ für jeden Wert von $\xi_i \in [-1, 1]$ einen Subgradienten von F an x. Wenn unser Algorithmus also nun an einem Punkt x am Rand von S startet, müssen wir, um die Richtung des steilsten Anstiegs zu ermitteln, nun ein j finden, für das

$$|\partial F(x)_j| = \max_i |\partial F(x)_i| \tag{4}$$

gilt. Wenn $|\partial F(x)_j| \leq \partial F(x)x$ gilt, dann stoppt unser Algorithmus und gibt $\|Bx\|_1$ als Abschätzung für $\|B\|_1$ aus. Denn nehmen wir an es sei $(Bx)_i \neq 0$ für alle i, dann ist x ein lokales Maximum von F [6]. Wenn hingegen $|\partial F(x)_j| > \partial F(x)x$ gilt, dann folgt mit der Ungleichung (1) und mit $F(-e^j) = F(e^j)$, dass $F(e^j) > F(x)$ gilt und wir ersetzen x durch e^j. Aufgrund der strengen Monotonie von F wird jeder Punkt nur einmal besucht und der Algorithmus endet nach endlich vielen Schritten.
Zusammengefasst bekommen wir also folgenden Algorithmus:
Schritt 0: Initialisierung
Sei $x = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$.
Schritt 1: Berechnung vom Gradient oder Subgradient z von F

$$\begin{aligned} \text{Berechne } y &= Bx \\ \xi &= sign(y), \\ z &= B^T \xi. \end{aligned}$$

Schritt 2: Richtung des größten Anstiegs
Suche j, sodass $|z_j| = \max_{1 \leq i \leq n} |z_i|$.
Schritt 3: Abbruchbedingung
Falls $|z_j| \leq z^T x$ stoppe mit $\|y\|_1$ als Abschätzung für $\|B\|_1$, sonst setze $x = e^j$ und gehe zu Schritt 1.

Wenn wir also eine Abschätzung von $\|A^{-1}\|_1$ bekommen wollen, setzen wir $A^{-1} = B$ und berechnen y, z wie folgt:

$$\begin{aligned} y = A^{-1}x \quad &\Leftrightarrow \quad Ay = x, \\ z = (A^{-1})^T \xi \quad &\Leftrightarrow \quad A^T z = \xi. \end{aligned}$$

Wenn eine LR-Zerlegung von A bekannt ist, welche im Allgemeinen eine Komplexität von $\mathcal{O}(n^3)$ hat, müssen wir also in jedem Durchlauf nur vier Dreieckssysteme lösen.

4.3 Beispiel einer Matrix mit einer schlechten Approximation

Wir betrachten die bidiagonale Matrix [8]

$$B_n = \begin{bmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & 1 & \ddots & \\ & & & \ddots & 1 \\ & & & & 1 \end{bmatrix}, \quad B_n^{-1} = \begin{bmatrix} 1 & -1 & 1 & \dots & (-1)^{n+1} \\ & 1 & -1 & & \vdots \\ & & 1 & \ddots & \vdots \\ & & & \ddots & -1 \\ & & & & 1 \end{bmatrix}$$

mit n ungerade. Dann erhalten wir nach der ersten Iteration

$$\begin{aligned} y &= \left(\frac{1}{n}, 0, \frac{1}{n}, 0, \dots\right)^T, \\ \xi &= 1, \\ z &= (1, 0, 1, 0, \dots)^T, \\ z^T x &< 1. \end{aligned}$$

Somit wird als nächster Vektor e^1 ausgewählt und wir erhalten nach der zweiten Iteration

$$\begin{aligned} y &= (1, 0, 0, 0, \dots)^T, \\ \xi &= 1, \\ z &= (1, 0, 1, 0, \dots)^T, \\ z^T x &= 1. \end{aligned}$$

Unser Algorithmus gibt uns also $\|(1, 0, 0, 0, \dots)^T\|_1 = 1$ als Approximation für $\|B_n^{-1}\|_1$ zurück, jedoch gilt $\|B_n^{-1}\|_1 = n$.

4.4 Implementation und Verbesserungen des Algorithmus

Wir implementieren den Algorithmus wie folgt in Python:

```
import numpy as np
import scipy as sc
import scipy.linalg

def hager1normCond(A, L, R, x):
    steps = 0
    n = x.size
    v = np.zeros(n)
    while True:
        steps += 1
        yl = sc.linalg.solve_triangular(L, x, lower=True)
        y = sc.linalg.solve_triangular(R, yl)
```

```
        xi = 2 * (y >= 0) - 1
        zl = sc.linalg.solve_triangular(R.T, xi, lower=True)
        z = sc.linalg.solve_triangular(L.T, zl)
        j = np.argmax(z)
        if abs(z[j]) <= z.T @ x:
            return np.linalg.norm(y, 1) * np.linalg.norm(A, 1), steps, v
        x = np.zeros(n)
        v[j] = 1
        x[j] = 1
```

Im folgenden seien s die Anzahl der benötigten Durchläufe durch die Schleife bei der Berechnung mittels Hagers 1-Norm Abschätzung und x das Ergebnis der Abschätzung. Sei y das Ergebnis welches wir bei der direkten Berechnung über die Inverse erhalten, dann sei $e = 1 - \frac{x}{y}$ der relative Fehler den wir erhalten. Wir untersuchen nun den Fehler den wir mit dem Algorithmus bekommen. Hierzu haben wir den Algorithmus auf jeweils 500 zufälligen Matrizen unterschiedlicher Größe ausgeführt.

Tabelle 2: Einmaliges Ausführen von Hagers Algorithmus

	$\varnothing s$	Max s	$\varnothing e$	Max e	Anteil an $e > 0,1$
$n = 100$	$2,134$	4	$0,0984$	$0,8156$	$36,9\%$
$n = 200$	$2,165$	5	$0,0799$	$0,5843$	$30,2\%$
$n = 500$	$2,17$	5	$0,0774$	$0,6292$	31%

Wir beobachten, dass der Fehler nicht von der Größe der Matrix abzuhängen scheint und der Algorithmus im allgemeinen kein zuverlässiges Ergebnis liefert. Weiter beobachten wir, dass wir in unseren Tests immer in maximal fünf Schritten fertig waren. Es empfiehlt sich also statt einer Endlosschleife den Algorithmus nach fünf Schritten zu terminieren, da in sehr seltenen Fällen Rundungsfehler in der Abbruchbedingung dazu führen können, dass der Algorithmus nicht terminiert [8].
Wir hatten schon erwähnt, dass ein mehrmaliges Ausführen des Algorithmus auf unterschiedlichen Startvektoren ein besseres Ergebnis liefern kann. Wir untersuchen nun also, wie sich ein mehrmaliges Ausführen auf das Ergebnis auswirkt und wie wir am besten die Startvektoren wählen sollten. Eine Idee ist, sich zu merken welche Einheitsvektoren wir schon durchlaufen haben und die entsprechenden Komponenten auf Null zu setzen womit wir folgenden Code erhalten:

```
def hager1normCondMultipleTimes(A, L, R, i):
    n = np.shape(A)[0]
    x = np.ones(n) / n
    results = []
    sum_steps = 0
    for j in range(0, i):
        result, steps, c = hager1normCond(A, L, R, x)
        results.append(result)
        sum_steps += steps
        x = (np.ones(n) - c) / (n - sum(c))
    return max(results), sum_steps
```

Eine andere Idee ist einfach immer zufällige Startvektoren zu nehmen womit wir einen nicht deterministischen Algorithmus erhalten. Der Code hierfür ist:

```
def hager1normCondMultipleTimesRandom(A, L, R, i):
    n = np.shape(A)[0]
    results = []
    sum_steps = 0
    for j in range(0, i):
        x = np.random.randint(2, size=n)
        x = x / np.linalg.norm(x, 1)
        result, steps, c = hager1normCond(A, L, R, x)
        results.append(result)
        sum_steps += steps
    return max(results), sum_steps
```

Um beide Methoden zu vergleichen, wählen wir im folgendem $n = 200$, da wir schon gesehen haben, dass die Größe nicht relevant ist.

Tabelle 3: Mehrmaliges Ausführen von Hagers Algorithmus mit unterschiedlichen Startvektoren

	$\varnothing s$	Max s	$\varnothing e$	Max e	Anteil an $e > 0,1$
1 mal Ausführen	$2,165$	5	$0,0799$	$0,5843$	$30,2\%$
2 mal Ausführen	$4,190$	6	$0,0155$	$0,6703$	$4,8\%$
3 mal Ausführen	$6,194$	9	$0,0099$	$0,6288$	$3,4\%$
8 mal Ausführen	$16,248$	20	$0,0035$	$0,3699$	$1,2\%$
1 mal Ausführen mit Zufallszahlen	$2,134$	4	$0,0984$	$0,8156$	$36,9\%$
2 mal Ausführen mit Zufallszahlen	$4,274$	7	$0,0331$	$0,3659$	$14,8\%$
3 mal Ausführen mit Zufallszahlen	$6,408$	10	$0,0231$	$0,4052$	$9,2\%$
8 mal Ausführen mit Zufallszahlen	$17,136$	23	$0,0014$	$0,1556$	$0,4\%$

Wir beobachten, dass die erste Methode bei wenigen Wiederholungen zunächst besser ist, aber bei vielen Wiederholungen die Methode mit den Zufallszahlen die bessere ist. Grund hierfür ist, dass, wenn wir bei der ersten Methode zweimal nacheinander durch dieselben Vektoren laufen, der Code festhängt und immer wieder denselben Startvektor auswählt. Kombinieren wir nun die beiden Methoden, indem wir immer dann, wenn der Code festhängt, einen zufälligen Vektor auswählen, erhalten wir folgenden Code:

```
def hager1normCondMultipleTimesCombined(A, L, R, i):
    n = np.shape(A)[0]
    x = np.ones(n) / n
    results = []
    sum_steps = 0
    used_vectors = np.zeros(n)
    for j in range(0, i):
        result, steps, c = hager1normCond(A, L, R, x)
        results.append(result)
        sum_steps += steps
```

```
        used = np.logical_or(used_vectors, c)
        if np.array_equiv(used_vectors, used):
            x = np.random.randint(2, size=n)
            x = x / np.linalg.norm(x, 1)
        else:
            x = (np.ones(n) - used) / (n - sum(used))
        used_vectors = used
    return max(results), sum_steps
```

Tabelle 4: Mehrmaliges Ausführen von Hagers Algorithmus mit hager1normCondMultipleTimesCombined

	$\varnothing s$	Max s	$\varnothing e$	Max e	Anteil an $e > 0,1$
1 mal Ausführen	$2,165$	5	$0,0799$	$0,5843$	$30,2\%$
2 mal Ausführen	$4,190$	6	$0,0155$	$0,6703$	$4,8\%$
3 mal Ausführen	$6,492$	9	$0,0049$	$0,2979$	$1,8\%$
8 mal Ausführen	$17,250$	25	$0,0008$	$0,1363$	$0,2\%$

Auf dieser Weise bekommen wir also, wenn wir einen linearen Faktor an unsere Laufzeit hängen, ein sehr zuverlässiges Ergebnis.
Sei $A \in \mathbb{R}^{n\times n}$ und x unsere Abschätzung für $\|A^{-1}\|$ die wir von Hagers Algorithmus erhalten, dann schlägt Higham [8] weiter vor, dass wir $\max\{x, \frac{2|c|_1}{3n}\}$ am Ende wählen, wobei c das Ergebnis vom folgendem Gleichungssystem ist:

$$Ac = b \quad \text{mit}$$
$$b_i = (-1)^{i+1}\Big(1 + \frac{i-1}{n-1}\Big).$$

Der Vektor b soll die Wahrscheinlichkeit verringern, dass große Elemente von A sich gegenseitig auslöschen. Wir testen dies aus, hierzu testen wir wieder 500 zufällige Matrizen mit $n = 100$ und führen Hagers Algorithmus einmal aus.

Tabelle 5: Vergleich am Ende

	$\varnothing e$ ohne Vergleich am Ende	$\varnothing e$ mit Vergleich am Ende
volle Matrix	$0,0905$	$0,0905$
untere Dreiecksmatrix	$0,0452$	$0,0452$
Bidiagonalmatrix	$0,0276$	$0,0276$
Tridiagonalmatrix	$0,0426$	$0,0426$

Wir sehen also, dass der Vergleich am Ende bei Zufallszahlen bei unseren Tests keine besseren Werte geliefert hat. Wenn wir uns jedoch das Beispiel 4.3 mit $n = 99$ anschauen, dann gibt uns ein einmaliges Ausführen von Hagers Algorithmus als Abschätzung für die Konditionszahl eine 2 obwohl der tatsächliche Wert eine 198 ist. Mit dem Vergleich am Ende erhalten wir $65,3534$, was eine Verbesserung ist, die jedoch beim mehrmaligen Ausführen von Hagers Algorithmus schnell weiter verbessert wird.

5 Praktischer Vergleich der unterschiedlichen Verfahren und Fazit

Im folgenden wollen wir uns die für die direkte Berechnung mittels der Inversen (1- und 2-Norm), LR-Zerlegung, Berechnung mittels der Singulärwertzerlegung (SVD) und dem einmaligen Ausführen von Hagers Algorithmus benötigte Laufzeit anschauen. Hierfür führen wir die Algorithmen 500 mal auf zufälligen Matrizen aus und nehmen die Summe von den Einzellaufzeiten.

Tabelle 6: Vergleich der Laufzeiten unterschiedlicher Verfahren zur Berechnung der Konditionszahl zur 2-Norm

	Direkt (2-Norm)	LR-Zerlegung	SVD
$n = 100$, allgemeine Matrix	$3,4531$	$3,9531$	$1,5156$
$n = 250$, allgemeine Matrix	$39,8281$	$20,8125$	$19,6406$
$n = 500$, allgemeine Matrix	$190,2188$	$50,2188$	$85,6094$

Bei der 2-Norm sehen wir hier, dass die direkte Berechnung mehr als doppelt so lang braucht wie die Singulärwertzerlegung. Dies liegt daran, dass Numpy die 2-Norm berechnet, indem der größte Singulärwert berechnet wird. Somit berechnen wir bei der direkten Berechnung die Singulärwertzerlegung sowohl von A als auch von A^{-1}, obwohl wir nur die Singulärwertzerlegung von A benötigen.

Tabelle 7: Vergleich der Laufzeiten unterschiedlicher Verfahren zur Berechnung der Konditionszahl zur 1-Norm

	Direkt (1-Norm)	Hager
$n = 100$, allgemeine Matrix	$0,4688$	$0,2813$
$n = 500$, allgemeine Matrix	14.0938	$2,3906$
$n = 1000$, allgemeine Matrix	$102,5469$	$9,4531$
$n = 100$, untere Dreiecksmatrix	$0,0469$	$0,2031$
$n = 500$, untere Dreiecksmatrix	$4,625$	$1,2031$
$n = 1000$, untere Dreiecksmatrix	$29,8906$	$4,4844$
$n = 100$, Bidiagonalmatrix	$0,1094$	$0,1406$
$n = 500$, Bidiagonalmatrix	$0,5313$	$1,3125$
$n = 1000$, Bidiagonalmatrix	$1,7031$	$5,0156$
$n = 100$, Tridiagonalmatrix	$0,4531$	$0,4063$
$n = 500$, Tridiagonalmatrix	$11,0781$	$2,2031$
$n = 1000$, Tridiagonalmatrix	$62,6563$	$10,9688$

Bei der 1-Norm beobachten wir also, dass bei Bidiagonalmatrizen die direkte Berechnung schneller läuft als Hagers Algorithmus und bei den anderen Matrizen jeweils Hagers Algorithmus der schnellste ist. Weiter sehen wir zusammen mit Tabelle 5, dass, wenn wir die LR-Zerlegung noch nicht kennen, sie den größten Anteil an der Laufzeit hat. Die Singulärwertzerlegung benötigt außerdem die meiste Zeit, entsprechend geht es schneller, die Konditionszahl zur 1-Norm anstatt zur 2-Norm zu berechnen.
Ein weiterer Vorteil von Hagers Algorithmus ist, dass wir bis auf den für die LR-Zerlegung benötigten Speicher keinen weiteren Speicher für weitere Matrizen benötigen, während wir bei der direkten Berechnung zusätzlichen Speicher für die Inversen benötigen. Insbesondere für dünnbesetzte Matrizen ist dies vom Vorteil, da

die Inverse einer dünnbesetzten Matrix in der Regel nicht dünnbesetzt ist.
Zum Schluss vergleichen wir noch die Laufzeit von Hagers Algorithmus und der direkten Berechnung bei dünnbesetzten (sparse) Matrizen, welche im compressed sparse column (CSC) Format gespeichert sind, und bei vollbesetzten Matrizen, die durch die Umwandlung der dünnbesetzten Matrizen in zweidimensionale Numpy Arrays entstehen. Hierzu testen wir 8 unterschiedliche Matrizen, wobei die Matrizen mat_galerkin_tria hierbei einem einfachen Laplace-Problem auf Verfeinerungen eines einfachen Dreiecksgitters auf dem Einheitsquadrat entstammen und die Matrizen mat_supg_quad einem Konvektions-Diffusions-Problem mit einer SUPG-Stabilisierung auf Verfeinerungen des Einheitsquadrats entstammen. Wir führen die Algorithmen 20 mal auf diesen Matrizen aus und nehmen wieder die Summe der Einzellaufzeiten. Die LR-Zerlegung wird hierbei auch jedes Mal einmal ausgeführt und fließt mit in die Ergebnisse ein. Der Code für die Berechnung von Hagers 1-Norm Abschätzung für dünnbesetzte Matrizen sieht hierbei wie folgt aus:

```
import numpy as np
import scipy as sc
import scipy.linalg
import scipy.sparse as sp
import scipy.sparse.linalg

def hager1normCondSparse(A, B, x):
    """
    :param A: csc-matrix A
    :param B: LU-decomposition (result of splu(A))
    :param x: starting vector
    :return: condition number estimate for norm 1
    """
    while True:
        y = B.solve(x)
        xi = 2 * (y >= 0) - 1
        z = B.solve(xi, 'T')
        j = np.argmax(z)
        if abs(z[j]) <= z.T @ x:
            n1 = np.linalg.norm(y, 1)
            return n1 * sp.linalg.norm(A, 1)
        x = np.zeros(x.size)
        x[j] = 1
```

Als Ergebnis erhalten wir folgendes:

Tabelle 8: Vergleich der Laufzeiten bei dünnbesetzten Matrizen

	Direkt sparse	Direkt vollbesetzt	Hager sparse	Hager vollbesetzt
mat_galerkin_tria4 (n=289)	0,9688	1,1094	0,0	0,9688
mat_galerkin_tria5 (n=1089)	6,2968	9,4899	0,0156	4,2969
mat_galerkin_tria6 (n=4225)	72,4687	292,0156	0,0469	49,4531
mat_galerkin_tria7 (n=16641)	1418,7813		1,3125	2336,75
mat_supg_quad4 (n=289)	0,9844	0,8281	0,0	0,75
mat_supg_quad5 (n=1089)	6,2031	6,1563	0,0	2,125
mat_supg_quad6 (n=4225)	73,7657	293,5156	0,0469	51,0313
mat_supg_quad7 (n=16641)	1463,7354		0,7232	2150,7806

Tabelle 9: Vergleich der berechneten Konditionszahlen

	Direkt sparse	Hager sparse
mat_galerkin_tria4 (n=289)	320728,7332	320728,7332
mat_galerkin_tria5 (n=1089)	1224985,6837	1224985,6837
mat_galerkin_tria6 (n=4225)	4782665,1779	4782665,1779
mat_galerkin_tria7 (n=16641)	18894969,3515	18894969,3515
mat_supg_quad4 (n=289)	494,9738	489,0793
mat_supg_quad5 (n=1089)	2245,211	2244,3493
mat_supg_quad6 (n=4225)	9483,5859	9483,5859
mat_supg_quad7 (n=16641)	33970,8329	33970,8329

Tabelle 10: Vergleich der Laufzeiten der LR Zerlegung

	LR Zerlegung sparse	LR Zerlegung vollbesetzt
mat_galerkin_tria4 (n=289)	0,0156	0,0
mat_galerkin_tria5 (n=1089)	0,0938	0,0156
mat_galerkin_tria6 (n=4225)	0,2969	2,2813
mat_galerkin_tria7 (n=16641)	5,875	110,1875
mat_supg_quad4 (n=289)	0,0156	0,0
mat_supg_quad5 (n=1089)	0,0625	0,0156
mat_supg_quad6 (n=4225)	0,3906	2
mat_supg_quad7 (n=16641)	6,8906	108,7656

Wir sehen im dünnbesetzten Fall liefert uns Hagers Algorithmus bei noch sehr großen Matrizen ein schnelles Ergebnis. Weiterhin wurden für die Berechnung von mat_galerkin_tria7 und mat_supg_quad7 im vollbesetzten Fall bei Hagers Algorithmus bis zu 16 Gigabyte Speicher auf meinem Computer benötigt, während es im dünnbesetzten Fall nur bis zu 170 Megabyte waren. Bei der direkten Berechnung über die Inverse waren es im dünnbesetzten Fall bis zu 15 Gigabyte.

Wir können nun also sagen, dass wir bei der 1-Norm im Falle einer Bidiagonalmatrix am besten die direkte Berechnung nach Kapitel 2.1 verwenden und sonst am besten insbesondere für dünnbesetzte Matrizen Hagers Algorithmus verwenden, wobei man sich bewusst sein sollte, dass dieser lediglich eine untere Schranke gibt über deren Qualität wir Abseits von statistischen Auswertungen mit Testmatrizen keinerlei Aussage treffen können.
Wenn uns egal ist, in welcher Norm wir die Konditionszahl berechnen wollen, nutzen wir am besten die 1-Norm, insbesondere wenn es um das Lösen von Gleichungssystemen geht und wir schon eine Zerlegung wie die LR-Zerlegung kennen. Die 2-Norm nehmen wir am besten wenn die Singulärwertzerlegung bereits bekannt ist oder wir die Singulärwertzerlegung einfacher aus den bekannten Ergebnissen berechnen können.

Literatur

[1] M. Bollhofer und V. Mehrmann. *Numerische Mathematik*. Springer, 2004.

[2] R. Kornhuber und C. Schütte. *Computerorientierte Mathematik*. Skript Inst.f.Mathematik, Freie Universität Berlin, 2021.

[3] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, Soc. for Industrial and Applied Mathematics, 2002.

[4] P. Deuflhard und A. Hohmann. *Numerische Mathematik 1*. De Gruyter, 2019.

[5] D. Scholz. *Numerik interaktiv: Grundlagen verstehen, Modelle erforschen und Verfahren anwenden mit taramath*. Springer Berlin Heidelberg, 2016.

[6] W. W. Hager. "Condition Estimates". In: *SIAM Journal on Scientific and Statistical Computing* 5.2 (1984), S. 311–316.

[7] D. Jungnickel. *Optimierungsmethoden: Eine Einführung*. Springer Berlin Heidelberg, 2015.

[8] N. J. Higham. "FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation". In: *ACM Trans. Math. Softw.* 14.4 (Dez. 1988), S. 381–396.