

Matrixfunktionen

Analyse und Implementierung

Masterarbeit im Departement Mathematik
Unter der Betreuung von Herrn Prof. Dr. Daniel Kressner und Herrn Dr. Martin Mächler
Michael Stadelmann - Februar 2009 - ETH Zürich

Inhaltsverzeichnis

1	Einführung	4
2	Grundlagen	5
2.1	Definition von $f(A)$	5
2.2	Polynomiale Auswertungen	8
2.2.1	Horner's Methode	8
2.2.2	Explizite Potenzierung	9
2.2.3	Paterson & Stockmeyer Methode	9
2.2.4	Rundungsfehler	10
2.3	Die Padé-Approximation	11
3	Matrixexponential	14
3.1	Einführung	14
3.2	Konstruktion der „Scaling & Squaring“-Methode	15
3.3	Der Algorithmus	19
3.4	Vorverarbeitung und Ward's Algorithmus	20
3.4.1	Vorverarbeitung	20
3.4.2	Norm-minimierende Translation	20
3.4.3	Ausgleichen	21
3.4.4	Ward's Algorithmus	22
4	Exponentialkonditionszahl	23
4.1	Die Fréchet-Ableitung	23
4.2	Numerische Annäherung	25
4.2.1	Die Fréchet-Ableitung im numerischen Kontext	25
4.2.2	Herleitung des Algorithmus für die Fréchet-Ableitung	27
4.3	Konditionszahl	33
4.3.1	Exakte Berechnung	33
4.3.2	Schätzung der Frobenius-Norm	34
4.3.3	Schätzung der 1-Norm	35
5	Matrixlogarithmus	38
5.1	Einführung	38
5.2	Exkurs: Die Wurzel einer Matrix	39
5.3	Konstruktion der „Inverse Scaling & Squaring“-Methode	41
6	Schlusswort & Ausblick	44
A	Tests implementierter Algorithmen	45
A.1	Tests mit der Exponentialfunktion	45
A.1.1	Einzelne Beispiele	45
A.1.2	Laufzeit mit randomisierten Matrizen	46
A.1.3	Matrizen vom Matrix Market	48
A.1.4	Hessenbergmatrix	51

A.1.5	Balancing	52
A.2	Tests mit der Logarithmusfunktion	53
B	R-Implementierungen	55
B.1	Matrixexponential	55
B.1.1	Exponentialfunktion mit Ausgleichen	55
B.1.2	Fréchet-Ableitung und Exponentialfunktion	56
B.2	Exponentialkonditionszahl	58
B.2.1	Exakte Exponentialkonditionszahl	58
B.2.2	Schätzung der Exponentialkonditionszahl (1-Norm)	58
B.2.3	Schätzung der Exponentialkonditionszahl (Frobenius-Norm)	59
B.3	Matrixlogarithmus	60
B.3.1	Wurzelfunktion	60
B.3.2	Wurzelfunktion für die Logarithmusfunktion	61
B.3.3	Logarithmusfunktion	62

1 Einführung

Matrixfunktionen vereinen verschiedene Gebiete der Mathematik, wie Lineare Algebra, Analysis, Approximationstheorie und Numerik. Vor allem ist nicht nur der theoretische Betrachtungspunkt sehr interessant, sondern auch die numerische Umsetzung.

Der Grundstein der Matrixfunktionen legte Cayley mit der Analyse der Wurzelfunktion in seiner Arbeit „A Memoir on the Theory of Matrices“ (1858). Danach dauerte es nicht lange bis $f(A)$ für beliebige f vorgeschlagen und dessen weiten Nutzen erkannt wurde. Eine neue Bedeutung erhielten Matrixfunktionen im 20. Jahrhundert, wo Computer das Feld der numerischen Mathematik eröffneten. Verschiedene Artikel erschienen zu dieser Thematik, jedoch wurde mit Higham's „Functions of Matrices - Theory and Computation“ (2008) [12] erstmals ein Werk geschaffen, das eine aktuelle und zusammenfassende Übersicht gibt. Die folgenden Ausführungen halten sich mehrheitlich an dieses Buch, welches wir für interessierte Leser nur empfehlen können.

Durch den grossen Erfolg von lizenzfreien wissenschaftlichen Programmen wie R oder Octave hat sich das Interesse für Matrixfunktionen noch ausgeweitet. Denn leistungsstarke Computer und der freie Zugang zu solchen Programmen ermöglicht es immer mehr Individuen, komplexe Algorithmen im Privaten oder in der Wirtschaft zu verwenden, ohne teure Lizenzen und Supercomputer zu erwerben. Diese Nachfrage hat uns veranlasst, die im Rahmen dieser Arbeit behandelten Algorithmen in R zu implementieren und zur Verfügung zu stellen.

Die Arbeit wird wie folgt ablaufen:

Zuerst setzen wir uns mit Matrixfunktionen im Allgemeinen auseinander und erarbeiten uns einige Grundlagen. Danach legen wir den Fokus auf die Exponentialfunktion, wobei wir zuerst die mathematischen Überlegungen erarbeiten und dann zwei Algorithmen untersuchen und miteinander vergleichen. Zum Abschluss der Exponentialfunktion beschäftigen wir uns noch mit der Exponentialkonditionszahl, welche wir ebenfalls durch einen Algorithmus schätzen möchten. Schliesslich wechseln wir zur Logarithmusfunktion über, wobei wir die Analysestruktur mehr oder weniger übernehmen. Die zugehörigen numerischen Tests der implementierten Algorithmen als auch die R-Implementierungen findet man im Appendix.

2 Grundlagen

Unter dem Begriff „Matrixfunktionen“ kann man verschiedene Klassen von Funktionen verstehen. Hier einige Beispiele:

- Skalare Funktionen auf die einzelnen Komponenten der Matrix anwenden, also zum Beispiel $\exp(A) := (\exp a_{ij})$.
- Funktionen von Matrizen, die skalare Ergebnisse ergeben, wie zum Beispiel die Determinantenfunktion.
- Funktionen von Skalaren, die Matrizen als Ergebnisse liefern, wie zum Beispiel $f(t) = tA$ mit $A \in \mathbb{C}^{n \times n}$ und $t \in \mathbb{C}$.

Diese Arbeit handelt von einer weiteren Klasse, bei welcher man eine skalare Funktion f verwendet und eine Matrix $A \in \mathbb{C}^{n \times n}$ einsetzt und $f(A) \in \mathbb{C}^{n \times n}$ berechnet. In diesem Sinne ist dies eine Verallgemeinerung einer skalaren Funktion $f(z)$, $z \in \mathbb{C}$.

Es ist offensichtlich, dass wenn $f(t)$ eine rationale Funktion ist, dass man $f(A)$ durch Substituieren von t durch A erhält. Hierbei wird die Division durch eine Matrixinversion (sofern die Matrix invertiert werden kann) und die 1 durch die Einheitsmatrix I ersetzt.

$$f(t) = \frac{1+t^2}{1-t} \quad \Rightarrow \quad f(A) = (I-A)^{-1}(I+A^2) \quad \text{wenn } 1 \notin \Lambda(A).$$

Hier bezeichne $\Lambda(A)$ das Spektrum von A . Man bemerke, dass rationale Funktionen einer Matrix kommutieren und daher können wir auch $f(A) = (I+A^2)(I-A)^{-1}$ schreiben. Wenn f eine Darstellung als konvergente Potenzreihe hat, so können wir wieder durch substituieren $f(A)$ berechnen:

$$\begin{aligned} \log(1+t) &= t - \frac{t^2}{2} + \frac{t^3}{3} - \frac{t^4}{4} + \dots, & |t| < 1 \\ \log(I+A) &= t - \frac{A^2}{2} + \frac{A^3}{3} - \frac{A^4}{4} + \dots, & \rho(A) < 1 \end{aligned}$$

Hier bezeichne ρ den Spektralradius.

Als Nächstes möchten wir diese Erkenntnisse formalisieren und Matrixfunktionen für ein beliebiges f definieren.

2.1 Definition von $f(A)$

Es gibt verschiedene, im Wesentlichen äquivalente Definitionen von $f(A)$, aber wir werden uns auf eine beschränken und weitere nur am Rande erwähnen. Für unsere Definition benötigen wir die Jordansche Normalform, die ein Standardresultat der Linearen Algebra ist.

Satz 2.1

Sei $A \in \mathbb{C}^{n \times n}$, dann ist die Jordansche Normalform von A die Matrix J , sodass:

$$Z^{-1}AZ = J = \text{diag}(J_1, J_2, \dots, J_p)$$

Dabei sind J_k quadratische Matrixblöcke der Form:

$$J_k = \begin{bmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k \times m_k}$$

Hier sind $\lambda_1, \dots, \lambda_s$ die Eigenwerte von A und es gilt $m_1 + m_2 + \dots + m_p = n$. Die Jordansche Normalform J ist bis auf die Reihenfolge der Blöcke J_k eindeutig, jedoch ist die invertierbare Matrix Z nicht eindeutig. Weiter sei n_i die Grösse des grössten Blockes, in welchem λ_i erscheint.

Um f auf A ohne Probleme anwenden zu können, benötigen wir noch folgende Definition.

Definition 2.2

Die Funktion f ist auf dem Spektrum von A definiert, wenn die folgenden Werte existieren:

$$f^{(j)}(\lambda_i), \quad j = 0 : n_i - 1, \quad i = 1 : s$$

WARNUNG: Es ist sehr wichtig, dass allenfalls auch die Ableitungen von f auf dem Spektrum definiert sind.

Mit diesen Grundlagen können wir nun $f(A)$ ganz formal definieren.

Definition 2.3 (Matrixfunktion via Jordansche Normalform)

Sei f auf dem Spektrum von $A \in \mathbb{C}^{n \times n}$ definiert und sei die Jordansche Normalform J von A wie in Satz 2.1 gegeben. Dann ist

$$f(A) := Zf(J)Z^{-1} = Z \text{diag}(f(J_k))Z^{-1}$$

wobei

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \dots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}$$

Hat die skalare Funktion f mehrere Zweige (Bsp. $\log, \sqrt{\cdot}$), so nehmen wir konsequent denselben Zweig für alle Jordanblöcke.

Um uns dies genauer aufzuzeigen, betrachte man folgendes Beispiel:

$$A = \begin{pmatrix} -49 & 24 \\ -64 & 31 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & -17 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}^{-1}$$

Somit erhalten wir:

$$\exp(A) = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} e^{-1} & 0 \\ 0 & e^{-17} \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}^{-1} \simeq \begin{pmatrix} 0.735759 & 0.551819 \\ -1.471518 & 1.103638 \end{pmatrix}$$

Als nächstes möchten wir noch erläutern, wie man die Definition 2.3 durch die Taylorreihe-Darstellung erhält, welche wir in der Einführung erwähnt haben. Wir schreiben $J_k = \lambda_k I + N_k \in \mathbb{C}^{m_k \times m_k}$. Hier ist N_k überall Null, ausser einer Superdiagonalen mit Einsen. Durch Potenzieren von N_k wandert die Superdiagonale diagonal in die rechte obere Ecke bis sie verschwindet und es gilt $N_k^{m_k} = 0$ - die Matrix ist also nilpotent. Man betrachte das folgende Beispiel mit $m_k = 3$:

$$N_k = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} N_k^2 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} N_k^3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Nun nehmen wir an, dass f eine Darstellung als konvergente Taylorreihe hat:

$$f(t) = f(\lambda_k) + f'(\lambda_k)(t - \lambda_k) + \dots + \frac{f^{(j)}(\lambda_k)(t - \lambda_k)^j}{j!} + \dots$$

Durch Ersetzen von t durch $J_k \in \mathbb{C}^{m_k \times m_k}$ erhalten wir eine endliche Reihe, da die N_k verschwinden:

$$f(J_k) = f(\lambda_k)I + f'(\lambda_k)N_k + \dots + \frac{f^{(j)}(\lambda_k)N_k^{m_k-1}}{(m_k - 1)!}$$

Da die N_k durch solche Superdiagonalen beschrieben werden, kann man die Definition 2.3 direkt ablesen.

Alternative Definitionen gehen über Hermitesche Interpolation oder über das Cauchy Integral, wobei man aber zeigen kann, dass all diese Definitionen im Wesentlichen äquivalent sind. Für diesen Beweis als auch eine etwas ausführlichere Einführung in die Matrixfunktionen verweisen wir auf [12], worauf auch dieses Kapitel basiert.

Matrixfunktionen dienen auch um nichtlineare Matrixgleichungen zu lösen, wie zum Beispiel $g(X) = A$. Im Falle von $X^2 = A$ oder $e^X = A$ müssen wir etwas unterscheiden, was Lösungen der Gleichungen sind und was das Ergebnis der Matrixfunktion ist. Jede Matrix X , die eine solche Gleichung erfüllt, ist offensichtlich eine Lösung. Wenden wir die Matrixfunktion f (wobei $f = g^{-1}$) im Sinne der Definition 2.3 auf A an, so erhalten wir ebenfalls eine Lösung - wir nennen dieses f die Hauptmatrixfunktion. Jedoch gibt es manchmal Lösungsmatrizen X , welche man nicht durch die Hauptmatrixfunktion f erhält, aber welche trotzdem Lösungen der Matrixgleichung sind. Solche X sind Beispiele

von Nichthauptmatrixfunktionen, welches wir am folgenden Beispiel erläutern möchten. Nehmen wir an, wir möchten die Wurzel der Einheitsmatrix I finden:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Die Matrix ist bereits in der Jordansche Normalform und mit $f = \sqrt{t}$ und der Definition 2.3 erhalten wir direkt I und $-I$ als Lösungen. Wenn wir nun für jeden Block beliebige Zweige der Wurzelfunktion verwenden, so erhält man noch weitere Wurzeln von I :

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Wir haben dabei für die beiden Eigenwerte 1 jeweils verschiedene Wurzelzweige berechnet und dabei Resultate der Nichthauptmatrixfunktionen erhalten. In dieser Arbeit werden wir uns jedoch ausschliesslich mit Hauptmatrixfunktionen beschäftigen.

2.2 Polynomiale Auswertungen

Da Approximationen immer auf Polynomen basieren, ist es nützlich kurz zu untersuchen, wie wir Polynome mit einer Matrix als Variable geschickt berechnen können. Da eine Matrixmultiplikation (kurz M genannt) zweier $n \times n$ Matrizen sehr teuer ist ($2n^3$ Flops=Maschinen-Rechenoperationen), muss man etwas mehr aufpassen als bei einer skalaren Polynomauswertung.

Im Folgenden bezeichne X eine reelle oder komplexe $n \times n$ Matrix und p_m ein Polynom der Ordnung m mit folgender Darstellung:

$$p_m(X) = \sum_{k=0}^m b_k X^k \quad b_k \in \mathbb{C}$$

2.2.1 Horner's Methode

Bei dieser Methode beginnt man bei $b_m X + b_{m-1}$, multipliziert jeweils ein X hinzu und addiert den nächsten Koeffizienten, bis man bei b_0 angelangt ist.

$$(((b_m X + b_{m-1})X + b_{m-2})X + b_{m-3})X + \dots$$

Algorithmus 2.4

- 1: $S_{m-1} = b_m X + b_{m-1} I$
- 2: **for** $k = m - 2$ to 0 **do**
- 3: $S_k = X S_{k+1} + b_k I$
- 4: **end for**
- 5: $p_m = S_0$

Die Kosten belaufen sich auf $(m - 1)M$. Diese Methode hat den grossen Nachteil, dass man die Ordnung des Polynoms bereits zu Beginn benötigt. Gerade bei Approximationen mit Potenzreihen ist dies jedoch häufig nicht der Fall.

2.2.2 Explizite Potenzierung

Im Vergleich zur Horner's Methode wird hier gerade umgekehrt vorgegangen. Man beginnt bei der kleinsten Potenz und addiert die nächst Höhere hinzu, welche man jeweils durch eine Matrixmultiplikation erhält.

$$(((b_0I + b_1X) + b_2XX) + b_3X^2X) + \dots)$$

Algorithmus 2.5

- 1: $P = X$
- 2: $S = b_0I + b_1X$
- 3: **for** $k = 2$ to m **do**
- 4: $P = PX$
- 5: $S = S + b_kP$
- 6: **end for**
- 7: $p_m = S$

Auch hier sind die Kosten $(m - 1)M$.

2.2.3 Paterson & Stockmeyer Methode

Hier wird eine spezielle Darstellung des Polynoms p_m ausgenützt:

$$p_m(X) = \sum_{k=0}^r B_k \cdot (X^s)^k \quad r = \lfloor m/s \rfloor$$

wobei s eine natürliche Zahl ist und die B_k wie folgt definiert sind:

$$B_k = \begin{cases} b_{sk+s-1}X^{s-1} + \dots + b_{sk+1}X + b_{sk}I, & k = 0 : r - 1 \\ b_mX^{m-sr} + \dots + b_{sr+1}X + b_{sr}I, & k = r \end{cases}$$

Nachdem man die Potenzen X^2, \dots, X^s berechnet hat, wertet man das Polynom mit Horner's Methode aus. Die beiden Extrembeispiele $s = 1$ resp. $s = m$ führen wieder auf die beiden Algorithmen 2.4 resp. 2.5 zurück. Man betrachte folgendes Beispiel:

$$p_6 = \underbrace{b_6I}_{B_2} (X^3)^2 + \underbrace{(b_5X^2 + b_4X + b_3I)}_{B_1} X^3 + \underbrace{(b_2X^2 + b_1X + b_0I)}_{B_0}$$

Um die X^2 und X^3 zu berechnen, benötigen wir $2M$. Danach werten wir ein Polynom 2. Grades aus, was nochmals ein M kostet, also haben wir total $3M$ benötigt. Im Vergleich benötigen die Algorithmen 2.4 und 2.5 total $5M$.

Allgemein gilt für die Kosten hier:

$$(s + r - 1 - f(s, m))M, \quad f(s, m) = \begin{cases} 1 & \text{wenn } m \text{ durch } s \text{ teilbar ist} \\ 0 & \text{sonst} \end{cases}$$

Mit dieser Methode benötigt man zum Beispiel für p_{16} mit $(s = r = 4)$ lediglich $6M$, wobei die beiden anderen Algorithmen Kosten von $15M$ aufweisen. Die einzigen Nachteile sind, dass man einerseits $(s + 2)n^2$ Elemente speichern muss, andererseits muss auch hier der Grad des Polynoms bereits im Voraus bekannt sein.

2.2.4 Rundungsfehler

Schliesslich möchten wir kurz die Rundungsfehler untersuchen. In diesem Abschnitt sind der Absolutwert und Ungleichungen für Matrizen komponentenweise festgelegt und das Maximum sei das maximale Element der Matrix, also $\max\{a_{ij}\}$. Weiter setzen wir $\tilde{\gamma}_n = cnu/(1 - cnu)$, wobei u die Maschinengenauigkeit bezeichnet und c eine kleine positive ganze Zahl ist.

Satz 2.6

Für das Polynom \hat{p}_m , welches durch eine der vorgestellten drei Methoden berechnet wurde, gilt:

$$|p_m - \hat{p}_m| \leq \tilde{\gamma}_{mn} \tilde{p}_m(|X|)$$

Wobei $\tilde{p}_m(X) = \sum_{k=0}^m |b_k| X^k$. Somit gilt $\max(|p_m - \hat{p}_m|) \leq \tilde{\gamma}_{mn} \tilde{p}_m(\max(|X|))$.

Obwohl die Schranke des Satzes grosszügig abgeschätzt wurde, ist die Aussage klar: Rundungsfehler können im berechneten Resultat relativ grosse Fehler verursachen.

2.3 Die Padé-Approximation

Für eine gegebene Funktion $f(x)$ ist die rationale Funktion $r_{km} = p_{km}(x)/q_{km}(x)$ eine $[k/m]$ Padé-Approximation für f , falls:

- p_{km} ein Polynom k -ten Grades ist
- q_{km} ein Polynom m -ten Grades ist
- $q_{km}(0) = 1$
- $f(x) - r_{km} = O(x^{k+m+1})$

Für gegebene f , k und m kann es sein, dass die Padé-Approximation nicht existiert - aber falls diese existiert, ist diese eindeutig [12, Kap. 4.4.2]. Für einige f wurde die Existenz von r_{km} für alle k und m bewiesen. Für eine detailliertere Einführung in die Padé-Approximation verweisen wir auf Baker's „Essentials of Padé Approximants“ [13]. Die vierte Bedingung zeigt, dass r_{km} die ersten $k + m + 1$ Terme der Taylorentwicklung reproduziert. Es ist auch ersichtlich, dass r_{k0} gerade der abgebrochenen Taylorentwicklung entspricht.

Die Kettenbruchentwicklung und die Padé-Approximation stehen direkt im Zusammenhang:

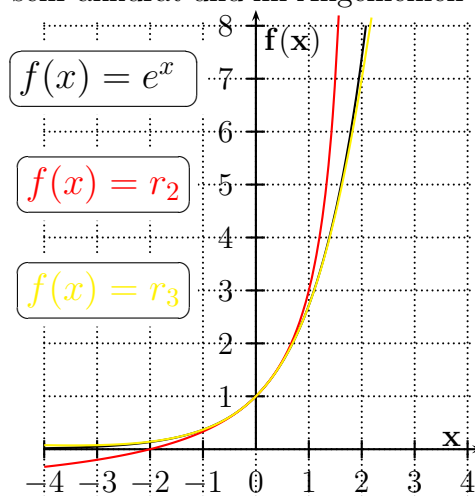
$$f(x) = b_0 + \frac{a_1 x}{b_1 + \frac{a_2 x}{b_2 + \frac{a_3 x}{\dots}}}$$

Durch diese Darstellung erhält man nun leicht die Padé-Approximation. Sind alle $b_0 = b_1 = \dots = 1$ und alle a_i ungleich Null, so sind Kettenbrüche

$$r_m(x) := r_{mm} = 1 + \frac{a_1 x}{1 + \frac{a_2 x}{1 + \frac{a_3 x}{1 + \dots + \frac{a_{2m-2} x}{1 + \frac{a_{2m-1} x}{1 + \frac{a_{2m} x}{1}}}}}$$

die $[0/0], [1/0], [1/1], [2/1] \dots$ Padé-Approximationen von f .

Man wird sich fragen, was der Vorteil dieser Approximation ist im Vergleich zur Taylorentwicklung. Die Padé-Approximation konvergiert schneller als die Taylorreihe, ist daher akkurater und hat tiefere Kosten für die Auswertung. Des Weiteren ist die Padé-Approximation nahe bei 0 sehr akkurat und im Allgemeinen eine gut entwickelte Theorie.



Beispiel

1. Als erstes Beispiel betrachte man die Exponentialfunktion, die wir später noch verwenden werden. Diese wurde bereits sehr gut analysiert und man kennt die Approximation für alle k und m :

$$p_{km} = \sum_{j=0}^k \frac{(k+m-j)!k!}{(k+m)!(k-j)!} \frac{x^j}{j!}, \quad q_{km} = \sum_{j=0}^k \frac{(k+m-j)!m!}{(k+m)!(m-j)!} \frac{(-x)^j}{j!}$$

Man bemerke die Relation $p_{km}(x) = q_{km}(-x)$, welche die Eigenschaft $1/e^x = e^{-x}$ wieder spiegelt. Dass dies die Bedingungen für eine Padé-Approximation erfüllt, ist direkt vom Fehlerterm ablesbar:

$$e^x - r_{km} = (-1)^m \frac{k!m!}{(k+m)!(k+m+1)!} x^{k+m+1} + O(x^{k+m+2})$$

Die Herleitung der letzten beiden Formeln sind in Gautschi's „Numerical Analysis: An Introduction“ [5] zu finden.

Hier noch die Liste der Koeffizienten b_i von $p_m(x) = \sum_{i=0}^m b_i x^i$:

m	b_i mit $i = 0 : m$
3	[120,60,12,1]
5	[30240, 15120, 3360, 420, 30,1]
7	[17297280, 8648640, 1995840, 277200, 25200, 1512, 56, 1]
9	[17643225600, 8821612800, 2075673600, 302702400, 30270240, 2162160, 110880, 3960, 90,1]
13	[64764752532480000,32382376266240000,7771770303897600, 1187353796428800, 129060195264000,10559470521600, 670442572800, 33522128640, 1323241920, 40840800,960960,16380, 182,1]

2. Als weiteres Beispiel betrachte man die Logarithmus-Funktion. Die elegante Darstellung der Exponentialfunktion findet man hier leider nicht wieder.

Trotzdem kennt man die Approximationsfunktion r_m sehr genau, zwar sind die Koeffizienten der beiden Polynome p_m, q_m nicht elegant in einer geschlossenen Form beschreibbar, jedoch gilt dies für die Koeffizienten des Kettenbruchs:

$$a_1 = 1, \quad a_{2j} = \frac{j}{2(2j-1)}, \quad a_{2j+1} = \frac{j}{2(2j+1)}, \quad j = 1, 2, \dots$$

Dieser Kettenbruch kann man in eine Partialbruchreihe der folgenden Form zerlegen:

$$r_m(x) = \sum_{j=1}^m \frac{\alpha_j^{(m)} x}{1 + \beta_j^{(m)} x}$$

wobei $\alpha_j^{(m)}$ die Gewichte und $\beta_j^{(m)}$ die Knoten der m -Punkte Regel auf $[0, 1]$ sind. Diese sind reell, wie die Gauss'sche Quadratur Theorie aufzeigt. Algorithmen, um diese Gewichte und Knoten zu berechnen, findet man in [3],[4], [6] und [16]. In der folgenden Tabelle sind die Koeffizienten, wie wir diese in Matlab berechnet haben und später auch noch verwenden werden:

m	α_j^m für $j = 1 : m$
3	2.4647 , 0.88889 , 0.31306
4	2.505, .98807, 0.48668, 0.1869
5	2.5253 ,1.037 ,0.56889 ,0.31111
6	2.537, 1.0649,0.61456,0.37777,0.21717,0.088656
7	2.5443,1.0822,0.64264,0.41796,0.2716,0.16061,0.066433

m	$1/\beta_j^m$ für $j = 1 : m$
3	8.873, 2, 1.127
4	14.403 ,3.0302 ,1.4926 ,1.0746
5	21.317,4.3334,2,1.3 ,1.0492
6	29.616,5.9034,2.6268,1.6147,1.2039,1.0349
7	39.299,7.7379,3.3661,2,1.4226,1.1484,1.0261

Schliesslich möchten wir auch hier noch die Koeffizienten der Padé-Approximation $r_m = p_m/q_m$ der Vollständigkeit halber angeben:

m	b_i von p_m mit $i = 0 : m$
3	[0, 60, 60, 11]
4	[0, 420, 630, 260, 25]
5	[0, 7560, 15120, 9870, 2310, 137]
6	[0, 9240, 23100, 20720, 7980, 1218, 49]
7	[0, 240240,720720,823900,446600,115668,12488,363]

m	b_i von q_m mit $i = 0 : m$
3	[60, 90, 36, 3]
4	[420, 840, 540, 120, 6]
5	[7560, 18900, 16800, 6300, 900, 30]
6	[9240, 27720, 31500, 16800, 4200, 420, 10]
7	[240240,840840,1164240,808500,294000,52920,3920,70]

3 Matrixexponential

3.1 Einführung

Das Exponential einer Matrix ist wohl eine der meist untersuchten Matrixfunktionen. Viele mathematische Modelle in allen Wissenschaftssparten benötigen lineare gewöhnliche Differentialgleichungssysteme mit konstanten Koeffizienten der folgenden Form:

$$\dot{x}(t) = Ax(t)$$

$$x(0) = x_0$$

Hier stellt A eine reelle oder komplexe $n \times n$ Matrix dar und $x(0)$ setzt die Anfangsbedingungen. Die Lösung dieses Gleichungssystems wird durch $x(t) = e^{tA}x_0$ gegeben, wobei e^{tA} formal durch die Taylorentwicklung definiert wird:

$$e^{tA} = \sum_{i=0}^{\infty} \frac{(tA)^i}{i!}$$

Leider gelten dabei nicht alle Gesetze der skalaren Funktion:

Satz 3.1

Sei $A, B \in \mathbb{C}^{n \times n}$. Dann ist $e^{(A+B)t} = e^{At}e^{Bt}$ für alle t genau dann, wenn $AB = BA$ ist.

Beweis

Wenn $AB = BA$ ist, so kommutieren alle Terme in der Potenzreihe und das Gesetz gilt wegen den skalaren Eigenschaften der Exponentialfunktion. Gilt $e^{(A+B)t} = e^{At}e^{Bt}$, so können wir die Koeffizienten von t^2 vergleichen und wir sehen, dass $(AB + BA)/2 = AB$ ist, sodass die Kommutativität direkt folgt. \square

Es ist offensichtlich, dass ein Computer eine unendliche Reihe als Formel nicht verwenden kann. Daher gibt es für die Berechnung von e^A verschiedene Methoden, sowohl abhängig von der Form der Matrix und als auch ob man lediglich e^A , $e^A v$ für einen fixen Vektor v oder auch e^{tA} für verschiedene Werte von t berechnen möchte. Für eine Übersicht möglicher Methoden verweisen wir den interessierten Leser auf „Nineteen Dubious Ways to Compute the Exponential of a Matrix“ [14].

Im Folgenden möchten wir den Algorithmus „Scaling & Squaring“ analysieren. Dieser wird sehr häufig implementiert, um e^A für eine beliebige Matrix A zu berechnen.

„Scaling & Squaring“ besteht aus drei Schritten:

- Die Norm von $A/2^s =: B$ genügend klein skalieren (*Scaling*)
- e^B approximieren (Padé-Approximation)
- $e^A = (e^B)^{2^s}$ berechnen (*Squaring*)

Die folgenden Ausführungen stützen sich auf Kapitel 4 und 10 in Nicholas J. Higham's Buch „Functions of Matrices“ [12].

3.2 Konstruktion der „Scaling & Squaring“-Methode

Diese Methode nützt einerseits die Gleichung $e^A = (e^{A/\sigma})^\sigma$ aus, wobei $A \in \mathbb{C}^{n \times n}$ ist, andererseits, dass die Padé-Approximation im Ursprung sehr akkurat ist, d.h. also für kleine $\|A\|$. Wie bereits bei der Einführung erklärt ist die Idee nun, ein $\sigma = 2^s$ so zu wählen, dass die Norm von A/σ genügend klein ist. Dann berechnet man $e^{A/2^s} \approx r_{km}(A/2^s)$ und durch Potenzieren erhält man schliesslich $e^A \approx (r_{km}(A/2^s))^{2^s}$. Natürlich müssen k, m und s noch optimal bestimmt werden.

Glücklicherweise sind nur diagonale Padé-Approximationen ($k = m$) interessant, denn wenn $k \neq m$ ist, so ist r_{km} weniger akkurat als r_{jj} für $j = \max\{k, m\}$, aber die Anzahl nötiger Matrixmultiplikationen ist gleich. Es bezeichne von nun an

$$r_m := r_{mm} = p_{mm}/q_{mm}$$

Nun möchten wir uns mit der Wahl des optimalen s , dem Skalierungsfaktor, als auch mit dem Grade m der Padé-Approximation beschäftigen. Genau gesagt ist es unser Ziel, m und s so zu wählen, dass der Rückwärtsfehler des berechnete e^A durch die Maschinengenauigkeit beschränkt ist und die Berechnungskosten minimal sind. Bei der Rückwärtsfehleranalyse nehmen wir exakte Arithmetik an und untersuchen lediglich den Approximationsfehler durch die Padé-Approximation.

Im Algorithmus möchten wir s in Abhängigkeit von $\|A\|$ bestimmen, wobei die Norm eine beliebige konsistente Matrix Norm ist. Daher ist es sinnvoll den Rückwärtsfehler in Abhängigkeit von $\|2^{-s}A\|$ zu beschränken und dann, für jeden Grad m , das maximale $\|2^{-s}A\|$ bestimmen, für welches r_m den gewünschten Rückwärtsfehler liefert. Sei nun für $s \in \mathbb{N}$:

$$I \approx e^{-2^{-s}A} r_m(A/2^s) = I + G = e^H$$

wobei wir annehmen dass $\|G\| < 1$ ist, sodass $H = \log(I + G)$ sicher existiert. Mit der Taylorreihe des natürlichen Logarithmus folgt direkt die Ungleichung $\|H\| \leq -\log(1 - \|G\|)$:

$$\begin{aligned} \|H\| &= \|\log(I + G)\| = \left\| \sum_{n=1}^{\infty} (-1)^{n+1} \frac{G^n}{n} \right\| \\ &\leq \sum_{n=1}^{\infty} \frac{\|G^n\|}{n} = -\log(1 - \|G\|) \end{aligned}$$

Da Matrizen im Allgemeinen nicht kommutieren, gilt das Exponentialgesetz

$$e^A e^B = e^{A+B}$$

nicht. Aber da G offensichtlich eine Funktion in Abhängigkeit von A ist, gilt dies ebenfalls für H und deshalb kommutieren A und H . Es folgt:

$$r_m(A/2^s) = e^{2^{-s}A} e^H = e^{2^{-s}A+H} \text{ resp. } r_m(A/2^s)^{2^s} = e^{A+2^s H}$$

Wir setzen $E := 2^s H$ und schreiben $e^E = e^{2^s H} = (I + G)^{2^s}$, sodass mit der oben genannten Ungleichung

$$\|E\| \leq -2^s \log(1 - \|G\|)$$

folgt. Wir fassen unsere Ergebnisse nun im folgenden Theorem zusammen:

Satz 3.2

Sei r_m die diagonale Padé Approximation und sei die folgende Gleichung für ein G mit $\|G\| < 1$ für eine beliebige konsistente Matrixnorm erfüllt:

$$e^{-2^{-s}A} r_m(2^{-s}A) = I + G$$

Dann gilt

$$r_m(2^{-s}A)^{2^s} = e^{A+E}$$

wobei E mit A kommutiert. Weiter gilt folgende Abschätzung:

$$\frac{\|E\|}{\|A\|} \leq \frac{-\log(1 - \|G\|)}{\|2^{-s}A\|}$$

Mit dieser Rückwärtsfehler-Analyse interpretieren wir den Abbruchfehler der Padé-Approximation als eine Störung in der ursprünglichen Matrix A . Der Vorteil ist, dass diese Analyse sogar den „Squaring“ Teil, also das Potenzieren mit 2^s des Approximationsfehlers, mit in die Fehlerberechnung einbezieht.

Als Nächstes möchten wir die Norm von G in Abhängigkeit von $\|2^{-s}A\|$ beschränken. Hierzu definieren wir die Funktion

$$\rho(x) := e^{-x} r_m(x) - 1 = \sum_{i=2m+1}^{\infty} c_i x^i$$

wobei die zweite Gleichheit direkt aus den Eigenschaften der Padé-Approximation folgt. Diese Reihe konvergiert absolut für $|x| < \min\{|t| : q_m(t) = 0\} =: \nu_m$. Somit folgt direkt:

$$\|G\| = \|\rho(2^{-s}A)\| \leq \sum_{i=2m+1}^{\infty} |c_i| \theta^i =: f(\theta),$$

wobei $\theta = \|2^{-s}A\| < \nu_m$ ist. Nun kombinieren wir dies mit den Resultaten aus dem Satz 3.2 und erhalten:

$$\frac{\|E\|}{\|A\|} \leq \frac{-\log(1 - f(\theta))}{\theta}$$

Die folgende Tabelle wurde von [12] übernommen. Diese zeigt einerseits die ersten zwei Stellen von θ_m , welches die obere Schranke von $\theta = \|2^{-s}A\|$ ist, sodass die Rückwärtsfehlerschranke nicht die Maschinengenauigkeit $u = 2^{-53} \approx 1.1 \times 10^{-16}$ überschreitet. Andererseits zeigt die zweite Zeile $\nu_m := \min\{|t| : q_m(t) = 0\}$. Zu guter Letzt ist in der dritten Zeile die obere Schranke von $\|q_m(A)^{-1}\|$, worauf wir später zurückkommen. Wie man sieht, wird die Ungleichung $\theta_m < \nu_m$ nochmals bestätigt und weist auf den Fakt hin, dass $q_m(A)$ für $\|A\| \leq \theta_m$ nicht singulär ist.

m	1	2	3	4	5	6	7	8	9	10	
θ_m	3.7e-8	5.3e-4	1.5e-2	8.5e-2	2.5e-1	5.4e-1	9.5e-1	1.5e0	2.1e0	2.8e0	
ν_m	2.0e0	3.5e0	4.6e0	6.0e0	7.3e0	8.7e0	9.9e0	1.1e1	1.3e1	1.4e1	
η_m	1.0e0	1.0e0	1.0e0	1.0e0	1.1e0	1.3e0	1.6e0	2.1e0	3.0e0	4.3e0	
m	11	12	13	14	15	16	17	18	19	20	21
θ_m	3.6e0	4.5e0	5.4e0	6.3e0	7.3e0	8.4e0	9.4e0	1.1e1	1.2e1	1.3e1	1.4e1
ν_m	1.5e1	1.7e1	1.8e1	1.9e1	2.1e1	2.2e1	2.3e1	2.5e1	2.6e1	2.7e1	2.8e1
η_m	6.6e0	1.0e1	1.7e1	3.0e1	5.3e1	9.8e1	1.9e2	3.8e2	8.3e2	2.0e3	6.2e3

Der Vollständigkeit wegen möchten wir noch hinweisen, dass die Werte in der Tabelle mit MATLAB's Symbolic Math Toolbox und der 250 Dezimalstellenarithmetik berechnet wurden, man $f(\theta)$ durch die ersten 150 Terme approximiert und die c_i symbolisch herleitet.

Als Nächstes müssen wir uns noch überlegen, wieviel es kostet $r_m(A)$ zu evaluieren. Wegen der Relation $q_m(x) = p_m(-x)$ ist es ein effizienter Weg zuerst alle geraden Potenzen von A zu berechnen, dann p_m und q_m zu bilden und schliesslich $q_m r_m = p_m$ zu lösen. Sei nun $p_m(x) = \sum_{i=0}^m b_i x^i$.

- Für gerade Grade $2m$ setzen wir:

$$p_{2m}(A) = b_{2m}A^{2m} + \dots + b_2A^2 + b_0I + A(b_{2m-1}A^{2m-2} + \dots + b_3A^2 + b_1I) =: U + V.$$

Dies kostet uns $(m+1)M$, indem wir A^2, A^4, \dots, A^{2m} berechnen und einsetzen. Wir erhalten direkt und ohne zusätzliche Kosten auch $q_{2m}(A) = U - V$.

- Für ungerade Grade $2m+1$ setzen wir:

$$p_{2m+1}(A) = A(b_{2m+1}A^{2m} + \dots + b_3A^2 + b_1I) + b_{2m}A^{2m} + \dots + b_2A^2 + b_0I =: U + V.$$

Somit kann man p_{2m+1} und q_{2m+1} für die gleichen Kosten berechnen wie p_{2m} und q_{2m} .

Für $m \geq 12$ kann man dieses Schema sogar noch verbessern. Als Beispiel betrachte man

$$\begin{aligned} p_{12}(A) &= A^6(b_{12}A^6 + b_{10}A^4 + b_8A^2 + b_6I) + b_4A^4 + b_2A^2 + b_0I \\ &+ A[A^6(b_{11}A^4 + b_9A^2 + b_7I) + b_5A^4 + b_3A^2 + b_1I] =: U + V \end{aligned}$$

respektive $q_{12} = U - V$. Damit kann man p_{12} und q_{12} mit nur $6M$ berechnen, da man für A^2, A^4, A^6 drei Matrixmultiplikationen benötigt und nochmals drei für die Evaluierung. Es ist nun offensichtlich, dass für $m = 13$ die analoge Formel gilt. Man betrachte die folgende Tabelle, wobei π_m die Anzahl Matrixmultiplikationen bezeichnet um p_m und q_m zu berechnen.

m	1	2	3	4	5	6	7	8	9	10
π_m	0	1	2	3	3	4	4	5	5	6
C_m	25	12	8.1	6.6	5.0	4.9	4.1	4.4	3.9	4.5

m	11	12	13	14	15	16	17	18	19	20	21
π_m	6	6	6	7	7	7	7	8	8	8	8
C_m	4.2	3.8	3.6	4.3	4.1	3.9	3.8	4.6	4.5	4.3	4.2

Aus diesen beiden Tabellen können wir nun einige Informationen lesen. Ist $\|A\| \geq \theta_{21}$, so müssen wir A mit 2^{-s} skalieren - die Frage ist nun, wie wir s und m bestimmen. Zuerst überlegen wir uns, dass nach der zweiten Tabelle das m in der Menge $\mathbb{M} = \{1, 2, 3, 5, 7, 9, 13, 17, 21\}$ liegen sollte. Denn es macht zum Beispiel keinen Sinn $m = 10$ zu verwenden, denn wir können für dieselben Kosten das akkuratere r_{12} berechnen.

In unserer Auswahlliste \mathbb{M} brauchen wir jeweils eine Matrixmultiplikation mehr, wenn wir ein Element höher gehen. Dies ermöglicht uns jedoch auch ein grösseres $\theta_m = \|2^{-s}A\|$. Ist nun θ_m um den Faktor 2 grösser, so reduziert dies s bei 1, sodass wir eine Matrixmultiplikation sparen im „Squaring“-Schritt. Man kann nun anhand der Tabelle ebenfalls feststellen, dass dieser Faktor 2 immer erfüllt ist bis $m = 13$, was uns schliessen lässt, dass dies die beste Lösung ist. Um dies genauer zu analysieren betrachte wir die Gesamtkosten des Algorithmus. Da $s = \lceil \log_2(\|A\|/\theta_m) \rceil$ ist, wenn $\|A\| > \theta_m$ bzw. $s = 0$ sonst, gilt:

$$\pi_m + s = \pi_m + \max(\lceil \log_2 \|A\| - \log_2 \theta_m \rceil, 0).$$

Man bemerke, dass wir die zu lösende Matrixgleichung vernachlässigen, da diese für alle m gleich ist. Nun möchten wir m so wählen, dass die Gesamtkosten minimal sind. Für $\|A\| \geq \theta_m$ können wir das \max und den $\|A\|$ Term vernachlässigen, da dies lediglich eine Verschiebung um eine Konstante ist. Daher müssen wir den folgenden Ausdruck minimieren:

$$C_m = \pi_m - \log_2 \theta_m$$

Diese Werte sind in der vorangegangenen Tabelle in der zweiten Reihe eingetragen und wir stellen wiederum fest, dass $m = 13$ die optimale Wahl ist.

Als nächstes möchten wir noch kurz die Rundungsfehler bei der der Evaluation von $r_m(A)$ untersuchen. Hier können wir $m = 1$ und $m = 2$ sofort als mögliche Kandidaten streichen, denn bei r_1 und r_2 können starke Auslöschungen vorkommen. Nun wenden wir das Theorem 2.6 an, wobei wir $\|A\|_1 \leq \theta_m$ annehmen und vermerken, dass p_m nur positive Koeffizienten hat:

$$\begin{aligned} \|p_m(A) - \hat{p}_m(A)\|_1 &\leq \tilde{\gamma}_{mn} p_m(\|A\|_1) \\ &\approx \tilde{\gamma}_{mn} e^{\|A\|_1/2} \leq \tilde{\gamma}_{mn} \|e^{A/2}\|_1 e^{\|A\|_1} \\ &\approx \tilde{\gamma}_{mn} \|p_m(A)\|_1 e^{\|A\|_1} \leq \tilde{\gamma}_{mn} \|p_m(A)\|_1 e^{\theta_m} \end{aligned}$$

Der relative Fehler ist also durch $\tilde{\gamma}_{mn} e^{\theta_m}$ beschränkt, was man angesichts den Werten von θ_m als genügend gut bezeichnen kann.

Analog erhalten wir durch ersetzen von A durch $-A$ die folgende Abschätzung:

$$\|q_m(A) - \hat{q}_m(A)\|_1 \lesssim \tilde{\gamma}_{mn} \|q_m(A)\|_1 e^{\theta_m}$$

Zusammengefasst sind die Fehler von p_m und q_m gut beschränkt.

Um r_m zu erhalten, müssen wir ein lineares Gleichungssystem lösen, wobei $q_m(A)$ als Koeffizientenmatrix zu verstehen ist. Um sicherzustellen, dass wir das System auch akkurat lösen können, müssen wir noch prüfen ob $q_m(A)$ gut konditioniert ist. Daher möchten wir eine Schranke für $\|q_m(A)^{-1}\|$ herleiten, wobei unser Vorgehen analog ist, wie bei der Herleitung der θ_m . Wir vermerken kurz, dass $q_m(x) \approx e^{x/2}$ ist und schreiben für $\|A\| \leq \theta_m$:

$$q_m(A) = e^{-A/2}(I + e^{A/2}q_m(A) - I) =: e^{-A/2}(I + F).$$

Für $\|F\| < 1$ haben wir somit

$$\|q_m(A)^{-1}\| \leq \|e^{A/2}\| \|(I + F)^{-1}\| \leq \frac{e^{\theta_m/2}}{1 - \|F\|}.$$

Mit der Reihenentwicklung schreiben wir $e^{x/2}q_m(x) - 1 = \sum_{i=2}^{\infty} d_i x^i$, sodass $\|F\| \leq \sum_{i=2}^{\infty} |d_i| \theta_m^i$ folgt. Abschliessend erhalten wir die Schranke

$$\|q_m(A)^{-1}\| \leq \frac{e^{\theta_m/2}}{1 - \sum_{i=2}^{\infty} |d_i| \theta_m^i}.$$

Die η_m wurden genau wie die θ_m berechnet (MATLAB's Symbolic Toolbox) und sind in der Tabelle auf Seite 17 zu finden. Wir stellen fest, dass q_m für $m \leq 13$ sehr gut konditioniert ist für $\|A\| \leq \theta_m$.

3.3 Der Algorithmus

Nun möchten wir unsere Erkenntnisse aus den vorangegangenen Kapiteln zusammentragen und in einen Algorithmus packen. Dieser verläuft wie folgt:

- Prüfe, ob $\|A\| \leq \theta_m$ für $m \in \{3, 5, 7, 9, 13\}$
- Falls dies gilt, so evaluiere r_m für das kleinste m
- Ansonsten benütze die „Scaling & Squaring“-Methode mit $m = 13$

Algorithmus 3.3

- 1: **for** m in $[3, 5, 7, 9]$ **do**
- 2: **if** $\|A\|_1 \leq \theta_m$ **then**
- 3: Berechne U and V und löse $(-U + V)X = U + V$
- 4: Quit
- 5: **end if**
- 6: **end for**
- 7: $A \leftarrow A/2^s$ mit $s \in \mathbb{N}$ sodass $\|A/2^s\|_1 \leq \theta_{13}$
- 8: Berechne r_{13}
- 9: Erhalte $X = r_{13}^{2^s}$ durch wiederholtes potenzieren.

Die Kosten belaufen sich wie teilweise bereits evaluiert auf $(\pi_m + \lceil \log_2(\|A\|_1/\theta_m) \rceil)M + D$ wobei D für eine Matrixdivision steht, welche man durch die Lösung der Gleichung $AX = B$ benötigt.

3.4 Vorverarbeitung und Ward's Algorithmus

3.4.1 Vorverarbeitung

Man hat die Möglichkeit, die Matrix bereits vor dem Algorithmus zu bearbeiten um die Norm zu verkleinern. Im Folgenden möchten wir kurz zwei mögliche Vorverarbeitungen vorstellen, die Robert C. Ward 1977 in „Numerical Computation of the Matrix Exponential with Accuracy Estimate“ [18] vorschlägt. Jedoch relativiert die kritische Haltung von Higham [12] die erhoffte Effizienzsteigerung, wie man in den beiden folgenden Abschnitten entnehmen kann.

3.4.2 Norm-minimierende Translation

Diese erste Technik möchte die Norm der Matrix ohne grosse Kosten verkleinern. Hierzu wird eine Translation der Matrix A um ein Vielfaches μ der Einheitsmatrix I gesucht, sodass $\|A - \mu I\|$ minimal wird. Dieses Optimierungsproblem kann (je nach Norm) sehr kompliziert sein. Daher wählen wir eine Norm, bei welcher das Minimierungsproblem einfach zu lösen ist und verwenden dies als allgemeine Approximation:

Satz 3.4

Sei $A \in \mathbb{C}^{n \times n}$ und bezeichne $\text{tr}(A)$ die Spur von A . Dann gilt:

$$\operatorname{argmax}_{\mu \in \mathbb{R}} \|A - \mu I\|_F = \frac{\text{tr}(A)}{n}$$

Beweis

Da $\mu \in \mathbb{R}$ ist, gilt $(\mu I)^H = (\mu I)^T = \mu I$. Der Rest folgt direkt aus der Definition der Frobeniusnorm:

$$\begin{aligned} \|A - \mu I\|_F^2 &= \text{tr}((A - \mu I)^H(A - \mu I)) = \text{tr}(A^H A - \mu A^H - \mu A + \mu^2 I) \\ &= \text{tr}(A^H A) - 2\mu \text{tr}(A) + n\mu^2 \end{aligned}$$

Nun leiten wir nach μ ab, setzen dies gleich Null und erhalten:

$$\mu = \text{tr}(A)/n$$

Die zweite Ableitung zeigt noch auf, dass es sich tatsächlich um ein Minimum handelt. \square

Da die Spur gleich der Summe der Eigenwerte ist, kann man $\text{tr}(A)/n$ als den Mittelwert der Eigenwerte interpretieren. Somit schreiben wir:

$$\tilde{A} = A - (\text{tr}(A)/n)I \quad \text{und} \quad e^A = e^{\text{tr}/n} e^{\tilde{A}}$$

wobei wir $e^{\text{tr}/n} e^{\tilde{A}} = e^{\text{tr}/n I} e^{\tilde{A}}$ verwendet haben.

Higham [12] verweist darauf, dass diese Translation nicht immer geeignet ist. Wenn A ein oder mehrere Eigenwerte mit grossem negativem Realteil hat, so kann μ sehr gross und negativ sein. Schliesslich hat dann $A - \mu I$ einen Eigenwert mit grossem positivem Realteil, was zu Schwierigkeiten führen kann.

3.4.3 Ausgleichen

Ausgleichen versucht die 1-Norm von A (resp. \tilde{A}) über alle möglichen nichtsingulären Diagonalmatrizen mit der Ähnlichkeitstransformation zu minimieren, d.h. $\min_{D \in \mathbb{D}} \|D^{-1} \tilde{A} D\|_1$, wobei \mathbb{D} die Menge aller nichtsingulären $n \times n$ Diagonalmatrizen ist. Parlett und Reinsch [15] beschreiben einen Algorithmus *BALANCE*, welcher diese Minimierung über die Menge \mathbb{D}_u zu erreichen versucht, wobei im Voraus noch Permutationen verwendet werden. Die Menge \mathbb{D}_u bezeichnet alle $n \times n$ nichtsingulären Diagonalmatrizen, deren Einträge ganzzahlige Potenzen der Maschinenbasis B (fast immer $B = 2$) sind. Der Algorithmus *BALANCE* gibt die transformierte Matrix $\tilde{\tilde{A}}$, die Diagonalmatrix D und die Permutationsmatrix P zurück. Somit können wir zusammenfassend schreiben:

$$\tilde{\tilde{A}} = D^{-1} P^T \tilde{A} P D \quad \text{und} \quad e^A = e^{\text{tr}(A)/n} P D e^{\tilde{\tilde{A}}} D^{-1} P^T$$

Dieser Algorithmus ist in LAPACK und MATLAB implementiert und hat Kosten von $O(n^2)$, wobei es keine Rundungsfehler gibt.

Auch hier weist Higham [12] darauf hin, dass dieses Ausgleichen nicht garantiert, die Norm von A zu verkleinern. Daher muss man aufpassen, dass man A nur durch das ausgeglichene B ersetzt, wenn $\|B\| < \|A\|$ ist. Weiter vermerkt er, dass sich gezeigt hat, dass die Vorverarbeitung wenig Einfluss auf die Kosten oder die Exaktheit des Algorithmus hat. Da sich teilweise dabei sogar die Exaktheit verschlechtert, empfiehlt er nicht die automatische Benützung dieser Vorverarbeitungen. Unsere Tests (siehe Appendix A) zeigten jedoch, dass das Ausgleichen meistens sehr viel bringt. Daher sind wir der Meinung, dass Ausgleichen sehr wertvoll ist und wir die automatische Benützung empfehlen.

3.4.4 Ward's Algorithmus

Dieser Algorithmus wurde von Ward 1977 [18] vorgestellt und ist in diversen Bibliotheken implementiert, wie zum Beispiel bei Octave oder beim R-Package *Matrix* in R. Das Prinzip des Algorithmus ist ebenfalls die „Scaling & Squaring“-Methode, wobei die Parameter anders gesetzt und zusätzlich noch die beiden Vorverarbeitungen verwendet werden. Folgender Algorithmus beschreibt die Implementation nach Ward [18], wobei für die Padé-Approximation der Grad 8 gewählt wurde:

Algorithmus 3.5 : Ward 1977

- 1: Vorverarbeitung 1: Norm-minimierende Translation $A \leftarrow A - (\text{tr}(A)/n)I$
- 2: Vorverarbeitung 2: Ausgleichen mit *BALANCE*.
 $B \leftarrow \text{BALANCE}(A)$
Man erhält die Matrix B , die Diagonalmatrix D und Informationen zur Permutationsmatrix P .
- 3: **if** $\|B\|_1 \leq 1$ **then**
- 4: Finde $s > 0$, sodass $2^{-s}\|B\|_1 \leq 1$ und berechne $B \leftarrow B2^{-s}$.
- 5: **end if**
- 6: Berechne die Padé-Approximation r_8 .
- 7: **if** $s > 0$ **then**
- 8: Berechne $e^B \leftarrow (e^B)^{2^s}$.
- 9: **end if**
- 10: Berechne $e^A \leftarrow e^{\text{tr}(A)/n} P D e^B D^{-1} P^T$

Die Kosten belaufen sich hier auf $(8 + \log_2(\|B\|))M + D$. Es ist zu bemerken, dass der Algorithmus in seinen Implementationen entweder die Horner-Methode oder die explizite Potenzierung verwendet. Hier könnte man sicherlich mit der Paterson-Stockmeyer Methode die Kosten etwas reduzieren und die Padé-Approximation mit Grade 9 wählen, was den Algorithmus effizienter gestalten und die Genauigkeit erhöhen würde.

Für Tests, die die beiden Algorithmen auf Laufzeit und als auch Genauigkeit vergleichen, verweisen wir auf den Appendix A.

4 Exponentialkonditionszahl

Im letzten Kapitel haben wir den Algorithmus *Scaling & Squaring* untersucht, der sehr effizient und akkurat das Matrixexponential berechnet. Diesen möchten wir nun ausbauen um die Exponentialkonditionszahl zu berechnen. Dieser wertvolle Indikator zeigt, wie die Exponentialfunktion auf kleine Störungen in A reagiert:

$$\text{cond}(f, A) := \lim_{\varepsilon \rightarrow 0} \sup_{\|E\| \leq \varepsilon \|A\|} \frac{\|f(A + E) - f(A)\|}{\varepsilon \|f(A)\|}$$

Um uns dieser anzunähern, werden wir die Fréchet-Ableitung einführen, mit welcher wir diese Definition etwas vereinfacht schreiben können. Danach erweitern wir unseren Algorithmus 3.3 um die Fréchet-Ableitung zu berechnen, welche wir dann verwenden können, um $\text{cond}(f, A)$ auszuwerten. Die folgenden Ausführungen beziehen sich auf den Artikel „Computing the Fréchet Derivative of the Matrix Exponential, with an Application to the Condition Number Estimation“ von Higham und Al-Mohy [1].

4.1 Die Fréchet-Ableitung

Die Sensitivität einer Matrixfunktion f bei einer kleinen Störung E wird durch die Fréchet-Ableitung charakterisiert.

Definition 4.1 (Fréchet-Ableitung)

Sei $A, E \in \mathbb{C}^{n \times n}$, wobei A die Ausgangsmatrix und E die Störungsmatrix beschreibt. Weiter sei $f : \mathbb{C}^{n \times n} \rightarrow \mathbb{C}^{n \times n}$ eine Matrixfunktion. Dann wird die Fréchet-Ableitung durch folgende in E lineare Abbildung beschrieben:

$$\begin{aligned} \mathbb{C}^{n \times n} &\xrightarrow{L_f(A)} \mathbb{C}^{n \times n} \\ E &\mapsto L_f(A, E) \end{aligned}$$

sodass für alle $E \in \mathbb{C}^{n \times n}$ folgende Gleichheit gilt:

$$f(A + E) - f(A) - L_f(A, E) = o(\|E\|)$$

In diesem Sinne beschreibt es den Einfluss von E auf die erste Ableitung von f in A .

Nun können wir die Exponentialkonditionszahl wie folgt schreiben, wobei wir für die Herleitung dieser Gleichheit auf [12], Seite 56, Theorem 3.1. verweisen:

$$\text{cond}(f, A) = \frac{\|L_f(A)\| \|A\|}{\|f(A)\|}$$

wobei $\|L_f(A)\| := \max_{Z \neq 0} \frac{\|L_f(A, Z)\|}{\|Z\|}$ und die Norm $\|\cdot\|$ eine beliebige Matrixnorm ist. Genau der Ausdruck $\|L_f(A)\|$ birgt die Schwierigkeit der Berechnung, welchen wir später noch schätzen. Aber im Moment möchten wir unsere Erkenntnisse aus dem letzten Kapitel verwenden und ausbauen, sodass der neue Algorithmus gleichzeitig $\exp(A)$ und $L_f(A, E)$ auswertet, was wir wiederum verwenden können um $\text{cond}(\exp, A)$ zu berechnen.

Als nächstes werden wir noch eine sehr nützliche Beziehung herleiten, die essentiell für unseren Algorithmus sein wird. Hierzu muss jedoch f als Potenzreihe geschrieben werden können, was bei uns ja der Fall ist.

Satz 4.2

Sei f eine Funktion, die durch eine Potenzreihe beschrieben werden kann, also $f(x) = \sum_{k=0}^{\infty} a_k x^k$ mit Konvergenzradius r . Dann kann die Fréchet-Ableitung für $A, E \in \mathbb{C}^{n \times n}$ mit $\|A\| < r$ wie folgt geschrieben werden:

$$L_f(A, E) = \sum_{k=1}^{\infty} a_k \sum_{j=1}^k A^{j-1} E A^{k-j}.$$

Beweis

Da $L_f(A, E)$ der lineare Term in E ist, können wir $f(A + E)$ wie folgt schreiben:

$$\begin{aligned} f(A + E) &= \sum_{k=0}^{\infty} a_k (A + E)^k = \sum_{k=0}^{\infty} a_k A^k + \sum_{k=1}^{\infty} a_k \sum_{j=1}^k A^{j-1} E A^{k-j} + O(\|E\|^2) \\ &= f(A) + L_f(A, E) + O(\|E\|^2) \end{aligned}$$

Ist nun $\|A\| < r$, so können wir E so mit θ skalieren, dass $\|A + \theta E\| < r$ ist und $L_f(A, \theta E) = \theta L_f(A, E)$. □

Das nächste Theorem zeigt eine Beziehung auf, die wir verwenden können um die Reihe aus Satz 4.2 zu berechnen.

Satz 4.3

Sei f gleich wie in Satz 4.2. Dann können wir die Fréchet-Ableitung wie folgt schreiben:

$$L_f(A, E) = \sum_{k=1}^{\infty} a_k M_k,$$

Wobei $M_k = L_{x^k}(A, E)$ die Gleichung

$$M_k = M_{l_1} A^{l_2} + A^{l_1} M_{l_2}, \quad M_1 = E \tag{1}$$

erfüllt. Dabei gelte $k = l_1 + l_2$ und l_1 und l_2 sind positive ganze Zahlen. Im Speziellen gilt:

$$M_k = M_{k-1} A + A^{k-1} E, \quad M_1 = E. \tag{2}$$

Zusätzlich gilt für $\tilde{f}(x) = \sum_{k=0}^{\infty} |a_k| x^k$:

$$\|f(A)\| \leq \tilde{f}(\|A\|), \quad \|L_f(A)\| \leq \tilde{f}'(\|A\|).$$

Beweis

Da wir die Potenzreihe innerhalb des Konvergenzkreises termweise differenzieren können, haben wir

$$L_f(A, E) = \sum a_k M_k, \quad M_k = L_{x^k}(A, E).$$

Die Beziehung (1) folgt direkt aus der Produktregel für die Fréchet-Ableitung.

Lemma 4.4

Sei g und h Fréchet-differenzierbar in A und sei $f = gh$, dann ist

$$L_f(A, E) = L_g(A, E)h(A) + g(A)L_h(A, E).$$

Beweis

$$\begin{aligned} f(A + E) &= g(A + E)h(A + E) = (g(A) + L_g(A, E) + o(\|E\|))(h(A) + L_h(A, E) + o(\|E\|)) \\ &= f(A) + L_g(A, E)h(A) + g(A)L_h(A, E) + o(\|E\|) \end{aligned}$$

□

Daraus folgt direkt (2) und mit $l_1 = k - 1$ und $l_2 = 1$ folgt auch (2). Des Weiteren ist es offensichtlich, dass $\|f(A)\| \leq \tilde{f}(\|A\|)$ gilt. Nehmen wir noch die Norm der Potenzreihendarstellung der Fréchet-Ableitung, so folgt:

$$\|L_f(A, E)\| \leq \|E\| \sum_{k=1}^{\infty} k|a_k| \|A\|^{k-1} = \|E\| \tilde{f}'(\|A\|)$$

Das Maximieren über alle nicht verschwindende E ergibt $\|L_f(A)\| \leq \tilde{f}'(\|A\|)$. □

Damit haben wir die wichtigsten Eigenschaften der Fréchet-Ableitung hergeleitet. Vor allem die Gleichung (1) wird von grossem Nutzen sein.

4.2 Numerische Annäherung

4.2.1 Die Fréchet-Ableitung im numerischen Kontext

Im letzten Kapitel haben wir gesehen, dass wir einen sehr guten Algorithmus für die Exponentialfunktion mit der Padé-Approximation r_m und geeignetem m bilden konnten. Dies möchten wir nun ebenfalls tun für die Fréchet-Ableitung, also L_f durch L_{r_m} approximieren. Das folgende Lemma zeigt uns auf, wie wir diese Approximation L_{r_m} effizient evaluieren können.

Lemma 4.5

Die Fréchet-Ableitung L_{r_m} einer rationalen Funktion $r_m = p_m(x)/q_m(x)$ erfüllt folgende Gleichung:

$$q_m(A)L_{r_m}(A, E) = L_{p_m}(A, E) - L_{q_m}(A, E)r_m(A).$$

Beweis

Diese Relation geht direkt aus dem Lemma 4.4 hervor, welches auf $q_m r_m = p_m$ angewendet folgende Gleichung ergibt:

$$L_{p_m}(A, E) = L_{q_m r_m}(A, E) = L_{q_m}(A, E)r_m(A) + q_m(A)L_{r_m}(A, E)$$

Durch Umformen erhalten wir die gewünschte Beziehung. \square

Nun stellen wir uns noch die Frage, wie teuer die Berechnung ist, um die Fréchet-Ableitung eines Polynoms auszuwerten. Der folgende Satz gibt uns hierzu eine Antwort.

Satz 4.6

Sei p ein Polynom, sei $A, E \in \mathbb{C}^{n \times n}$ und bezeichne π_p die Kosten für die Auswertung von $p(A)$ bei einer beliebigen Polynomauswertung aus Kapitel 2.2. Weiter bezeichne σ_p die Extrakosten um $L_p(A, E)$ auszuwerten, indem man die Polynomauswertung für $p(A)$ differenziert. Dann ist $\sigma_p \leq 2\pi_p$.

Beweis

Zuerst überlegen wir uns, wie wir all diese Methoden zur Polynomauswertung allgemein beschreiben können. Hierzu betrachte man das Polynom p_m mit Grade m , welches durch s Schritte der folgenden Form ausgewertet wird:

$$q_1^{(k)}(A) = q_2^{(k-1)}(A)q_3^{(k-1)}(A) + q_4^{(k-1)}(A), \quad k = 1 : s$$

$$\deg q_i^{(k)} < m, \quad i = 1 : 4, \quad k < s, \quad \deg q_i^{(k)} \geq 1, \quad i = 2 : 3$$

wobei die Polynome $q_i^{(k)}, i = 2 : 4$ Linearkombinationen von $q_1^{(1)}, \dots, q_1^{(k-1)}$ sind und $p_m(A) = q_1^{(s)}(A)$ ist. Dieses Schema deckt nun alle vorgestellten Methoden zur Polynomauswertung ab, sei es Horner's Methode, die explizite Potenzierung oder die Paterson-Stockmeyer Methode.

Nun können wir diese Struktur verwenden um einen Beweis durch Induktion zu führen. Sei p_m ein Polynom vom Grade m und als Verankerung sei $m = 1$. Somit ist $p_1(A) = b_0 I + b_1 A$ die einzige mögliche Auswertung mit $\pi_p = 0$. Die korrespondierende Auswertung der Fréchet-Ableitung ist damit $L_{p_1}(A, E) = b_1 E$ mit $\sigma_p = 0$ und der Satz gilt somit für $m = 1$.

Nun nehmen wir an, dass die Aussage stimmt für alle Grade bis $m - 1$ und sei nun p_m ein Polynom vom Grade m . Somit ist der letzte Schritt für p_m folgende Rechnung:

$$p_m = q_1^{(s)}(A) = q_2^{(s-1)}(A)q_3^{(s-1)}(A) + q_4^{(s-1)}(A)$$

wobei die Polynome $q_i^{(s-1)}, i = 2 : 4$ alle vom Grade kleiner als m sind. Somit ist $\pi_{p_m} = \pi_{q_2} + \pi_{q_3} + \pi_{q_4} + 1$ und nach der Induktionshypothese gilt $\sigma_{q_i} \leq 2\pi_{q_i}$ für $i = 2 : 4$. Nach der Produktregel (Lemma 4.4) gilt nun

$$L_{p_m}(A, E) = L_{q_2}(A, E)q_3(A) + q_2(A)L_{q_3}(A, E) + L_{q_4}(A, E)$$

und somit gilt

$$\sigma_{p_m} \leq \sigma_{q_2} + \sigma_{q_3} + \sigma_{q_4} + 2 \leq 2(\pi_{q_2} + \pi_{q_3} + \pi_{q_4} + 1) = 2\pi_{p_m}.$$

Es ist zu erwahnen, dass dieser Beweis keine Beziehungen zwischen den einzelnen Polynomen $q_i^{(k)}$ verwendet, welche die Kosten noch reduzieren konnten. Aber diese wurden sich auch auf die Frechet-Ableitung ubertragen, sodass das Resultat gultig bleibt. \square

Die Idee ist es nun, den Algorithmus 3.3 auszubauen. Hierzu mussen wir lediglich:

- Nebst p_m und q_m auch noch L_{p_m} und L_{q_m} mit (1) und (2) auswerten.
- Zusatzlich zur Gleichung $q_m(A)r_m(A) = p_m(A)$ nach $r_m(A)$ muss man noch $q_m(A)L_{r_m}(A, E) = L_{p_m}(A, E) - L_{q_m}(A, E)r_m(A)$ nach $L_{r_m}(A, E)$ losen.

Im Rahmen der Polynomauswertungen kostet diese Prozedur somit $(\pi_m + 2\pi_m + 1)M + 2D$.

4.2.2 Herleitung des Algorithmus fur die Frechet-Ableitung

In diesem Abschnitt werden wir analysieren, wie wir das Erlernte benutzen konnen, um den Algorithmus 3.3 zu adaptieren. Zuerst betrachten wir nochmals die Schlusselgleichung der „Scaling & Squaring“-Methode $e^A = (e^{A/2})^2$. Diese differenzieren wir unter Verwendung von $L_{x^2}(A, E) = AE + EA$ und der Kettenregel, welche wir hier kurz einfuhren.

Satz 4.7

Seien h und g Frechet-differenzierbar in A resp. $h(A)$ und sei $f = g \circ h$. Dann ist f Frechet-differenzierbar in A und $L_f = L_g \circ L_h$, also $L_f(A, E) = L_g(h(A), L_h(A, E))$.

Beweis

$$\begin{aligned} f(A + E) - f(A) &= g(h(A + E)) - g(h(A)) \\ &= g(h(A) + L_h(A, E) + o(\|E\|)) - g(h(A)) \\ &= g(h(A)) + L_g(h(A), L_h(A, E) + o(\|E\|)) + o(\|E\|) - g(h(A)) \\ &= L_g(h(A), L_h(A, E)) + o(\|E\|) \end{aligned}$$

\square

Somit folgt nun direkt:

$$\begin{aligned} L_{\exp}(A, E) &= L_{x^2}(e^{A/2}, L_{\exp}(A/2, E/2)) \\ &= e^{A/2}L_{\exp}(A/2, E/2) + L_{\exp}(A/2, E/2)e^{A/2} \end{aligned} \tag{3}$$

Durch wiederholtes Anwenden dieser Beziehung erhalten wir:

$$\tilde{L}_s = L_{\text{exp}}(2^{-s}A, 2^{-s}E)$$

$$\tilde{L}_{i-1} = e^{2^{-i}A}\tilde{L}_i + \tilde{L}_ie^{2^{-i}A}, \quad i = s : -1 : 1$$

wobei $\tilde{L}_0 = L_{\text{exp}}(A, E)$. Für die numerische Realisierung verwenden wir an der Stelle von L_{exp} die Approximation L_{r_m} und für $e^{2^{-i}A}$ den Ausdruck $r_m(2^{-s}A)^{2^{s-i}}$, welche die \tilde{L}_i durch L_i approximiert:

$$\begin{aligned} X_s &= r_m(2^{-s}A) \\ L_s &= L_{r_m}(2^{-s}A, 2^{-s}E) \\ \left. \begin{aligned} L_{i-1} &= X_iL_i + L_iX_i \\ X_{i-1} &= X_i^2 \end{aligned} \right\} \quad i = s : -1 : 1. \end{aligned} \quad (4)$$

Man fragt natürlich sofort nach der Stabilität und Genauigkeit von L_0 relativ zur Stabilität und Genauigkeit der Approximation $X_0 = (r_m(2^{-s}A))^{2^s}$ zu e^A . Hierzu betrachte man den leicht adaptierten Satz 3.2:

Satz 4.8

Sei

$$\|e^{-A}r_m(A) - I\| < 1, \quad \|A\| < \min\{|t| : q_m(t) = 0\} \quad (5)$$

für eine beliebige konsistente Matrixnorm, sodass $g_m(A) = \log(e^{-A}r_m(A))$ existiert. Sei $r_m(A) = e^{A+g_m(A)}$ und $\|g_m(A)\| \leq -\log(1 - \|e^{-A}r_m(A) - I\|)$. Im Besonderen gilt, falls die Voraussetzung (5) für $2^{-s}A$ erfüllt ist:

$$r_m(2^{-s}A) = e^{2^{-s}A+g_m(2^{-s}A)} \quad (6)$$

und somit ist $r_m(2^{-s}A)^{2^s} = e^{A+2^s g_m(2^{-s}A)}$, wobei folgende Abschätzung gilt:

$$\frac{\|2^s g_m(2^{-s}A)\|}{\|A\|} \leq \frac{-\log(1 - \|e^{-2^{-s}A}r_m(2^{-s}A) - I\|)}{\|2^{-s}A\|}.$$

Nun differenzieren wir die Gleichung (6) durch Verwendung der Kettenregel:

$$\begin{aligned} L_s &= L_{r_m}(2^{-s}A, 2^{-s}E) \\ &= L_{\text{exp}}(2^{-s}A + g_m(2^{-s}A), 2^{-s}E + L_{g_m}(2^{-s}A, 2^{-s}E)) \end{aligned} \quad (7)$$

Somit folgt aus dem Hergeleiteten:

$$\begin{aligned} L_{s-1} &\stackrel{(4)}{=} r_m(2^{-s}A)L_s + L_s r_m(2^{-s}A) \\ &\stackrel{(7)}{=} e^{2^{-s}A+g_m(2^{-s}A)}L_{\text{exp}}(2^{-s}A + g_m(2^{-s}A), 2^{-s}E + L_{g_m}(2^{-s}A, 2^{-s}E)) \\ &\quad + L_{\text{exp}}(2^{-s}A + g_m(2^{-s}A), 2^{-s}E + L_{g_m}(2^{-s}A, 2^{-s}E))e^{2^{-s}A+g_m(2^{-s}A)} \\ &\stackrel{(3)}{=} L_{\text{exp}}(2^{-(s-1)}A + 2g_m(2^{-s}A), 2^{-(s-1)}E + L_{g_m}(2^{-s}A, 2^{-(s-1)}E)) \end{aligned}$$

wobei im letzten Schritt noch die Linearität im zweiten Argument der Fréchet-Ableitung benützt wurde. Durch induktives Weiterführen dieser Umformung und unter Verwendung der Relation

$$X_i = X_s^{2^{s-i}} = \left(e^{2^{-s}A + g_m(2^{-s}A)} \right)^{2^{s-i}} = e^{2^{-i}A + 2^{s-i}g_m(2^{-s}A)}$$

erhalten wir das folgende Theorem.

Satz 4.9

Sei (5) erfüllt für $2^{-s}A$, so ist

$$L_0 = L_{\exp}(A + 2^s g_m(2^{-s}A), E + L_{g_m}(2^{-s}A, E))$$

Wiederum haben wir hier eine Rückwärtsfehleranalyse, wobei wir nun die exakte Fréchet-Ableitung der Exponentialfunktion einer gestörten Matrix in eine gestörte Richtung haben. Es ist darauf hinzuweisen, dass dieser Rückwärtsfehler lediglich den Abbruchfehler der Padé-Approximation einbezieht und allfällige Rundungsfehler ignoriert.

Wir sehen also, dass $X_0 = e^{A+\Delta A}$ und $L_0 = L_{\exp}(A + \Delta A, E + \Delta E)$ ist, mit $\Delta A = 2^s g_m(2^{-s}A)$. Wir haben ja bereits im letzten Kapitel gesehen, wie wir m und s wählen müssen, sodass ΔA genügend klein ist. Nun müssen wir noch untersuchen wie sich dies mit der Norm von $\Delta E = L_{g_m}(2^{-s}A, E)$ verhält.

Sei nun $\tilde{g}_m(x) = \sum_{k=2m+1}^{\infty} c_k x^k$ die Potenzreihe von $g_m(x) = \log(e^{-x}r_m(x))$ mit absoluten Beträgen bei den Koeffizienten. Verwenden wir noch die letzte Ungleichung im Theorem 4.3, so erhalten wir

$$\frac{\|\Delta E\|}{\|E\|} = \frac{\|L_{g_m}(2^{-s}A, E)\|}{\|E\|} \leq \|L_{g_m}(2^{-s}A)\| \leq g'_m(\theta),$$

wobei $\theta = \|2^{-s}A\|$ ist. Wir definieren (wie die θ_m bei der Herleitung der „Scaling & Squaring“-Methode) $l_m = \max\{\theta : \tilde{g}'_m(\theta) \leq u\}$, wobei $u = 2^{-53} \approx 1.1 \times 10^{-16}$ die Maschinengenauigkeit bezeichnet. Wiederum wurde mit MATLAB's Symbolic Math Toolbox l_m für $m = 1 : 20$ berechnet, indem die ersten 150 Terme der Reihe symbolisch mit einer 250 Stellenarithmetik addiert wurden. Diese Tabelle wurde von [1] übernommen.

m	1	2	3	4	5	6	7	8	9	10
θ_m	3.7e-8	5.3e-4	1.5e-2	8.5e-2	2.5e-1	5.4e-1	9.5e-1	1.5e0	2.1e0	2.8e0
l_m	2.1e-8	3.6e-4	1.1e-2	6.5e-2	2.0e-1	4.4e-1	7.8e-1	1.2e0	1.8e0	2.4e0

m	11	12	13	14	15	16	17	18	19	20
θ_m	3.6e0	4.5e0	5.4e0	6.3e0	7.3e0	8.4e0	9.4e0	1.1e1	1.2e1	1.3e1
l_m	3.1e0	3.9e0	4.7e0	5.6e0	6.6e0	7.5e0	8.5e0	9.6e0	1.1e1	1.2e1

Wir sehen, dass für alle $m = 1 : 20$ die $l_m < \theta_m$ sind, was nicht besonders überraschend ist, da wir die Approximation L_{r_m} eher wegen praktischem Nutzen als den numerischen

Eigenschaften gewählt haben - zumindest ist das Verhältnis θ_m/l_m nahezu 1. Somit ist für jedes m mit $\theta \leq l_m$ die Rückwärtsstabilität gewährleistet:

$$X_0 = e^{A+\Delta A}, \quad L_0 = L_{\exp}(A + \Delta A, E + \Delta E), \quad \|\Delta A\| \leq u\|A\|, \|\Delta E\| \leq u\|E\|$$

Um nun den Algorithmus konstruieren zu können, müssen wir noch die Evaluationskosten für L_{r_m} genauer analysieren. Wir haben ja bereits gezeigt, dass wenn diese für r_m gleich π_m sind, so benötigen wir für L_{r_m} zusätzlich höchstens $2\pi_m$ Multiplikationen. Aber dies möchten wir jetzt doch noch genauer wissen.

Wir haben im Kapitel 2.3 die spezielle Eigenschaft der Padé-Approximation für die Exponentialfunktion ausgenutzt und p_m in ungerade (u_m) und gerade Potenzen (v_m) zerlegt, sodass $p_m = u_m + v_m$ und $q_m = -u_m + v_m$ ist. Dies wird ja durch das Differenzieren weiter vererbt und wir haben

$$L_{p_m} = L_{u_m} + L_{v_m}, \quad L_{q_m} = -L_{u_m} + L_{v_m}.$$

Genauer folgt aus

$$p_m(x) = x \sum_{k=0}^{(m-1)/2} b_{2k+1}x^{2k} + \sum_{k=0}^{(m-1)/2} b_{2k}x^{2k} =: u_m(x) + v_m(x)$$

direkt

$$L_{u_m}(A, E) = A \sum_{k=1}^{(m-1)/2} b_{2k+1}M_{2k} + E \sum_{k=0}^{(m-1)/2} b_{2k+1}A^{2k}$$

$$L_{v_m}(A, E) = \sum_{k=1}^{(m-1)/2} b_{2k}M_{2k},$$

wobei die $M_k = L_{x^k}(A, E)$ mit der Gleichung (1) berechnet werden können. Für $m = 13$ haben wir die noch effizientere Aufteilung $p_{13} = u_{13} + v_{13}$ verwendet, mit

$$u_{13}(x) = xw(x), \quad w(x) = x^6w_1(x) + w_2(x), \quad v_{13}(x) = x^6z_1(x) + z_2(x),$$

$$w_1(x) = b_{13}x^6 + b_{11}x^4 + b_9x^2, \quad w_2(x) = b_7x^6 + b_5x^4 + b_3x^2 + b_1$$

$$z_1(x) = b_{12}x^6 + b_{10}x^4 + b_8x^2, \quad z_2(x) = b_6x^6 + b_4x^4 + b_2x^2 + b_0.$$

Das Differenzieren führt wiederum zu folgenden Ausdrücken

$$L_{u_{13}}(A, E) = AL_w(A, E) + Ew(A),$$

$$L_{v_{13}}(A, E) = A^6L_{z_1}(A, E) + M_6z_1(A) + L_{z_2}(A, E),$$

mit

$$L_w(A, E) = A^6L_{w_1}(A, E) + M_6w_1(A) + L_{w_2}(A, E),$$

$$L_{w_1}(A, E) = b_{13}M_6 + b_{11}M_4 + b_9M_2,$$

$$L_{w_2}(A, E) = b_7M_6 + b_5M_4 + b_3M_2,$$

$$L_{z_1}(A, E) = b_{12}M_6 + b_{10}M_4 + b_8M_2,$$

$$L_{z_2}(A, E) = b_6M_6 + b_4M_4 + b_2M_2.$$

Somit ist $L_{p_{13}} = L_{u_{13}} + L_{v_{13}}$ und $L_{q_{13}} = -L_{u_{13}} + L_{v_{13}}$. Schliesslich muss man noch die folgenden Gleichungen für $r_m(A)$ und $L_{r_m}(A, E)$ lösen:

$$(-u_m + v_m)(A)r_m(A) = (u_m + v_m)(A)$$

$$(-u_m + v_m)(A) = L_{r_m}(A, E) = (L_{u_m} + L_{v_m})(A, E) + (L_{u_m} - L_{v_m})(A, E)r_m(A).$$

Natürlich müssen wir noch den Padé-Approximationsgrad m als auch den Skalierungsparameter s optimal bestimmen. In der folgenden Tabelle ist jeweils totale Anzahl Matrixmultiplikationen $w_m = 3\pi_m + 1$ angegeben, die benötigt werden um r_m und L_{r_m} auszuwerten, basierend auf der obigen Analyse und der Tabelle auf Seite 18.

m	1	2	3	4	5	6	7	8	9	10
w_m	1	4	7	10	10	13	13	16	16	19
C_m	25.5	12.5	8.5	6.9	5.3	5.2	4.4	4.7	4.2	4.7

m	11	12	13	14	15	16	17	18	19	20
w_m	19	19	19	22	22	22	222	25	25	25
C_m	4.4	4	3.8	4.5	4.3	4.1	3.9	4.7	4.6	4.5

Um die Gesamtkosten analysieren zu können, müssen wir die „Squaring“-Phase mit in Betracht ziehen. Ist $\|A\| > l_m$, so müssen wir das A skalieren, sodass $\|2^{-s}A\| \leq l_m$ gilt, also mit $s = \lceil \log_2(\|A\|/l_m) \rceil$. Von der Beziehung (4) ist ersichtlich, dass wir jeweils $3s$ Matrixmultiplikationen benötigen (s für r_m , $2s$ für L_{r_m}). Somit haben wir Gesamtkosten in Matrixmultiplikationen von

$$w_m + 3s = 3\pi_m + 1 + 3 \max\{\lceil \log_2 \|A\| - \log_2 l_m \rceil, 0\}.$$

Da $\log_2 \|A\|$ nur als Verschiebung wirkt und die 3 ein konstanter Faktor ist, müssen wir lediglich den folgenden Ausdruck minimieren:

$$C_m = \pi_m - \log_2 l_m.$$

Wir sehen, dass dies erreicht wird bei $m = 13$, genau wie bei der „Scaling & Squaring“-Methode der Exponentialfunktion. Weiter muss man $m = 1, 2, 3, 5, 7, 9$ auch in Betracht ziehen, wenn $\|A\| < l_{13}$ ist, wobei wir $m = 1, 2$ sofort wieder ausschliessen wegen der möglichen starken Auslöschung.

Schliesslich müssen wir noch prüfen, ob die Auswertungen genügend akkurat sind in der „Floating Point“-Arithmetik. Wir wissen ja aus den Analysen im Kapitel 3.2, dass die $q_m(A)$ gut konditioniert sind. Durch Anwendung von Satz 2.6 erhalten wir die folgende Abschätzung für L_{p_m}

$$\|L_{p_m}(A, E) - \hat{L}_{p_m}(A, E)\| \leq \tilde{\gamma}_{n^2} p'_m(\|A\|) \|E\| \approx \tilde{\gamma}_{n^2} e^{\|A\|/2} \|E\|.$$

Hier wurde noch verwendet, dass $p_m(x) \approx e^{x/2}$ ist. Es ist ersichtlich, dass diese Schranke für $\|A\| \leq l_{13}$ genügend klein ist. Eine analoge Schranke kann man auch für L_{q_m} erhalten, da $q_m(x) = p_m(-x)$ ist.

Schliesslich können wir mit diesen Erkenntnissen den Algorithmus 3.3 ausbauen. Sei $A, E \in \mathbb{C}^{n \times n}$ gegeben. Der folgende Algorithmus berechnet $X = e^A$ und $L = L_{\exp}(A, E)$ mit der „Scaling & Squaring“ Methode.

Algorithmus 4.10

```

1: for  $m$  in  $[3, 5, 7, 9]$  do
2:   if  $\|A\|_1 \leq l_m$  then
3:     Berechne  $u_m$  and  $v_m$  und löse  $(-u_m + v_m)r_m = u_m + v_m$  nach  $r_m$ 
4:     Berechne  $L_u = L_{u_m}(A, E)$ ,  $L_v = L_{v_m}(A, E)$  und
       löse  $(-u_m + v_m)L = L_u + L_v + (L_u - L_v)R$  nach  $L$ 
5:     Quit
6:   end if
7: end for
8:  $A \leftarrow A/2^s$  mit  $s \in \mathbb{N}$ , sodass  $\|A/2^s\|_1 \leq l_{13}$ 
9: Berechne  $X = r_{13}$ 
10: Berechne  $L = L_{r_{13}}$ 
11: for  $k=1:s$  do
12:    $X \leftarrow X^2$ 
13:    $L \leftarrow XL + LX$ 
14: end for

```

Wie wir bereits hergeleitet haben, belaufen sich die Kosten auf $(w_m + 3s)M + 2D = (3\pi_m + 1 + 3s)M + D$. Im Vergleich hierzu benötigt der Algorithmus 3.3 $(\pi_m + s)M + D$, also etwa 3 mal soviel Matrixmultiplikationen.

Da $L_{\exp}(A, \alpha E) = \alpha L_{\exp}(A, E)$ ist, sollte ein Algorithmus nicht besonders durch $\|E\|$ beeinflusst werden, wie es auch der Fall ist für den Algorithmus 4.10. Damit können wir direkt die Fréchet-Ableitung als auch e^A berechnen. Im nächsten Abschnitt möchten wir den Algorithmus so weiterverwenden, dass wir die Exponentialkonditionszahl schätzen können.

4.3 Konditionszahl

4.3.1 Exakte Berechnung

Wir kehren nun zurück zur Exponentialkonditionszahl.

$$\text{cond}(\exp, A) = \frac{\|L_{\exp}(A)\| \|A\|}{\|e^A\|}$$

Unser Algorithmus 4.10 kann zwar $L_{\exp}(A, E)$ für ein beliebiges E berechnen, aber für $\|L_{\exp}(A)\|$ müssen wir die Norm von $L_{\exp}(A, E)$ über alle E maximieren.

Für den Moment betrachten wir ein allgemeines f . Da $L_f(A, E)$ in E linear ist, können wir folgende Gleichung schreiben:

$$\text{vec}(L_f(A, E)) = \mathcal{K}(A) \text{vec}(E) \quad (8)$$

Hier bezeichne vec den Operator, der alle Spalten einer Matrix in einen Vektor der Länge n^2 stapelt. Somit ist $\mathcal{K}(A) \in \mathbb{C}^{n^2 \times n^2}$, $\text{vec}(E) \in \mathbb{C}^{n^2}$ und $\mathcal{K}(A)$ ist unabhängig von E . Wir nennen $\mathcal{K}(A)$ die Kroneckerform der Fréchet-Ableitung. Nun folgt direkt, dass $\|L_f(A, E)\|_F = \|\mathcal{K}(A) \text{vec}(E)\|_2$ ist und durch dividieren mit $\|E\|_F = \|\text{vec}(E)\|_2$ und maximieren über alle E erhalten wir:

$$\|L_f(A)\|_F = \|\mathcal{K}(A)\|_2.$$

Dadurch können wir $\|L_f(A)\|_F$ exakt berechnen, indem wir $\mathcal{K}(A)$ formen. Offensichtlich hat $\mathcal{K}(A)$ die Spalten $\text{vec}(L_f(A, e_i e_j^T))$ für $i, j = 1 : n$, sodass man damit direkt die 2-Norm berechnen kann.

Algorithmus 4.11

```
1: for  $j$  in  $1 : n$  do
2:   for  $i$  in  $1 : n$  do
3:     Berechne  $Y = L_f(A, e_i e_j^T)$ 
4:      $\mathcal{K}(:, (j-1)n+1) = \text{vec}(Y)$ 
5:   end for
6: end for
7:  $\text{cond}(f, A) = \|\mathcal{K}\|_2 \|A\|_F / \|f(A)\|_F$ 
```

Dies ist jedoch für grosse n sehr teuer, da die Aufstellung von \mathcal{K} $O(n^5)$ Flops benötigt. Des Weiteren braucht dieser Algorithmus ziemlich viel Speicherplatz, sodass es sinnvoller ist die Exponentialkonditionszahl zu schätzen anstatt exakt zu berechnen - denn schliesslich sind wir an der Grössenordnung der Konditionszahl interessiert.

4.3.2 Schätzung der Frobenius-Norm

Um Probleme mit dem Speicherplatz zu umgehen und die Berechnung zu beschleunigen, möchten wir nun die Frobeniusnorm von $L_f(A)$ schätzen, respektive die 2-Norm von $\mathcal{K}(A)$. Hierzu ist es angebracht, die sogenannte Potenzmethode zu verwenden.

Für ein gegebenes $B \in \mathbb{C}^{n \times n}$ berechnet der folgende Algorithmus eine Schätzung $\gamma \leq \|B\|_2$, wobei die Potenzmethode auf B^*B angewendet wird.

Algorithmus 4.12 Potenzmethode

- 1: Wähle einen beliebigen Startvektor $z_0 \in \mathbb{C}$, jedoch nicht den Nullvektor.
- 2: **for** $k = 0 : \infty$ **do**
- 3: $\tilde{w}_{k+1} = Bz_k$
- 4: $w_{k+1} = \tilde{w}_{k+1} / \|\tilde{w}_{k+1}\|_2$
- 5: $\tilde{z}_{k+1} = B^*w_{k+1}$
- 6: $z_{k+1} = \tilde{z}_{k+1} / \|\tilde{z}_{k+1}\|_2$
- 7: $\gamma_{k+1} = \|z_{k+1}\|_2 / \|w_{k+1}\|_2$
- 8: **if** γ konvergiert **then**
- 9: $\gamma = \gamma_{k+1}$
- 10: Quit
- 11: **end if**
- 12: **end for**

Wir gehen nicht genauer auf diese Methode und deren Konvergenzanalyse ein und verweisen auf anderweitige Literatur, wie zum Beispiel [7], [17] oder sonstige Quellen.

Nun möchten wir diesen Algorithmus auf $B = \mathcal{K}(A)$ anwenden, um $\|L_f(A)\|_F$ zu erhalten. Natürlich wollen wir $\mathcal{K}(A)$ nicht direkt verwenden, aber wir können nun die Gleichung (8) ausnützen. Denn zum Beispiel folgt mit Zeile 3 im Algorithmus 4.12:

$$Bz_k = \mathcal{K}(A)z_k = \text{vec}(L_f(A, Z_k))$$

wobei Z_k eine $n \times n$ -Matrix beschreibt, in die der Vektor z_k der Länge n^2 spaltenweise eingeschrieben wurde. Somit können wir den ganzen Algorithmus nur mit $L_f(A, \cdot)$ und $L_f^*(A, \cdot)$, die Adjungierte von $L_f(A, \cdot)$, ausdrücken und brauchen die Matrix $\mathcal{K}(A)$ gar nicht zu verwenden. Die Adjungierte ist bezüglich dem Produkt $\langle X, Y \rangle = \text{tr}(Y^*)$ definiert, wobei für $A \in \mathbb{R}^{n \times n}$ und $f : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{n \times n}$ die Adjungierte durch $L_f^*(A, E) = L_f^*(A^T, E)$ und im Komplexen durch $L_f^*(A, E) = L_{\bar{f}}(A^*, E)$ gegeben ist, mit $\bar{f}(z) := \overline{f(\bar{z})}$.

Der folgende Algorithmus schätzt für ein gegebenes $A \in \mathbb{C}^{n \times n}$ ein $\gamma \leq \|L_f(A)\|_F$.

Algorithmus 4.13

```

1: Wähle eine beliebige Startmatrix  $Z_0 \in \mathbb{C}^{n \times n}$ , jedoch nicht die Nullmatrix.
2: for  $k = 0 : \infty$  do
3:    $W_{k+1} = L_f(A, Z_k)$ 
4:    $Z_{k+1} = L_f^*(A, W_{k+1})$ 
5:    $\gamma_{k+1} = \|Z_{k+1}\|_F / \|W_{k+1}\|_F$ 
6:   if  $\gamma_{k+1}$  konvergiert then
7:      $\gamma = \gamma_{k+1}$ 
8:     Quit
9:   end if
10: end for

```

Als Startmatrix eignet sich eine Zufallsmatrix. Ein mögliches Konvergenzkriterium in den Algorithmen 4.12 und 4.13 ist

$$k > \text{it}_{\max} \text{ or } |\gamma_{k+1} - \gamma_k| \leq \text{tol}.$$

Hier ist it_{\max} die maximale zugelassene Anzahl Iterationen und tol ist eine Konvergenztoleranz. Da wir lediglich an der Größenordnung interessiert sind, sind $\text{tol} = 10^{-1}$ oder 10^{-2} angemessen. Da jedoch die lineare Konvergenz beliebig langsam sein kann, ist es sehr schwer einen verlässlichen Konvergenztest zu konstruieren.

4.3.3 Schätzung der 1-Norm

Bei der Schätzung der 1-Norm gehen wir analog wie bei der Frobenius-Norm vor. Zuerst führen wir einen Schätzungsalgorithmus ein, den wir dann für unser Problem anpassen. Der folgende Algorithmus ist der sogenannte „LAPACK“ Matrixnormschätzer. Für gegebenes $B \in \mathbb{C}^{n \times n}$ berechnet der Algorithmus ein γ und ein $v = Bw$, sodass $\gamma \leq \|B\|_1$ ist mit $\|v\|_1 / \|w\|_1 = \gamma$, wobei w nicht ausgegeben wird. Für $z \in \mathbb{C}$ ist $\text{sign}(z) = z/|z|$ bei $z \neq 0$ und $\text{sign}(0) = 1$.

Algorithmus 4.14 LAPACK Matrixnormschätzer

```

1:  $v = B(n^{-1}e)$ 
2:  $\gamma = \|v\|_1$ 
3:  $\zeta = \text{sign}(v)$ 
4:  $m = B^*\zeta$ 
5:  $k = 2$ 
6: repeat
7:    $j = \min\{i : |m_i| = \|m\|_\infty\}$ 
8:    $v = Be_j$ 
9:    $\bar{\gamma} = \gamma$ 
10:   $\gamma = \|v\|_1$ 
11:  if (( $A$  ist reell und  $\text{sign}(v) = \pm\zeta$ ) oder  $\gamma \leq \bar{\gamma}$ ) then
12:    Gehe zu Ln 18

```

```

13: end if
14:  $\zeta = \text{sign}(v)$ 
15:  $m = B^* \zeta$ 
16:  $k = k + 1$ 
17: until  $\|m\|_\infty = x_j$  oder  $k > 5$ 
18:  $m_i = (-1)^{i+1} (1 + \frac{i-1}{n-1})$ , für  $i = 1 : n$ 
19:  $m = Bm$ 
20: if  $(2\|m\|_1/(3n) > \gamma)$  then
21:  $v = m$ 
22:  $\gamma = 2\|m\|_1/(3n)$ 
23: end if

```

Hier ist e ein Vektor der Länge n mit lauter Einsen und die Funktion sign wird komponentenweise angewendet. Der Algorithmus ist die Basis aller Konditionszahlsschätzer in „LAPACK“ und wird auch in der „MATLAB“-Funktion `rcond` verwendet. Er wurde von Higham [8] entwickelt und basiert auf der p -Norm-Potenzmethode von Boyd [2]. Für weitere Details verweisen wir auf folgende Quellen von Higham [8], [9] und [11].

Mit typischerweise 4-5 Matrix-Vektor-Produkten produziert der Algorithmus oft exakte Schätzungen, also $\gamma = \|B\|_1$. Jedoch kann er beliebig schlechte Schätzungen machen, vor allem bei speziell konstruierten Gegenbeispielen - aber er schätzt meist korrekt innerhalb vom Faktor 3.

Analog zum letzten Kapitel möchten wir nun diesen Algorithmus auf $\mathcal{K}(A)$ anwenden, ohne dieses konkret zu verwenden. Hierzu noch folgendes Theorem, das die Relation zwischen $L_f(A)$ und $\mathcal{K}(A)$ bezüglich der 1-Norm aufzeigt:

Satz 4.15

Für $A \in \mathbb{C}^{n \times n}$ und eine beliebige Funktion f , gilt:

$$\frac{\|L_f(A)\|_1}{n} \leq \|\mathcal{K}(A)\|_1 \leq n\|L_f(A)\|_1$$

Beweis

Für $E \in \mathbb{C}^{n \times n}$ haben wir $\|E\|_1 \leq \|\text{vec}(E)\|_1 \leq n\|E\|_1$, wobei die Gleichheit links erfüllt ist, wenn $E = ee_1^T$ ist und rechts, wenn $E = ee^T$ ist. Somit gilt mit der Gleichung (8):

$$\frac{1}{n} \frac{\|L_f(A, E)\|_1}{\|E\|_1} \leq \frac{\|\mathcal{K}(A) \text{vec}(E)\|_1}{\|\text{vec}(E)\|_1} \leq n \frac{\|L_f(A, E)\|_1}{\|E\|_1}$$

Durch das Maximieren über alle E erhält man das Resultat. □

Nun wenden wir den Algorithmus 4.14 auf $\mathcal{K}(A)$ an, um $\|L_f(A)\|_1$ zu schätzen.

Der folgende Algorithmus berechnet mit einem gegebenem $A \in \mathbb{C}^{n \times n}$ eine Schätzung γ von $n\|L_f(A)\|_1$. Genauer gesagt gilt $\gamma \leq \|\mathcal{K}(A)\|_1$, wobei $\|\mathcal{K}(A)\|_1 \in [n^{-1}\|L_f(X)\|_1, n\|L_f(X)\|_1]$ ist.

Um die Notation einfach zu halten, führen wir folgende Funktionen für eine Matrix $X \in \mathbb{C}^{n \times n}$ ein:

- $|X|$: komponentenweiser Betrag, d.h. $\{|x_{ij}|\}$
- $\max(X)$: Komponentenweises Maximum, d.h. $\max_{i,j}\{x_{ij}\}$
- $\sum(X)$: Summe aller Komponenten, d.h. $\sum_{i,j} x_{ij}$
- $\text{sign}(X)$: Komponentenweise Signumfunktion, d.h. $\{\text{sign}(x_{ij})\}$
- $E(n)$: $n \times n$ Matrix mit lauter Einsen als Einträge

Somit gilt für uns:

Algorithmus 4.16

```

1:  $V = L_f(A, \frac{1}{n} E(n))$ 
2:  $\gamma = \sum(|V|)$ 
3:  $\zeta = \text{sign}(V)$ 
4:  $M = L_f^*(A, \zeta)$ 
5:  $k = 2$ 
6: repeat
7:    $(i, j) = \min\{(i, j) : |m_{ij}| = \max(|M|)\}$ 
8:    $V = L_f(A, E_{ij})$ 
9:    $\bar{\gamma} = \gamma$ 
10:   $\gamma = \sum(|V|)$ 
11:  if (( $A$  ist reell und  $\text{sign}(V) = \pm\zeta$ ) oder  $\gamma \leq \bar{\gamma}$ ) then
12:    Gehe zu Ln 18
13:  end if
14:   $\zeta = \text{sign}(V)$ 
15:   $M = L_f^*(A, \zeta)$ 
16:   $k = k + 1$ 
17: until ( $\max(|M|) = m_{ij}$  oder  $k > 5$ )
18:  $m_{ij} = (-1)^{(i+(j-1)n+1)}(1 + \frac{i+(j-1)n-1}{n^2-1})$ , für  $i, j = 1 : n$ 
19:  $M = L_f(A, M)$ 
20: if ( $2 \sum(|M|)/(3n^2) > \gamma$ ) then
21:    $\gamma = 2 \sum(|M|)/(3n^2)$ 
22: end if

```

Vorteile des Algorithmus 4.16 im Gegensatz zum Algorithmus 4.13 ist einerseits, dass die Startmatrix gegeben ist und andererseits, dass der Konvergenztest vermieden werden kann. Weiter ist die 1-Norm auch leichter zugänglich und die mehr voraussehbare Anzahl von Iterationen sprechen ebenfalls für diesen Algorithmus.

5 Matrixlogarithmus

5.1 Einführung

Der englische Mathematiker Henry Briggs benützte bereits im 17. Jahrhundert eine geschickte Methode, um den Logarithmus einer Zahl grob abzuschätzen. Hierzu verwendete er das Logarithmusgesetz $\log(ab) = \log(a) + \log(b)$ für $a, b \in (0, \infty)$, respektive $\log(a) = 2 \log(a^{1/2})$ und durch wiederholtes anwenden erhält man $2^k \log a^{(1/2)^k}$. Des Weiteren ist $\log(1+x) \approx x$ für kleine x , sodass man zusammengefasst $\log a \approx 2^k \cdot (a^{(1/2)^k} - 1)$ als Approximation verwenden kann.

Für den Matrixlogarithmus ist vor allem die Gleichung

$$\log(A) = 2^k \log(A^{(1/2)^k})$$

sehr von Nutzen.

Wir weisen jedoch noch darauf hin, dass Analog zum Satz 3.1 auch beim Matrixlogarithmus nicht immer alle Gesetze der skalaren Funktion gelten:

Satz 5.1

Sei $B, C \in \mathbb{C}^{n \times n}$, wobei B, C als auch $A = BC$ keine Eigenwerte in $(-\infty, 0]$ haben. Wenn $BC = CB$, dann gilt $\log(A) = \log(B) + \log(C)$.

Der Beweis folgt direkt aus der Potenzreihendarstellung des Logarithmus als auch der Kommutativität.

Ähnlich wie bei der Exponentialfunktion möchten wir k so wählen, sodass wir $\log(A^{(1/2)^k})$ möglichst gut approximieren können, was wir ebenfalls mit der Padé-Approximation tun werden. Hierzu sollte $A^{(1/2)^k}$ möglichst nahe bei der Identität sein, was für genug grosse k gegeben ist, da $\lim_{k \rightarrow \infty} A^{(1/2)^k} = I$ gilt.

Der sogenannte „Inverse Scaling & Squaring“ Algorithmus wird folgende Struktur aufweisen:

1. Die Norm von $A^{1/2^k} - I = B$ genügend klein skalieren (*Scaling*)
2. $\log(B + I)$ approximieren (Padé-Approximation)
3. $\log(A) = 2^k \log(B + I)$ berechnen (*Squaring*)

Im Vergleich zur Struktur der „Scaling & Squaring“-Methode für die Exponentialfunktion ist es offensichtlich, warum diese Methode „Inverse Scaling & Squaring“ genannt wird.

Im nächsten Abschnitt werden wir herleiten, wie wir die Wurzel einer Matrix berechnen können, was wir für den ersten Schritt benötigen. Hierfür ist es von Vorteil die Matrix A in die Schurform zu zerlegen. Diese Zerlegung ist ein fundamentales Ergebnis aus der Linearen Algebra, welche wir hier ohne Beweis angeben:

Satz 5.2

Sei $A \in \mathbb{C}^{n \times n}$, dann hat A eine Schur-Darstellung als obere Dreiecksmatrix $T = Q^* A Q$, wobei Q eine unitäre Matrix ist und die Eigenwerte von A in der Diagonale von T stehen.

Diese Abschnitte basieren hauptsächlich auf Kapitel 6 und 11 in Nicholas J. Higham's Buch „Functions of Matrices“ [12].

5.2 Exkurs: Die Wurzel einer Matrix

Um den Algorithmus der Logarithmusfunktion herzuleiten müssen wir zuerst die Wurzel einer Matrix berechnen können. Hierfür möchten wir einen Algorithmus herleiten, der auf der Schurzerlegung basiert. Sei $A \in \mathbb{C}^{n \times n}$ mit der Zerlegung $A = Q T Q^*$, wobei Q unitär und T eine obere Dreiecksmatrix ist. Somit ist $A^{1/2} = Q T^{1/2} Q^*$, sodass wir lediglich die Wurzel U von T berechnen müssen, was wiederum eine Dreiecksmatrix ist. Schreibt man $U^2 = T$ koeffizientenweise auf, so erhält man die (i, i) - und (i, j) -Elemente für $i \neq j$ direkt durch die folgenden Gleichungen:

$$\begin{aligned} u_{ii}^2 &= t_{ii} \\ (u_{ii} + u_{jj})u_{ij} &= t_{ij} - \sum_{k=i+1}^{j-1} u_{ik}u_{kj} \end{aligned} \quad (9)$$

Somit können wir zuerst die diagonalen Elemente von U berechnen und dann die Gleichungen (9) für die u_{ij} lösen, wobei wir entweder superdiagonal oder spaltenweise vorgehen. Dieser Prozess kann nicht abbrechen, weil $0 = u_{ii} + u_{jj} = (t_{ii})^2 + (t_{jj})^2$ nicht möglich ist, da $t_{ii} \neq 0$. Wir erhalten daher unseren ersten Algorithmus:

Algorithmus 5.3

- 1: Berechne die komplexe Schurzerlegung $A = Q T Q^*$.
- 2: $u_{ii} = \sqrt{t_{ii}}$, $i = 1 : n$
- 3: **for** $j = 2 : n$ **do**
- 4: **for** $i = j - 1 : -1 : 1$ **do**
- 5:

$$u_{ij} = \frac{t_{ij} - \sum_{k=i+1}^{j-1} u_{ik}u_{kj}}{u_{ii} + u_{jj}}$$

- 6: **end for**
- 7: **end for**
- 8: $X = Q U Q^*$

Die Kosten belaufen sich auf $25n^3$ Flops für die Schurzerlegung, $n^3/3$ um U und $3n^2$ um X zu formen, also $28\frac{1}{3}n^3$ Flops total.

Wenn A und die Eigenwerte reell sind, so können wir die komplexe Arithmetik weglassen und den Algorithmus 5.4 ebenfalls verwenden. Ist A reell und sind einige Eigenwerte komplex, so ist es nicht sinnvoll komplexe Arithmetik zu verwenden, da diese viel teurer ist. In diesem Falle können wir die reelle Schurzerlegung verwenden. Diese hat ebenfalls die Form $A = Q R Q^T$, wobei Q eine orthogonale Matrix und R eine obere Block-Dreiecksmatrix ist,

mit 1×1 Blöcken bei reellen Eigenwerten und mit 2×2 Blöcken bei komplexen Eigenwerten auf der Diagonale. Die komplexen Eigenwerte treten dabei jeweils paarweise komplex konjugiert auf und werden durch den 2×2 Block repräsentiert. Analog zum komplexen Fall können wir nun $A^{1/2} = QR^{1/2}Q^T$ schreiben, wobei $U = R^{1/2}$ ebenfalls eine obere Block-Dreiecksmatrix mit derselben Blockstruktur ist. Schreibt man nun die Gleichung $U^2 = R$ blockweise auf, so erhält man analoge Gleichungen:

$$U_{ii}^2 = R_{ii}$$

$$U_{ii}U_{ij} + U_{ij}U_{jj} = R_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj} \quad (10)$$

Auch hier gilt, dass wenn wir die U_{ii} berechnet haben, so können wir die restlichen Blöcke U_{ij} nach demselben Muster direkt berechnen. Die Gleichung (10) ist die sogenannte Sylvester-Gleichung, welche als Bedingung hat, dass U_{ii} und $-U_{jj}$ nicht die gleichen Eigenwerte haben, was bei einer nichtsingulären Matrix garantiert ist.

Ist weder U_{ii} noch U_{jj} ein Skalar, so können wir die Gleichung wie folgt umschreiben:

$$(I \otimes U_{ii} + U_{jj}^T \otimes I) \text{vec}(U_{ij}) = \text{vec} \left(R_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj} \right)$$

Somit reduziert sich die Matrixgleichung auf ein lineares Gleichungssystem.

Es stellt sich nun die Frage, wie man $R_{ii}^{1/2}$ berechnet, wenn R_{ii} ein 2×2 Block ist. Hierzu berechne man die Eigenwerte $\lambda = \theta + i\mu$ von R :

$$R = \begin{pmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{pmatrix}$$

Mit der Determinante von $R - \lambda I$ erhält man direkt:

$$\theta = \frac{1}{2}(r_{11} + r_{22}) \quad \mu = \frac{1}{2} \left(-(r_{11} - r_{22})^2 - 4r_{21}r_{12} \right)^{1/2}.$$

Die Wurzel $\alpha + i\beta$ von $\lambda = \theta + i\mu$ können wir nun wie folgt berechnen:

$$\begin{cases} \alpha = ((\theta + (\theta^2 + \mu^2)^{1/2})/2)^{1/2}, & \beta = \mu/(2\alpha) & \text{wenn } \theta \geq 0 \\ \beta = ((|\theta| + (\theta^2 + \mu^2)^{1/2})/2)^{1/2}, & \alpha = \mu/(2\beta) & \text{wenn } \theta < 0 \end{cases}$$

Dies führt zu folgender Darstellung von $U_{ii} = R_{ii}^{1/2}$:

$$U_{ii} = \pm \left(\alpha I + \frac{1}{2\alpha} (R_{ii} - \theta I) \right) \quad (11)$$

$$= \begin{pmatrix} \alpha + \frac{1}{4\alpha}(r_{11} - r_{22}) & \frac{1}{2\alpha}r_{12} \\ \frac{1}{2\alpha}r_{21} & \alpha - \frac{1}{4\alpha}(r_{11} - r_{22}) \end{pmatrix}$$

Zusammengefasst erhalten wir folgenden Algorithmus:

Algorithmus 5.4

- 1: Berechne die reelle Schurzerlegung $A = QRQ^T$,
wobei R eine $m \times m$ Blockmatrix ist.
- 2: Berechne $U_{ii} = R_{ii}^{1/2}$ für $i = 1 : m$,
wobei die Gleichung (11) für 2×2 Blöcke verwendet wird.
- 3: **for** $j = 2 : m$ **do**
- 4: **for** $i = j - 1 : -1 : 1$ **do**
- 5: Löse $U_{ii}U_{ij} + U_{ij}U_{jj} = R_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj}$ für U_{ij} .
- 6: **end for**
- 7: **end for**
- 8: $X = QUQ^T$

Die Kosten belaufen sich auch hier auf $28\frac{1}{3}n^3$ Flops.

Auf die Fehleranalyse möchten wir nicht genauer eingehen, da dieser Algorithmus eher ein Mittel zum Zweck ist. Jedoch möchten wir trotzdem die Fehlerabschätzung angeben, aber verweisen interessierte Leser auf [12].

Die Standardfehleranalyse führt zum folgenden Ergebnis:

$$\frac{\|A - \hat{X}^2\|_F}{\|A\|_F} \leq \tilde{\gamma}_n^3 \frac{\|\hat{X}\|_F^2}{\|A\|_F}$$

wobei $\hat{X}^2 = A + \Delta A$ und $\tilde{\gamma}$ wie im Unterabschnitt 2.2.4 definiert ist.

5.3 Konstruktion der „Inverse Scaling & Squaring“-Methode

Wie im Abschnitt 5.2 erwähnt, müssen wir für die Wurzelfunktion jeweils die Matrix in die Schurform zerlegen. Natürlich ist es effizienter, wenn wir zu Beginn die Schurzerlegung vornehmen und die Padé-Approximation auf die obere Dreiecksmatrix $T \in \mathbb{C}^{n \times n}$ anwenden.

Es stellt sich offensichtlich die Frage, wie oft wir die Wurzel von T berechnen und welchen Grad der Padé-Approximation wir wählen sollen, natürlich mit dem Ziel einen Kompromiss zwischen minimalen Kosten und maximaler Genauigkeit zu finden.

Weiter müssen wir auch festlegen, wie wir das Polynom auswerten. Natürlich steht wiederum die Paterson-Stockmeyer-Methode zur Auswahl, jedoch hat sich bei einer detaillierten Analyse von Higham in [10] gezeigt, dass die Auswertung über die Partialbruchzerlegung die beste Methode ist, um r_m zu evaluieren. Wir halten kurz fest, wie die Fehlermatrix durch die Padé-Approximation beschränkt ist:

Satz 5.5 (Kenney and Laub)

Sei $X \in \mathbb{C}^{n \times n}$ mit $\|X\| < 1$ und $\|\cdot\|$ eine beliebige konsistente Matrixnorm, dann gilt:

$$\|r_m(A) - \log(I + X)\| \leq |r_m(-\|X\|) - \log(1 - \|X\|)|$$

Folgende zwei Fakten sind der Schlüssel für unsere weiteren Überlegungen:

- Die Wurzel einer oberen Dreiecksmatrix U können wir in $n^3/3$ Flops berechnen.
- $r_m(T)$ können wir über die Partialbruchzerlegung berechnen und benötigen hierfür $mn^3/3$ Flops.

Es lohnt sich also eine extra Wurzel zu ziehen, wenn sich der Approximationsgrad mindestens um eins verringert.

m	1	2	3	4	5	6	7	8	9
θ_m	1.10e-5	1.82e-3	1.62e-2	5.39e-2	1.14e-1	1.87e-1	2.64e-1	3.40e-1	4.11e-1
$\kappa(q_m)$	1.00e0	1.00e0	1.05e0	1.24e0	1.77e0	3.09e0	6.53e0	1.63e1	4.62e1
ϕ_m	1.00e0	1.00e0	1.03e0	1.11e0	1.24e0	1.44e0	1.69e0	2.00e0	2.36e0

m	10	11	12	13	14	15	16	32	64
θ_m	4.75e-1	5.31e-1	5.81e-1	6.24e-1	6.62e-1	6.95e-1	7.24e-1	9.17e-1	9.78e-1
$\kappa(q_m)$	1.47e2	5.07e2	1.88e3	7.39e3	3.05e4	1.31e5	5.80e5	> 1e16	> 1e16
ϕ_m	2.76e0	3.21e0	3.71e0	4.25e0	4.84e0	5.47e0	6.14e0	2.27e1	8.85e1

Man betrachte die obenstehende Tabelle, welche für den Grad m im Bereich 1 zu 64 die folgenden drei Werte angibt:

- θ_m : Das grösste $\|X\|$, sodass die Abschätzung aus Satz 5.5 kleiner gleich als die Maschinengenauigkeit ist, d.h. $\|r_m(X) - \log(I + X)\| \leq u = 2^{-53} \approx 1.1 \times 10^{-16}$.
- $\kappa(q_m)$: Dies bezeichnet die Abschätzung des folgenden Lemmas mit $\|X\| = \theta_m$:

Lemma 5.6 (Kenney and Laub)

Sei $X \in \mathbb{C}^{n \times n}$ mit $\|X\| < 1$ und $\|\cdot\|$ eine beliebige konsistente Matrixnorm. Weiter ist q_m der Nenner der Padé-Approximation $r_m = p_m/q_m$, dann gilt:

$$\kappa(q_m(X)) \leq \frac{q_m(\|X\|)}{q_m(-\|X\|)}.$$

- ϕ_m : Dies bezeichnet die folgende Abschätzung für $\|X\| = \theta_m$.

$$\phi_m = \max_j \kappa(I + \beta_j^{(m)} X) \leq \max_j \frac{1 + |\beta_j^{(m)}| \|X\|}{1 - |\beta_j^{(m)}| \|X\|}.$$

Man stellt sofort fest, dass $\kappa(q_m(X))$ sehr schnell wächst für $m \geq 10$, was vor allem andere Auswertungsmethoden betrifft wie zum Beispiel die Paterson-Stockmeyer-Methode. Weiter sehen wir, dass ϕ_m für alle m eine akkurate Auswertung durch die Partialbruchreihe zulässt.

Um die Daten der Tabelle genauer zu interpretieren, müssen wir wissen wie gross der Einfluss des Wurzelziehens auf T ist. Dies können wir unabhängig von der Dreiecksstruktur von T abschätzen. Da $(I - A^{(1/2)^{k+1}})(I - A^{(1/2)^{k+1}}) = I - A^{(1/2)^k}$ gilt und $A^{(1/2)^k} \rightarrow I$ für $k \rightarrow \infty$ folgt, können wir folgende approximative Gleichung verwenden:

$$\|I - A^{(1/2)^{k+1}}\| \approx \frac{1}{2}\|I - A^{(1/2)^k}\| \quad (12)$$

Daraus schliessen wir, dass wenn $A^{(1/2)^k}$ die Norm nahe bei 1 hat, dass ein weiteres Wurzelziehen die Distanz zur Identität etwa halbiert.

Kommen wir zurück zu unserer Tabelle aus der ersichtlich ist, dass $\theta_m < \theta_{m-2}$ wenn $m > 7$ ist. Daher macht es Sinn, die Wurzel zu ziehen bis $\|X\| \leq \theta_7$, da somit jede Wurzel die Kosten der Padé-Approximation reduziert.

Nun können wir bereits den Algorithmus beschreiben, wobei $A \in \mathbb{C}^{n \times n}$ keine Eigenwerte in $[0, -\infty)$ hat.

Algorithmus 5.7

- 1: Berechne die (reelle oder komplexe) Schurzerlegung $A = QTQ^*$
- 2: $k = 0, p = 0$
- 3: **while** true **do**
- 4: $\tau = \|T - U\|_1$
- 5: **if** $\tau \leq \theta_7$ **then**
- 6: $p = p + 1$
- 7: $j_1 = \min\{i : \tau \leq \theta_i, \quad i = 3 : 7\}$
- 8: $j_2 = \min\{i : \tau/2 \leq \theta_i, \quad i = 3 : 7\}$
- 9: wenn $j_1 - j_2 \leq$ oder $p = 2, m = j_1$: GOTO Ln 14
- 10: **end if**
- 11: $T \leftarrow T^{1/2}$ (Algorithmus 5.4)
- 12: $k = k + 1$
- 13: **end while**
- 14: Berechne $U = r_m(T - I)$ unter der Benutzung der Partialbruchreihe.
- 15: $X = 2^k QUQ^*$

Die Kosten belaufen sich auf $25n^3$ Flops für die Schurzerlegung und $(k + m)n^3/3$ Flops um U und $3n^3$ Flops um X zu formen: also etwa $(30 + \frac{k}{3})n^3$ Flops total.

Im Falle, dass die Abschätzung (12) nicht gut ist, könnte der Algorithmus höchstens eine unnötige Wurzel ziehen, da der Test $p = 2$ bei Zeile 9 dies kontrolliert.

6 Schlusswort & Ausblick

Durch die untersuchten Algorithmen erhält man einen Einblick, wie komplex und knifflig die Konstruktion eines geschickten und effizienten Algorithmus sein kann. Der Vergleich der Exponentialfunktion von Ward und Higham zeigt einerseits auf, wie viel in einem bereits etablierten Algorithmus noch optimierbar war, andererseits die verschiedenen Ansichten über gewisse Methoden, wie zum Beispiel das Ausgleichen, wo wir den Standpunkt Pro Ausgleichen vertreten. Meiner Meinung nach sind gerade solche Beispiele immens motivierend um Forschung weiter voran zu treiben.

Die Logarithmusfunktion macht im Kontext der Matrixfunktionen ihrem Ruf als „Umkehr“-Funktion alle Ehre - sogar der Algorithmus wird „umgekehrt“. Die Genauigkeit der behandelten Algorithmen ist im Allgemeinen sehr gut und die Laufzeiten sind mit den heutigen leistungsstarken Computern auch für grössere Matrizen vertretbar.

Die Thematik der Matrixfunktionen endet natürlich nicht bei der Exponentialfunktion oder beim Logarithmus. Von der Signumfunktion über höhere Wurzeln, Polarzerlegung bis zu Cosinus- und Sinusfunktion, die Auswahl an Funktionen und deren interessanten numerischen Inhalt ist riesig. Interessierte Leser finden im Verzeichnis noch weiterführende Literatur, jedoch ist das Buch „Functions of Matrices“ von Nicholas J. Higham wohl das umfassendste Werk.

Die Numerik als relativ neue Kerndisziplin gewinnt immer mehr an Bedeutung und bot einen sehr spannenden Ausflug als auch eine motivierende Grundlage für eine Abschlussarbeit des Mathematik Studiums. An dieser Stelle möchte ich mich herzlich bei meinen Betreuern bedanken. Herr Prof. Dr. Daniel Kressner vom Seminar für Angewandte Mathematik, der mit seiner freundlichen und unkomplizierten Art mich von der mathematischen Perspektive optimal unterstützte und mir diese Arbeit erst ermöglichte. Herr Dr. Martin Mächler vom Seminar für Statistik und R-Project Entwicklungsteam, der mit viel Zeit und Geduld meine implementierten Algorithmen polierte und mir mit Tipps, Tricks und Korrekturen einiges über die Programmiersprache von R lehrte. Natürlich danke ich allen, die mich während dieser intensiven Zeit unterstützt haben, ganz speziell meiner Partnerin.

A Tests implementierter Algorithmen

Die folgenden Tests wurden auf einem Sony Vaio VGN-FE21H, Intel Core Duo T2300 (2x 1.66GHz) mit 1GB RAM auf Windows XP durchgeführt, wobei die R-Version 2.8.0 und die Matlab-Version 7.7.0.471 (R2008b) verwendet wurden.

Des Weiteren werden wir häufig Laufzeiten von R und Matlab vergleichen, möchten jedoch darauf hinweisen, dass hier grosse Unterschiede vorzufinden sind. Um dies genauer aufzuzeigen, haben wir jeweils eine 1000×1000 , 500×500 und eine 200×200 -Zufallsmatrix generiert und mit diesen jeweils 50 Matrixmultiplikationen ($A \cdot A$) und 50 Matrixinversionen (A^{-1}) durchgeführt. Folgende Laufzeiten wurden benötigt:

$n = 1000$	$A \cdot A$	A^{-1}	$n = 500$	$A \cdot A$	A^{-1}	$n = 200$	$A \cdot A$	A^{-1}
R	205.1	223.2	R	20.7	21.7	R	1.15	1.56
Mlab	49.2	64.3	Mlab	5.77	7.03	Mlab	0.37	0.59

Dieser Unterschied (im Groben etwa Faktor 3) geht auf das durch die Standardinstallation verwendete BLAS zurück. Im Matlab wird die „Math Kernel Library“ (kurz MKL) von Intel als BLAS verwendet, welche auf den Prozessor zugeschnitten ist. Bei R hingegen liegt das BLAS zugrunde, also die Standardsammlung der Routinen (siehe auch <http://www.netlib.org/blas/>).

A.1 Tests mit der Exponentialfunktion

A.1.1 Einzelne Beispiele

Folgende „exakten“ Funktionsergebnisse wurden mit Hilfe von Matlab und der Symbolic Toolbox auf eine Genauigkeit von hundert Stellen berechnet und dann mit den Berechnungen der R-Implementation verglichen.

Beispiel 1 Zuerst betrachten wir ein allgemeines Beispiel (nach [18]). Die Testmatrix sei:

$$A = \begin{pmatrix} 4 & 2 & 0 \\ 1 & 4 & 1 \\ 1 & 1 & 4 \end{pmatrix}$$

Diese Matrix ist nicht diagonalisierbar und das Minimalpolynom entspricht dem charakteristischen Polynom. Eigenwerte sind 3 mit Multiplizität 2 und 6, die Norm ist $\|A\|_1 = 7$. Die 1-Normen der Fehlermatrizen auf zwei Stellen nach dem Komma gerundet sind wie folgt:

$$\|e_{\text{Ward}}^A - e_{\text{exakt}}^A\|_1 = 3.13e-13$$

$$\|e_{\text{Higham}}^A - e_{\text{exakt}}^A\|_1 = 4.26e-13$$

$$\|e_{\text{Higham(bal)}}^A - e_{\text{exakt}}^A\|_1 = 4.26e-13$$

Der Algorithmus von Ward ist minim exakter. In diesem Beispiel scheint das Balancing keine Auswirkung zu haben. Man darf sagen, dass hier alle Algorithmen sehr gut abschneiden.

Beispiel 2 Dieses Beispiel (ebenfalls nach [18]) stellt die Algorithmen etwas mehr auf die Probe:

$$B = \begin{pmatrix} -131 & 19 & 18 \\ -390 & 56 & 54 \\ -387 & 57 & 52 \end{pmatrix}$$

Die Eigenwerte der Matrix sind -1 , -2 und -20 , die Norm ist $\|B\|_1 = 908$. Die 1-Normen der Fehlermatrizen auf zwei Stellen gerundet sind wie folgt:

$$\|e_{\text{Ward}}^B - e_{\text{exakt}}^B\|_1 = 5.00e-12$$

$$\|e_{\text{Higham}}^B - e_{\text{exakt}}^B\|_1 = 3.6e-12$$

$$\|e_{\text{Higham(bal)}}^B - e_{\text{exakt}}^B\|_1 = 7.03e-13$$

Wir sehen hier die erste Wirkung von dem Balancing, denn der Algorithmus von Higham ist damit einiges exakter. Aber auch ohne Vorverarbeitung schneidet Higham besser ab als der Algorithmus von Ward - dies ist auf die bessere Padé-Approximation zurückzuführen.

Beispiel 3 Diese Matrix wurde in Matlab per Zufallsgenerator erstellt und dann noch mit der Ähnlichkeitsabbildung durch die Diagonalmatrix $D = \text{diag}(1, 10^{-4}, 10^4, 10^4, 10^{-4})$ schlecht konditioniert, das heisst $C = D^{-1}ED$. Danach wurde C mit $\|C\|_1 = 177088903$ in R eingelesen (hier auf 7 Stellen gerundet):

$$C = \begin{pmatrix} 8.147237e-01 & 9.754040e-06 & 1.576131e+03 & 1.418863e+03 & 6.557407e-05 \\ 9.057919e+03 & 2.784982e-01 & 9.705928e+07 & 4.217613e+07 & 3.571168e-02 \\ 1.269868e-05 & 5.468815e-09 & 9.571669e-01 & 9.157355e-01 & 8.491293e-09 \\ 9.133759e-05 & 9.575068e-09 & 4.853756e-01 & 7.922073e-01 & 9.339932e-09 \\ 6.323592e+03 & 9.648885e-01 & 8.002805e+07 & 9.594924e+07 & 6.787352e-01 \end{pmatrix}$$

$$\|e_{\text{Ward}}^C - e_{\text{exakt}}^C\|_1 = 1.19e-7$$

$$\|e_{\text{Higham}}^C - e_{\text{exakt}}^C\|_1 = 10934631$$

$$\|e_{\text{Higham(bal)}}^C - e_{\text{exakt}}^C\|_1 = 2.98e-7$$

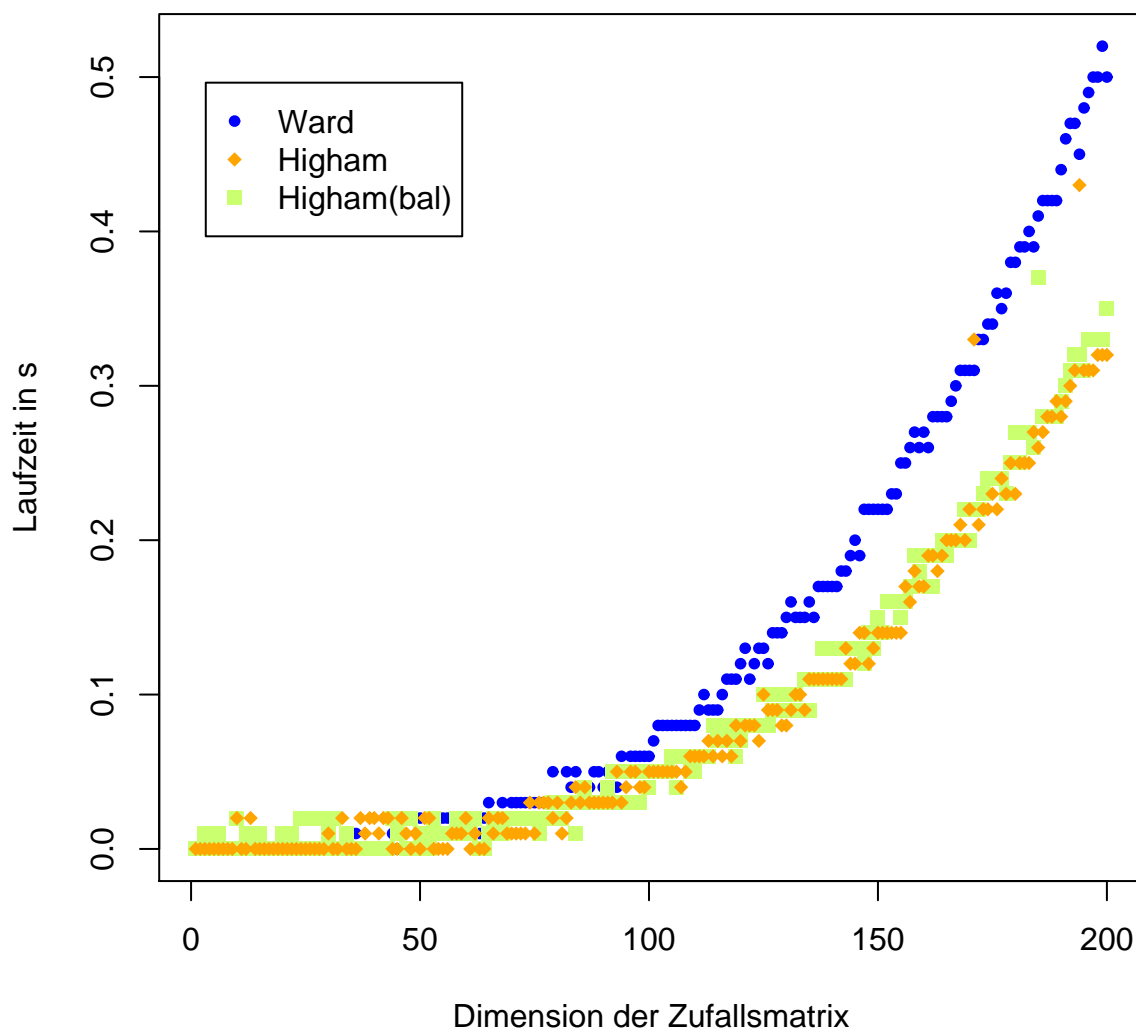
Wie zu erwarten bringt das Balancing bei dieser Matrix sehr viel. Die Algorithmen schneiden also mit der Vorverarbeitung etwa gleich gut ab.

A.1.2 Laufzeit mit randomisierten Matrizen

Wir bereits erwähnt, ist der Algorithmus von Higham schneller als der von Ward. Dies liegt hauptsächlich an der Polynom-Auswertung, die bei Higham effizienter gestaltet wurde - Higham benötigt maximal $6M$, Ward benötigt immer $7M$. Wir werden lediglich das quantitative Verhalten der Algorithmen anschauen, da die Analyse der Algorithmen bereits das Resultat verrät.

Wir haben 200 Zufallsmatrizen mit den Dimensionen von 1 bis 200 generiert, wobei die Einträge zwischen -1 und 1 gleichverteilt sind. In der folgenden Graphik sieht man die benötigten Laufzeiten der Algorithmen um $\exp(\cdot)$ der jeweiligen Matrix zu berechnen.

Laufzeit: Ward77 gegen Higham05



Wie erwartet ist der Algorithmus von Higham einiges schneller und das Balancing kann teilweise etwas mehr kosten, es fällt jedoch nicht ins Gewicht.

A.1.3 Matrizen vom Matrix Market

Die folgende Auswahl von Matrizen sind vom Matrix Market (<http://math.nist.gov/MatrixMarket/>):

BWM200.mtx	CDDE1.mtx	CDDE2.mtx	CDDE3.mtx	CDDE4.mtx
CDDE5.mtx	CDDE6.mtx	CK104.mtx	CK400.mtx	CK656.mtx
LOP163.mtx	OLM100.mtx	OLM1000.mtx	OLM500.mtx	PDE225.mtx
PDE900.mtx	PLAT362.mtx	PLSKZ362.mtx	QH768.mtx	QH882.mtx
RDB200.mtx	RDB200L.mtx	RDB450.mtx	RDB450L.mtx	RDB800L.mtx
RDB968.mtx	RW136.mtx	RW496.mtx	TOLS340.mtx	TOLS90.mtx
TUB100.mtx	TUB1000.mtx			

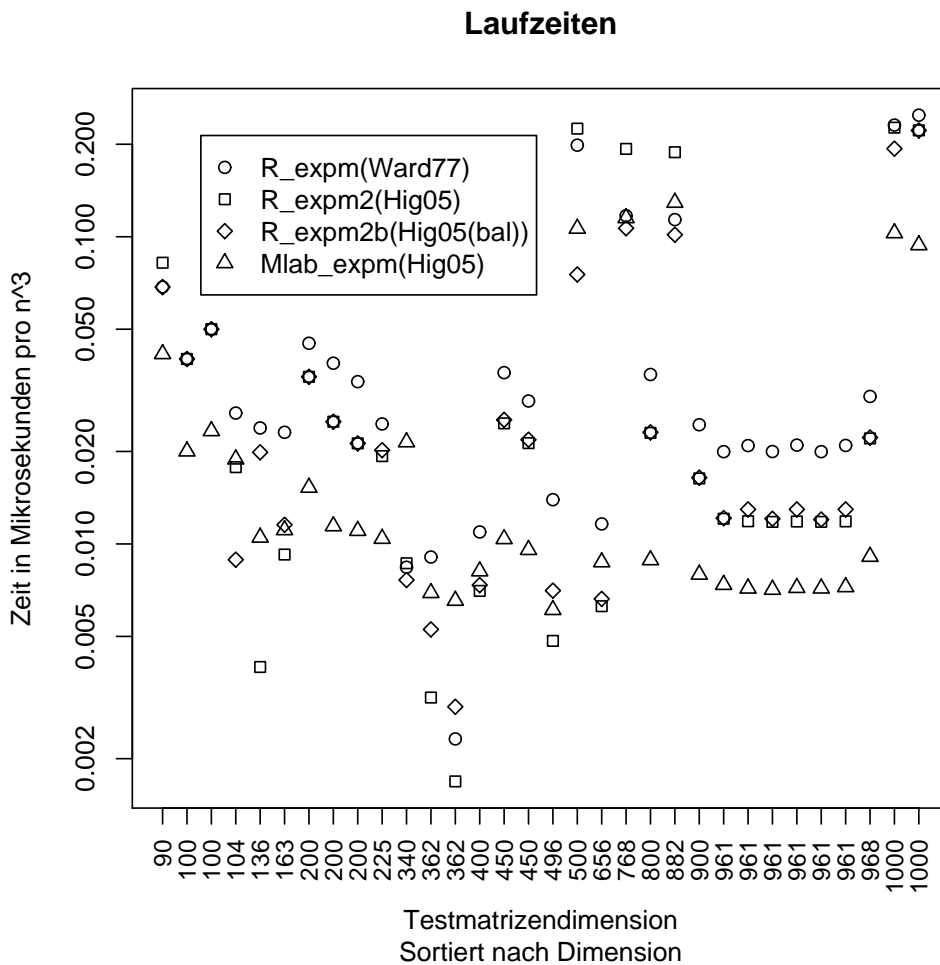
Einerseits wurden die Laufzeiten gemessen, andererseits die Grösse $\|e^{3A} - e^A e^A e^A\|_1 / \|e^{3A}\|_1$, welches ein Indikator für die Genauigkeit der Berechnung ist. Hier einige Beispiele des R Codes:

```
zeitexpm[i] <- system.time(expm(A)) [3]
absfehlexpm[i] <- norm(expm(3*A) - expm(A) %*% expm(A) %*% expm(A), "1")
expnorm[i] <- norm(expm(3*A), "1")
relfehl <- absfehlexpm/expnorm
```

Respektive in Matlab:

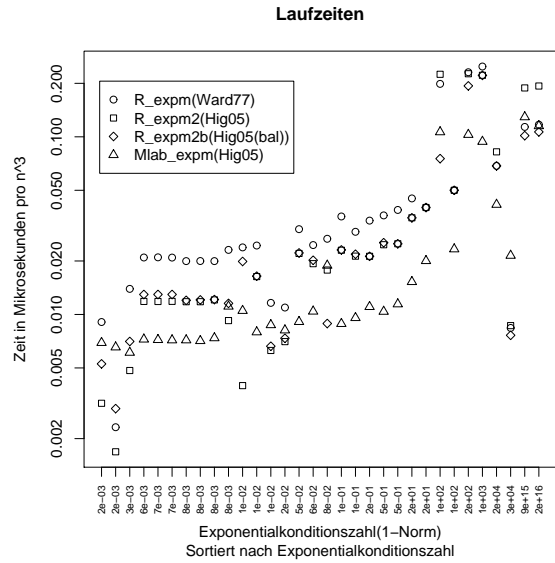
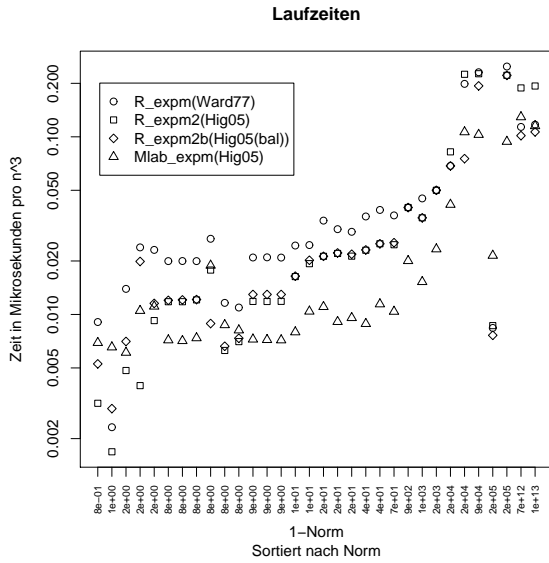
```
expnorm(i)= norm(expm(3*A),1)
normexpm(i)=norm(expm(3*A)-expm(A)*expm(A)*expm(A),1)
tic; expm(A);zeit(i)=toc;
```

Dies wurde realisiert für den Algorithmus von Ward [18] *R_expnm(Ward77)*, den Algorithmus von Higham auf Seite 19 (mit Ausgleichen *R_expnm(Hig05(bal))* und ohne *R_expnm(Hig05)*) und für die Matlab Funktion *Mlab_expnm(Hig05)*, welche ebenfalls Higham's Algorithmus in Matlab darstellt (seit V7 in Matlab) und ohne Ausgleichen ist.



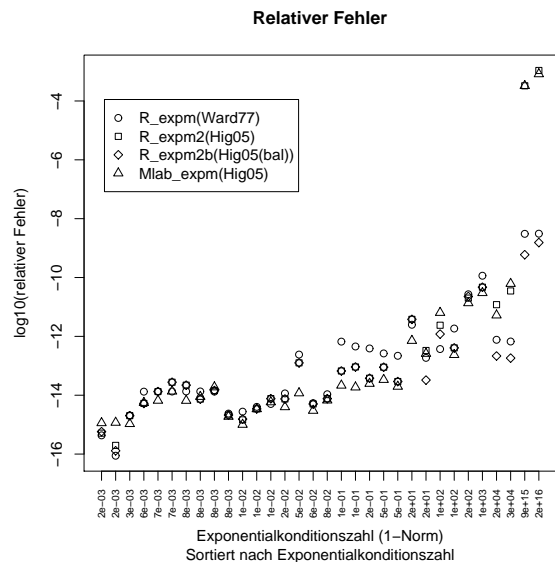
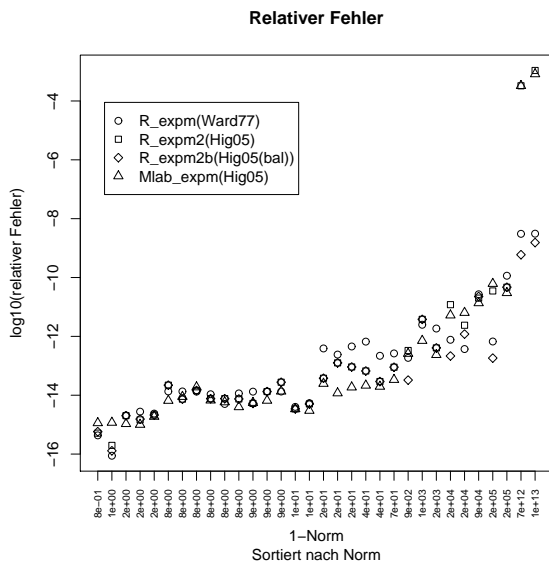
In der Grafik sehen wir die Laufzeiten in Mikrosekunden pro Dimension hoch drei, sodass wir den erwarteten linearen Zusammenhang sehen können. Es ist ersichtlich, dass wir einige Ausreisser haben (Matrizen mit der Dimension 500,768,882, 1000(1),1000(2)), welche auf sehr schlechte Konditionierung der Matrizen ($\kappa > 2000000$) und grosse Normen zurückzuführen ist. Bis auf wenige Ausnahmen ist normalerweise der Matlab-Algorithmus am schnellsten, dann Higham mit oder ohne Ausgleichen und schliesslich, als langsamste Berechnung, Ward's Algorithmus. Weiter ist erwähnenswert, dass das Ausgleichen bei einigen Fällen sogar sehr zeitsparend ist (Matrizen mit der Dimension 90,104,500,768,882,1000(1)) im Vergleich zum nicht Ausgleichen - vor allem bei schlecht konditionierten Matrizen. Das Ausgleichen zeigt sich lediglich bei einem Fall als signifikanter Nachteil für die Laufzeit (Matrix mit der Dimension 136).

In den nächsten beiden Grafiken findet man noch die Zeiten nach der Norm der Matrizen und der Exponentialkonditionszahl sortiert. Hier zeigt sich ebenfalls ein ähnliches, zu erwartendes Bild. Wir sehen, dass die Norm einen starken Einfluss auf die Laufzeit hat, wie zu erwarten ist wegen dem „Squaring“.



Als nächstes untersuchen wir die Genauigkeit. Hier haben wir im Graphen den Logarithmus der Grösse $\|e^{3A} - e^A e^A e^A\|_1 / \|e^{3A}\|_1$. Es ist ersichtlich, dass in vielen Fällen die Matlab-Berechnung am Besten ist. Verwunderlich ist jedoch, dass manchmal sogar der Algorithmus von Ward am genauesten ist. Dies würde sich bei der Matrix mit Norm $1e+00$ erklären lassen, wegen der kleinen Norm und der genaueren Padé-Approximation, jedoch bei der Matrix mit Norm $2e+04$ ist dies nicht klar begründbar.

Weiter sehen wir wiederum an einigen Beispielen den Vorteil des Ausgleichens (Matrizen mit Norm: $9e + 02, 2e + 04(1), 2e + 04(2), 2e + 05, 7e + 12, 1e + 13$), jedoch findet man kein Beispiel, wo sich dies stark negativ auswirkt. Ob man nach der Norm oder nach der Exponentialkonditionszahl sortiert, scheint hier keine grosse Rolle zu spielen.

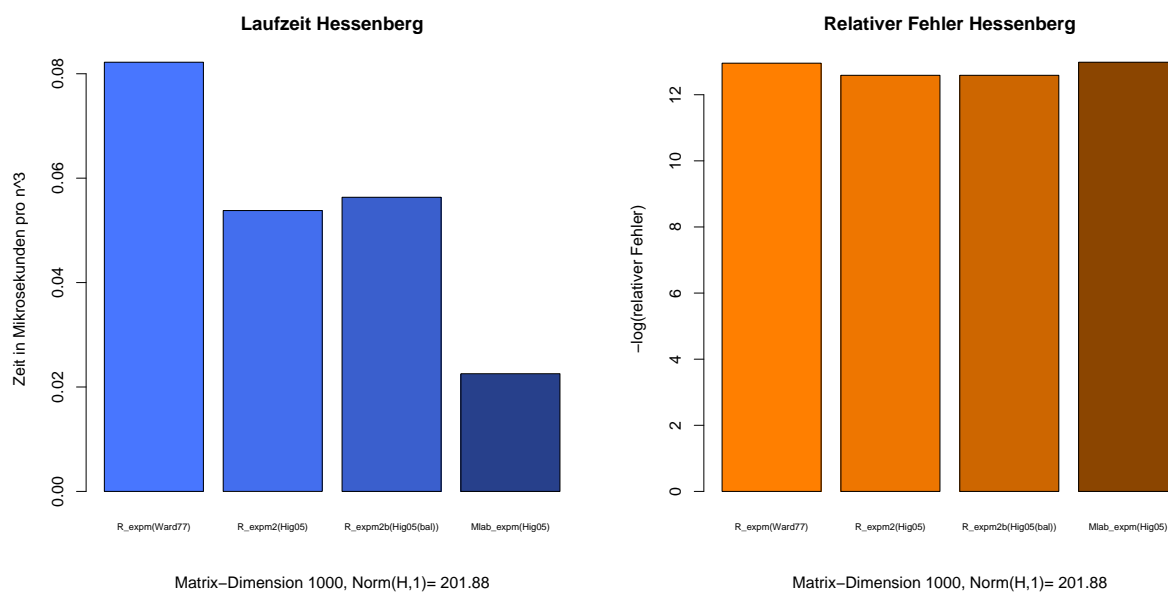


Zusammengefasst dürfen wir sagen, dass wir in der Testreihe kein Beispiel hatten, wo sich das Ausgleichen als signifikant verschlechternd herausstellte und somit Higham's Algo-

rithmus mit „Balancing“ sich als guter Algorithmus zeigt. Aber um dies etwas genauer zu analysieren, werden wir im Abschnitt A.1.5 dies noch etwas genauer untersuchen.

A.1.4 Hessenbergmatrix

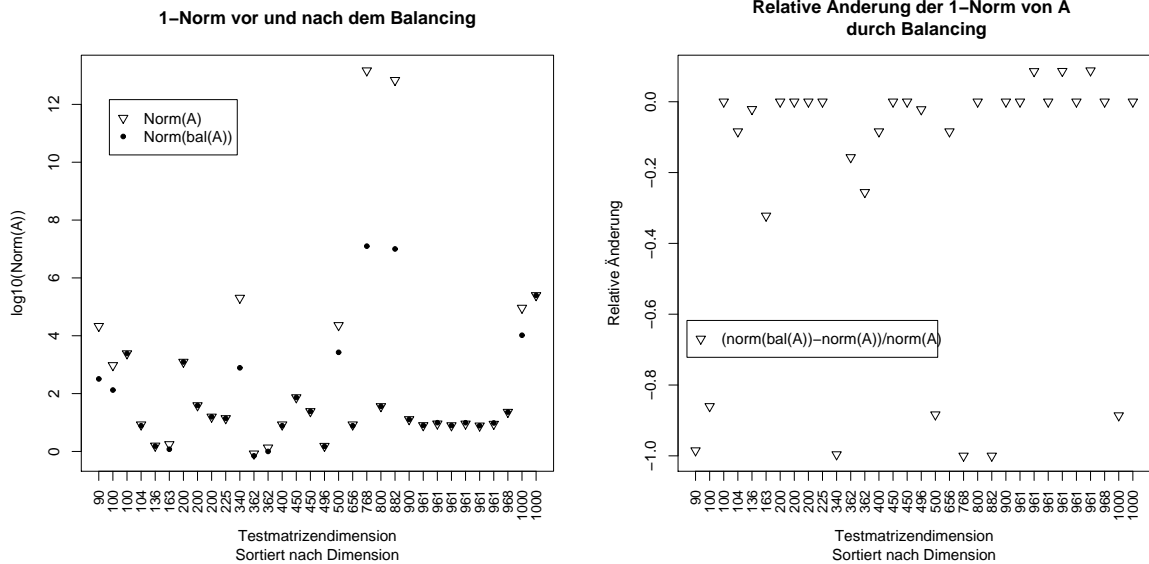
Wie bereits im Kapitel Vorverarbeitungen erwähnt, so kann das Ausgleichen schlecht für die Matrixnorm sein. Watkins [19] zeigt auf, dass dies insbesondere bei einer Hessenbergmatrix der Fall ist. Dies mag auch sein bei Eigenwertproblemen, jedoch scheint dies keine grossen Auswirkungen für die Exponentialfunktion zu haben. Für diesen Test haben wir eine 1000×1000 -Zufallsmatrix in Matlab generiert und diese in eine Hessenbergmatrix (Matlab: `hess()`) umgewandelt.



Es ist ersichtlich, dass das Ausgleichen etwas mehr Laufzeit kostet, aber keineswegs das Resultat verschlechtert. Man darf abschliessend sagen, dass unsere Tests keinen Nachteil des Ausgleichens aufgezeigt haben. Daher sind wir der Meinung, dass man dies im Allgemeinen verwenden sollte.

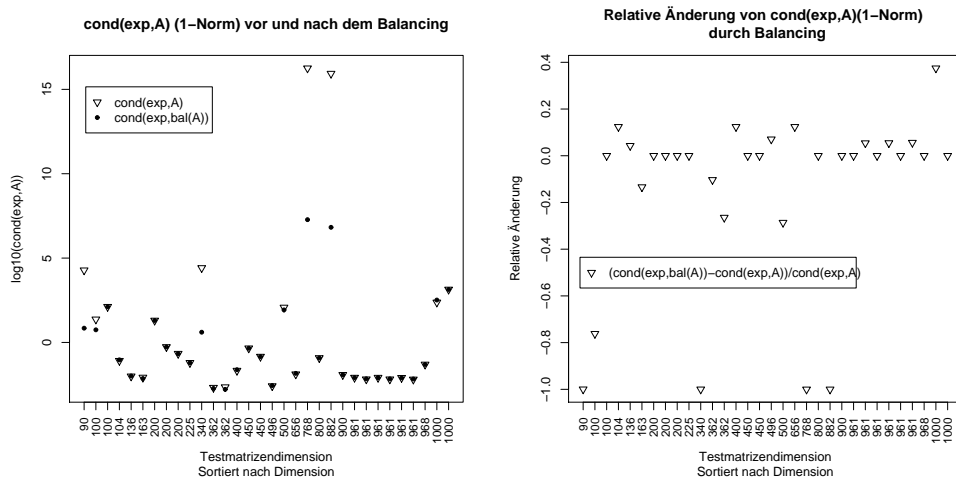
A.1.5 Balancing

Bis anhin haben wir festgestellt, dass das Balancing generell eher positive Auswirkungen hatte. Um dies genauer zu analysieren, haben wir noch weitere Tests gemacht. In den folgenden Graphen findet man die 1-Norm und die Exponentialkonditionszahl der „Matrix Market“-Matrizen, jeweils vor und nach dem Ausgleichen.



Bei der Norm sehen wir, dass das Balancing mehrheitlich die Norm stark verkleinert. Die einzigen Ausnahmen sind die Matrizen mit der Dimension 961(2), 961(4), 961(6), also die Matrizen „CDDE2“, „CDDE4“, „CDDE6“. Die relative Verschlechterung der Norm ist jedoch mit etwa -0.088 relativ klein.

Bei der Exponentialkonditionszahl haben wir auf den ersten Blick ein ähnliches Bild, jedoch hat hier das Balancing häufiger negative Auswirkungen. Bei den Matrizen mit der Dimension 104,136,400,496,656,961(2),961(4),961(6) haben wir eine relative Verschlechterung von etwa 0.124, was vertretbar ist. Dagegen ist die Matrix mit der Dimension 1000(1) ein Ausreisser mit 0.375.



Schliesslich betrachten wir noch die Hessembergmatrix aus dem vorherigen Abschnitt. Dies ergab folgende Resultate:

$$\|H\|_1 = 201.88 \quad \|\text{bal}(H)\|_1 = 199.55$$

Dies ist eine relative Verkleinerung der Norm von 0.011.

$$\text{cond}(\exp, H) = 0.3607 \quad \text{cond}(\exp, \text{bal}(H)) = 0.3185$$

Auch hier ergibt sich eine relative Verkleinerung der Exponentialkonditionszahl von 0.12

Als Fazit dürfen wir sagen, dass das Balancing mehrheitlich eher eine optimierende Wirkung hat und in wenigen Fällen eine Verschlechterung hervorruft, diese jedoch nicht signifikant ist. Wir empfehlen daher generell das Balancing zu verwenden, jedoch sollte man die Option haben es auszuschalten zu können.

A.2 Tests mit der Logarithmusfunktion

Bei den folgenden Tests haben wir zuerst die Ausgangsmatrizen in R und Matlab eingelesen, wobei es sich um dieselben Matrizen wie bei den Tests aus A.1.1 handelt. Danach haben wir die „exakte“ Exponentialmatrix mit Hilfe von Matlab und der Symbolic Toolbox auf eine Genauigkeit von hundert Stellen berechnet und in R importiert. Anschliessend wurde die R-Implementation des Logarithmus auf diese „exakten“ Exponentialmatrizen angewendet und mit der Ausgangsmatrix verglichen.

Beispiel 1 Zuerst betrachten wir ein allgemeines Beispiel (nach [18]), wie im Beispiel 1 der Exponentialfunktion. Die Ausgangsmatrix sei:

$$A = \begin{pmatrix} 4 & 2 & 0 \\ 1 & 4 & 1 \\ 1 & 1 & 4 \end{pmatrix}$$

Wie erwähnt, wurde $\exp(A)$ in Matlab auf hundert Stellen genau berechnet und dann in R eingelesen. Die Exponentialmatrix hat die Norm $\|\exp(A)\|_1 = 531.21$. Die 1-Norm der Fehlermatrix auf zwei Stellen nach dem Komma gerundet ist wie folgt:

$$\|\log(\exp A_{\text{exakt}}) - A\|_1 = 1.32e-14$$

Man darf sagen, dass die Logarithmusfunktion hier sehr gut abschneidet.

Beispiel 2 Dieses Beispiel (ebenfalls nach [18]) stellt die Algorithmen etwas mehr auf die Probe:

$$B = \begin{pmatrix} -131 & 19 & 18 \\ -390 & 56 & 54 \\ -387 & 57 & 52 \end{pmatrix}$$

Zwar ist hier die Norm der Ausgangsmatrix relativ gross mit $\|B\|_1 = 908$, jedoch ist die Norm der Exponentialmatrix wiederum sehr klein mit $\|\exp(B)\|_1 = 12.08$. Die 1-Norm der Fehlermatrix auf zwei Stellen gerundet ist wie folgt:

$$\|\log(\exp B_{\text{exakt}}) - B\|_1 = 1.02e-6$$

Mit dieser Ausgangsmatrix hat der Algorithmus schon mehr Mühe, da hier die verschiedenen Vorzeichen als auch die dominierende erste Spalte eher Probleme machen.

Beispiel 3 Diese Ausgangsmatrix wurde in Matlab per Zufallsgenerator erstellt und dann noch mit der Ähnlichkeitsabbildung durch die Diagonalmatrix $D = \text{diag}(1, 10^{-4}, 10^4, 10^4, 10^{-4})$ schlecht konditioniert, das heisst $C = D^{-1}ED$. Danach wurde C mit $\|C\|_1 = 177088903$ in R eingelesen (hier auf 7 Stellen gerundet):

$$C = \begin{pmatrix} 8.147237e-01 & 9.754040e-06 & 1.576131e+03 & 1.418863e+03 & 6.557407e-05 \\ 9.057919e+03 & 2.784982e-01 & 9.705928e+07 & 4.217613e+07 & 3.571168e-02 \\ 1.269868e-05 & 5.468815e-09 & 9.571669e-01 & 9.157355e-01 & 8.491293e-09 \\ 9.133759e-05 & 9.575068e-09 & 4.853756e-01 & 7.922073e-01 & 9.339932e-09 \\ 6.323592e+03 & 9.648885e-01 & 8.002805e+07 & 9.594924e+07 & 6.787352e-01 \end{pmatrix}$$

Die Exponentialmatrix hat die Norm $\|\exp(C)\|_1 = 1037255552$.

$$\|\log(\exp C_{\text{exakt}}) - C\|_1 = 0.34$$

Offensichtlich macht die schlechte Konditionierung auch dem Logarithmus zu schaffen, jedoch angesichts der grossen Norm von C ist dies ein kleiner relativer Fehler.

B R-Implementierungen

B.1 Matrixexponential

B.1.1 Exponentialfunktion mit Ausgleichen

55

```
1  ##-----OVERVIEW-----
2
3  #Input:  A; nxn Matrix
4  #Output: e^A Matrixexponential; nxn Matrix
5
6
7  #Function for Calculation of e^A with the Scaling&Squaring Method with Balancing
8
9  # Step 0:   Balancing
10 # Step 1:  Scaling
11 # Step 2:  Padé-Approximation
12 # Step 3:  Squaring
13 # Step 4:  Reverse Balancing
14
15 #R-Implementation of Higham's Algorithm from the Book
16 # "Functions of Matrices - Theory and Computation", Chapter 10, Algorithm 10.20
17
18
19  ##-----CODE-----
20 expm2 ← function(A, balancing=TRUE)
21 {
22
23   ## Check if A is quadratic
24   d ← dim(A)
25   if(length(d) != 2 || d[1] != d[2]) stop("'A' must be a quadratic matrix")
26   n ← d[1]
27
28   if (n <= 1) return(exp(A))
29
30   ## else n >= 2 ... non-trivial case : -----
31
32
33   ##-----STEP 0: BALANCING-----
34   ## if Balancing is designated, Balance the Matrix A
35   ## This Balancing Code is adapted from the R-Forge expm Package,
36   ## which is needed for the dgebal function
37
38   if (balancing) {
39
40     stopifnot(require(expm))
41     baP ← dgebal(A, "P")
42     baS ← dgebal(baP$z, "S")
43     A ← baS$z
44   }
45
46
47   ##-----STEP 1 and STEP 2 SCALING & PADÉ APPROXIMATION-----
48
49   ## Information about the given matrix
50
51   stopifnot(require("Matrix")) # just for this:
52   nA ← norm(A, "1")
53
54   ## try to remain in the same matrix class system:
55   I ← if(is(A,"Matrix")) Diagonal(n) else diag(n)
56
57   ## If the norm is small enough, use the Padé-Approximation (PA) directly
58   if (nA <= 2.1) {
59
60     t ← c(0.015, 0.25, 0.95, 2.1)
61     ## the minimal m for the PA :
62     l ← which.max(nA <= t)
63
64     ## Calculate PA
65     C ← rbind(c(120,60,12,1,0,0,0,0,0),
66              c(30240,15120,3360,420,30,1,0,0,0,0),
67              c(17297280,8648640,1995840,277200,25200,1512,56,1,0,0),
68              c(17643225600,8821612800,2075673600,302702400,30270240,
69                2162160,110880,3960,90,1))
70
71     A2 ← A %*% A
72     P ← I
73     U ← C[1,2]*I
74     V ← C[1,1]*I
75
76     for (k in 1:l) {
77       P ← P %*% A2
78       U ← U + C[1,(2*k)+2]*P
79       V ← V + C[1,(2*k)+1]*P
80     }
81     U ← A %*% U
82     X ← solve(V-U,V+U)
83
84     ## Else, check if norm of A is small enough for m=13.
85     ## If not, scale the matrix
86     else {
87       s ← log2(nA/5.4)
88       B ← A
89       ## Scaling
90       if (s > 0) {
91         s ← ceiling(s)
92         B ← B/(2^s)
93       }
94       else s=0
95
96
97       ## Calculate PA
98
99       c. ← c(64764752532480000,32382376266240000,7771770303897600,
100            1187353796428800, 129060195264000,10559470521600, 670442572800,
101            33522128640, 1323241920, 40840800,960960,16380, 182,1)
102
103       B2 ← B %*% B
104       B4 ← B2 %*% B2
105       B6 ← B2 %*% B4
```

```

106
107 U ← B %*% (B6 %*% (c.[14]*B6 + c.[12]*B4 + c.[10]*B2) +
108 c.[8]*B6 + c.[6]*B4 + c.[4]*B2 + c.[2]*I)
109 V ← B6 %*% (c.[13]*B6 + c.[11]*B4 + c.[9]*B2) +
110 c.[7]*B6 + c.[5]*B4 + c.[3]*B2 + c.[1]*I
111
112 X ← solve(V-U,V+U)
113
114 

---

### STEP 3 SQUARING

---


115
116 for (t in seq_len(s)) X ← X %*% X
117 }
118
119 

---

### STEP 4 REVERSE BALANCING

---


120 if (balancing) { ## reverse the balancing
121
122 d ← baS$scale
123 X ← X * (d * rep(1/d, each = n))
124
125 ## apply inverse permutation (of rows and columns):

```

```

126 pp ← as.integer(baP$scale)
127 if(baP$ii > 1) { ## The lower part
128 for(i in (baP$ii-1):1) { # 'p1' in *reverse* order
129 tt ← X[,i]; X[,i] ← X[,pp[i]]; X[,pp[i]] ← tt
130 tt ← X[i,]; X[i,] ← X[pp[i],]; X[pp[i],] ← tt
131 }
132 }
133
134 if(baP$ii2 < n) { ## The upper part
135 for(i in (baP$ii2+1):n) { # 'p2' in *forward* order
136 ## swap i ↔ pp[i] both rows and columns
137 tt ← X[,i]; X[,i] ← X[,pp[i]]; X[,pp[i]] ← tt
138 tt ← X[i,]; X[i,] ← X[pp[i],]; X[pp[i],] ← tt
139 }
140 }
141 }
142
143 X
144 }

```

B.1.2 Fréchet-Ableitung und Exponentialfunktion

56

```

1 

---

### OVERVIEW

---


2
3 #Input: A; nxn Matrix
4 # E; nxn Matrix
5 #Output: list X: X$expm; e^A Matrixexponential; nxn Matrix
6 # X$Lexpm; Exponential-Fréchet-Derivative L(A,E); nxn Matrix
7
8 #Function for Calculation of e^A and the Exponential Fréchet-Derivation L(A,E)
9 #with the Scaling&Squaring Method
10
11 # Step 1: Scaling (of A and E)
12 # Step 2: Padé-Approximation of e^A and L(A,E)
13 # Step 3: Squaring
14
15 #R-Implementation of Higham's Algorithm from the Article
16 # "Computing Fréchet Derivative of the Matrix Exponential, with an application
17 # to Condition Number Estimation", MIMS EPrint 2008.26, Algorithm 6.4
18
19
20 

---

### CODE

---


21 expm2frech ← function(A,E){
22 ## Check if A is quadratic
23 d ← dim(A)
24 if(length(d) != 2 || d[1] != d[2]) stop("'A' must be a quadratic matrix")
25 if(!identical(d,dim(E)) ) stop("A and E need to have the same Dimension")
26 n ← d[1]
27
28 if (n <= 1) {
29 X←exp(A)
30 X2←E*exp(A)
31 X←list(expm=X, Lexpm=X2)
32 return(X)
33 }
34 ## else n >= 2 ... non-trivial case : 

---


35
36 

---

### STEP 1 & STEP 2: SCALING & PADÉ APPROXIMATION

---



```

```

37
38 #Information about the given matrix
39 stopifnot(require("Matrix")) # just for this:
40 nA ← norm(A , "1")
41
42 ## try to remain in the same matrix class system:
43 I ← if(is(A,"Matrix")) Diagonal(n) else diag(n)
44
45
46 #If the norm is small enough, use directly the Padé-Approximation (PA)
47 if (nA <= 1.78) {
48
49 t ← c(0.0108,0.2,0.783,1.78)
50 ## the minimal m for the PA :
51 l ← which.max(nA <= t)
52
53
54 #Calculate PA for e^A and L(A,E)
55 C ← rbind(c(120,60,12,1,0,0,0,0,0),
56 c(30240,15120,3360,420,30,1,0,0,0,0),
57 c(17297280,8648640,1995840,277200,25200,1512,56,1,0,0),
58 c(17643225600,8821612800,2075673600,302702400,30270240,
59 2162160,110880,3960,90,1))
60
61 j ← l*2+1
62 P ← I
63 U ← C[1,2]*I
64 V ← C[1,1]*I
65 A2 ← A%*%A
66 M2 ← A%*%E+E%*%A
67 M ← M2
68 LU ← C[1,4]*M
69 LV ← C[1,3]*M
70 for (k in seq_len(l-1)){
71 #PA e^A
72 P ← P%*%A2

```

```

73     U ← U+C[1,(2*k)+2]*P
74     V ← V+C[1,(2*k)+1]*P
75
76     #PA L(A,E)
77     M ← A2%*%M + M2%*%P
78     LU ← LU+C[1,(2*(k+1))+2]*M
79     LV ← LV + C[1,(2*(k+1))+1]*M
80   }
81   #PA e^A & L(A,E)
82   P ← P%*%A2
83   U ← U+C[1,(2*(1))+2]*P
84   LU ← A %*% LU+E%*%U
85   U ← A%*%U
86   V ← V+C[1,(2*(1))+1]*P
87
88   X ← solve(V-U,V+U)
89   X2 ← solve(V-U, LU+LV+(LU-LV)%*%X)
90 }
91
92 #Else, check if norm of A is small enough for PA with m=13.
93 #If not, scale the matrix
94 else {
95   s ← log2(nA/4.74)
96   B ← A
97   D ← E
98   #Scaling
99   if (s > 0){
100     s ← ceiling(s)
101     B ← A/(2^s)
102     D ← D/(2^s)}
103
104   C. ← c(64764752532480000,32382376266240000,7771770303897600,
105     1187353796428800, 129060195264000,10559470521600,
106     670442572800,33522128640,1323241920,
107     40840800,960960,16380,182,1)
108
109   #Calculate PA
110   #PA e^A

```

```

111   B2 ← B%*%B
112   B4 ← B2%*%B2
113   B6 ← B2%*%B4
114   W1 ← C.[14]*B6+C.[12]*B4+C.[10]*B2
115   W2 ← C.[8]*B6+C.[6]*B4+C.[4]*B2+C.[2]*I
116   Z1 ← C.[13]*B6+C.[11]*B4+C.[9]*B2
117   Z2 ← C.[7]*B6+C.[5]*B4+C.[3]*B2+C.[1]*I
118   W ← B6%*%W1+W2
119   U ← B%*%W
120   V ← B6%*%Z1+Z2
121
122   #PA L(A,E)
123   M2 ← B%*%D+D%*%B
124   M4 ← B2%*%M2+M2%*%B2
125   M6 ← B4%*%M2+M4%*%B2
126   LW1 ← C.[14]*M6+C.[12]*M4+C.[10]*M2
127   LW2 ← C.[8]*M6+C.[6]*M4+C.[4]*M2
128   LZ1 ← C.[13]*M6+C.[11]*M4+C.[9]*M2
129   LZ2 ← C.[7]*M6+C.[5]*M4+C.[3]*M2
130   LW ← B6%*%LW1+M6%*%W1+LW2
131   LU ← B%*%LW+D%*%W
132   LV ← B6%*%LZ1+M6%*%Z1+LZ2
133
134   X ← solve(V-U,V+U)
135   X2 ← solve(V-U,LU+LV+(LU-LV)%*%X)
136
137   #-----STEP 3 SQUARING-----
138   #Squaring
139   if (s > 0) for (t in seq_len(s)) {
140     X2 ← X2%*%X+X%*%X2
141     X ← X%*%X
142   }
143
144   }
145   X ← list(expm=X, Lexpm=X2)
146   X
147 }

```

B.2 Exponentialkonditionszahl

B.2.1 Exakte Exponentialkonditionszahl

```
1 ##-----OVERVIEW-----
2
3 #Input: A; nxn Matrix
4 #Output: list C: C$expondF: Exponentialconditionnumber Frobeniusnorm; scalar
5 # C$expond1: Exponentialconditionnumber 1-Norm; scalar
6 # C$expm: e^A Matrixexponential; nxn Matrix
7
8
9 #Function for exact calculation of the Exponentialconditionnumber ("1" and
10 #Frobenius-Norm). This function uses the function expm2frech for the calculation
11 #of the fréchet derivative.
12
13 # Step 1: Calculate Kroneckermatrix of L(A)
14 # Step 2: Calculate Exponentialconditionnumber ("1" & Frobenius-Norm)
15
16 #R-Implementation of Higham's Algorithm from the book
17 #"Functions of Matrices – Theory and Computation", chapter 3.4, algorithm 3.17
18
19
20 ##-----CODE-----
21 expcond ← function(A){
22   d ← dim(A)
23   if(length(d) != 2 || d[1] != d[2] || d[1]<=1)
24     stop("'A' must be a quadratic matrix and n>1")
25   n ← d[1]
26   #-----STEP 1: Calculate Kroneckermatrix of L(A)-----
27   K ← matrix(c(rep(0,n^4)),n^2,n^2)
28   v ← c(rep(0,n^2))
```

```
29
30   for (j in 1:n){
31     ej ← c(rep(0,n))
32     ej[j] ← 1
33
34     for (i in 1:n){
35       ei ← c(rep(0,n))
36       ei[i] ← 1
37       Eij ← ei%*%t(ej)
38       calc ← expm2frech(A,Eij)
39       Y ← calc$Lexpm
40       K[, (j-1)*n+i] ← as.vector(Y)
41     }
42   }
43 }
44
45 #-----STEP 2 CALCULATE EXPONENTIALCONDITIONNUMBER-----
46 # Frobenius-Norm
47 normk ← sqrt(max(eigen(t(K)%*%K)$values))
48 CF ← normk*norm(A,"F")/norm(calc$expm,"F")
49
50 # 1-Norm
51 C1 ← norm(K,"1")*norm(A,"1")/(norm(calc$expm,"1")*n)
52 C ← list(expondF=CF, expond1=C1, expm=calc$expm)
53 }
54
55 }
```

B.2.2 Schätzung der Exponentialkonditionszahl (1-Norm)

```
1 ##-----OVERVIEW-----
2
3 #Input: A; nxn Matrix
4 #Output: list C: C$expond: Exponentialconditionnumber "1"-Norm; scalar
5 # C$expm: e^A Matrixexponential; nxn Matrix
6
7 #Function for Estimation of the "1"-norm exponentialconditionnumber based on
8 #the LAPACK marix norm estimator.
9 #This function uses the function expm2frech for the calculation
10 #of the fréchet derivative.
11
12 # Step 1: Estimate "1"-Norm of Kroneckermatrix K(A)
13 # This step is based on the equation: K(A)vec(E)=vec(L(A,E))
14
15 # Step 2: Calculate Exponentialconditionnumber ("1"-Norm)
16
17 #R-Implementation of Higham's Algorithm from the book
18 #"Functions of Matrices – Theory and Computation", chapter 3.4, algorithm 3.21
19
20
21 ##-----CODE-----
```

```
22 expcondest1 ← function(A){
23
24 #-----STEP 1 ESTIMATE "1"-NORM FROM THE KRONECKERMATRIX-----
25 ## Check if A is quadratic
26 d ← dim(A)
27 if(length(d) != 2 || d[1] != d[2] || d[1]<=1)
28   d stop("'A' must be a quadratic matrix and n>1")
29 n ← d[1]
30
31 E ← matrix(c(rep(1/n^2,n^2)),n,n)
32 calc ← expm2frech(A,E)
33 V ← calc$Lexpm
34 G ← sum(abs(V))
35 Z ← sign(V)
36 X ← expm2frech(t(A),Z)$Lexpm
37 k=2
38
39 repeat {
40
41   j ← which.max(as.vector(abs(X)))
42   Ej ← matrix(c(rep(0,n^2)),n,n)
```

```

43   Ej[j] ← 1
44   V ← expm2frech(A,Ej)$Lexpm
45   G ← sum(abs(V))
46   if (identical(sign(V),Z) || identical(sign(V),-Z)) break
47   Z ← sign(V)
48   X ← expm2frech(t(A),Z)$Lexpm
49   k ← k+1
50   if (max(abs(X))==X[j] | k>5) break
51 }
52
53 for (l in 1:(n^2)) {
54   X[l] ← (-1)^(1+l)*(1+(1-l)/(n^2-1))
55 }

```

```

56
57 X ← expm2frech(A,X)$Lexpm
58
59 if (2*sum(abs(X))/(3*n^2)>G) {
60   G ← 2*sum(abs(X))/(3*n^2)
61 }
62 #-----STEP 2 CALCULATE EXPONENTIALCONDITIONNUMBER-----
63 C ← G * norm(A,"1")/(norm(calc$expm,"1")*n)
64 C ← list(expcnd=C, expm=calc$expm)
65 C
66 }

```

B.2.3 Schätzung der Exponentialkonditionszahl (Frobenius-Norm)

```

1  ##-----OVERVIEW-----
2
3  #Input:  A: nxn Matrix
4  #Output: list C: C$expcnd: Exponentialconditionnumber Frobeniusnorm; scalar
5  #        C$expm: e^A Matrixexponential; nxn Matrix
6
7  #Function for estimation of the frobenius-Norm exponentialconditionnumber based
8  #on the powermethod-matrixnorm estimation.
9  #This function uses the function expm2frech for the calculation
10 #of the fréchet derivative.
11
12 # Step 1: Estimate "2"-Norm of Kroneckermatrix K(A)
13 #        This step is based on the equation: K(A)vec(E)=vec(L(A,E))
14
15 # Step 2: Calculate Exponentialconditionnumber (Frobenius-Norm)
16
17 #R-Implementation of Higham's Algorithm from the book
18 #"Functions of Matrices - Theory and Computation", chapter 3.4, algorithm 3.19
19
20
21 ##-----CODE-----
22 expcndestfrob ← function(A){
23
24   ## Check if A is quadratic
25   d ← dim(A)
26   if(length(d) != 2 || d[1] != d[2] || d[1]<=1)
27     stop("'A' must be a quadratic matrix and n>1")

```

```

28   n ← d[1]
29
30
31   #-----STEP 1 ESTIMATE 2-NORM OF KRONECKERMATRIX-----
32
33   n ← d[1]
34   Z1 ← if(is(A,"Matrix")) Matrix(rnorm(n*n),n,n) else matrix(rnorm(n*n),n,n)
35   calc ← expm2frech(A,Z1)
36   W1 ← calc$Lexpm
37   Z1 ← expm2frech(t(A),W1)$Lexpm
38   G2 ← norm(Z1,"F")/norm(W1,"F")
39   G1 ← G2-1
40
41   while (abs(G1-G2)>10^(-1)){
42     G1 ← G2
43     W2 ← expm2frech(A,Z1)$Lexpm
44     Z2 ← expm2frech(t(A),W2)$Lexpm
45     G2 ← norm(Z2,"F")/norm(W2,"F")
46     Z1 ← Z2
47   }
48
49   #-----STEP 2 CALCULATE EXPONENTIALCONDITIONNUMBER-----
50   C ← G2*norm(A,"F")/norm(calc$expm,"F")
51   C ← list(expcnd=C, expm=calc$expm)
52   C
53 }

```

B.3 Matrixlogarithmus

B.3.1 Wurzelfunktion

```
1  ##-----OVERVIEW-----
2
3  #Input: A; nxn matrix, no eigenvalues <=0, not singular
4  #Output: root of matrix A, nxn Matrix
5
6
7  #Function for calculation of A^(1/2) with the real Schur decomposition
8
9  # Step 0:    real Schur decomposition T of A
10 # Step 1:   Analyze block structure of T
11 # Step 2:   Calculate diagonal elements/blocks of T^(1/2)
12 # Step 3:   Calculate superdiagonal elements/blocks of T^(1/2)
13 # Step 4:   reverse Schur decomposition
14
15 #R-Implementation of Higham's Algorithm from the Book
16 # "Functions of Matrices - Theory and Computation", Chapter 6, Algorithm 6.7
17
18 ##-----CODE-----
19 root <- function(A){
20   #Generate Basic information of Matrix A
21   A <- as.matrix(A)
22   d <- dim(A)
23   if(length(d) != 2 || d[1] != d[2]) stop("'A' must be a quadratic matrix")
24   if(det(A)==0) stop("'A' is singular")
25   n <- d[1]
26   ##-----STEP 0: Schur Decomposition-----
27   stopifnot(require("Matrix")) # just for this:
28   Sch.A <- Schur(Matrix(A))
29   ev <- Sch.A@EValues
30   if(!all(Arg(ev)!=pi)) stop("'A' has negative real eigenvalues!")
31
32   S <- as.matrix(Sch.A@T)
33   Q <- as.matrix(Sch.A@Q)
34   X <- matrix(0,n,n)
35   I <- diag(1,2)
36   k <- 0
37   ##-----STEP 1: Analyze block structure-----
38   #Count 2x2 blocks (as Schur(A) is the real Schur Decomposition)
39   for (i in seq_len(n-1)){
40     if (S[i+1,i]!=0) k <- k+1
41   }
42
43   #Generate Blockstructure and save it as R.index
44   R.index <- vector("list",n-k)
45   i <- 1
46   l <- 1
47   while(i<n){
48     if (S[i+1,i]==0) {
49       R.index[[l]] <- i
50     }
51     else {
52       R.index[[l]] <- (i:(i+1))
53       i <- i+1
54     }
55     i <- i+1
56     l <- l+1
57   }
58   if (is.null(R.index[[n-k]])) R.index[[n-k]] <- n
59
60   ##-----STEP 2: Calculate diagonal elements/blocks-----
61   #Calculate the root of the diagonale Blocks of the Schur Decompostion S
62   for (j in seq_len(n-k)){
63
64     if (length(R.index[[j]])==1) {
65       X[R.index[[j]],R.index[[j]]] <- sqrt(S[R.index[[j]],R.index[[j]])
66     }
67     else { #This part would work easier due to the special form of the 2x2 block
68       ev <- Sch.A@EValues[R.index[[j]]]
69       Re.ev <- Re(ev[1])
70       Re.sqev <- Re(sqrt(ev[1]))
71       X[R.index[[j]],R.index[[j]]] <-
72       Re.sqev*I+1/(2*Re.sqev)*S[R.index[[j]],R.index[[j]]-Re.ev*I)
73     }
74   }
75   ##-----STEP 2: Calculate superdiagonal elements/blocks-----
76
77   #Calculate the remaining, not-diagonal blocks
78   if (n-k>1){
79     for (j in 2:(n-k)){
80       for (i in (j-1):1){
81         sumU <- 0
82
83         #Calculation for 1x1 Blocks
84         if (length(R.index[[i]])==1 & length(R.index[[j]])==1 ){
85           if (j-i>1){
86             for (l in (i+1):(j-1)) {
87               if (length(R.index[[l]])==2 ){
88                 sumU <-
89                 sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
90               else {sumU <-
91                 sumU+X[R.index[[i]],R.index[[l]]]*X[R.index[[l]],R.index[[j]]]
92               }
93             }
94             X[R.index[[i]],R.index[[j]]] <-
95             solve(X[R.index[[i]],R.index[[i]]+X[R.index[[j]],R.index[[j]]],
96                 S[R.index[[i]],R.index[[j]]]-sumU)
97           }
98
99           #Calculation for 1x2 Blocks
100          else if (length(R.index[[j]])==2 & length(R.index[[i]])==1 ){
101            if (j-i>1){
102              for (l in (i+1):(j-1)) {
103                if (length(R.index[[l]])==2 ){
104                  sumU <-
105                  sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
106                else {sumU <-
107                  sumU+X[R.index[[i]],R.index[[l]]]*X[R.index[[l]],R.index[[j]]]
108                }
109              }
110              X[R.index[[i]],R.index[[j]]] <-
111              solve(t(X[R.index[[i]],R.index[[i]]+I+X[R.index[[j]],R.index[[j]]]),
112                  as.vector(S[R.index[[i]],R.index[[j]]-sumU))
113            }
114            #Calculation for 2x1 Blocks
```

```

115     else if (length(R.index[[j]])==1 & length(R.index[[i]])==2 ){
116         if (j-i>1){
117             for (l in (i+1):(j-1)) {
118                 if (length(R.index[[l]])==2 ){
119                     sumU ← sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
120                     else {sumU ←
121                         sumU+X[R.index[[i]],R.index[[l]]]*X[R.index[[l]],R.index[[j]]]}
122                 }
123             }
124             X[R.index[[i]],R.index[[j]]] ←
125             solve(X[R.index[[i]],R.index[[i]]]+X[R.index[[j]],R.index[[j]]]*
126             I,S[R.index[[i]],R.index[[j]]]-sumU)
127         }
128         #Calculation for 2x2 Blocks with special equation for solver
129         else if (length(R.index[[j]])==2 & length(R.index[[i]])==2 ){
130             if (j-i>1){
131                 for (l in (i+1):(j-1)) {
132                     if (length(R.index[[l]])==2 ){
133                         sumU ←
134                         sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
135                         else {sumU ←
136                             sumU+X[R.index[[i]],R.index[[l]]]*%*%t(X[R.index[[l]],R.index[[j]]))}
137                     }
138                 }
139             }
140             tUii ← Matrix(0,4,4)
141             tUii[1:2,1:2] ← X[R.index[[i]],R.index[[i]]]
142             tUii[3:4,3:4] ← X[R.index[[i]],R.index[[i]]]
143             tUjj ← Matrix(0,4,4)
144             tUjj[1:2,1:2] ← t(X[R.index[[j]],R.index[[j]])[1,1]*I
145             tUjj[3:4,3:4] ← t(X[R.index[[j]],R.index[[j]])[2,2]*I
146             tUjj[1:2,3:4] ← t(X[R.index[[j]],R.index[[j]])[1,2]*I
147             tUjj[3:4,1:2] ← t(X[R.index[[j]],R.index[[j]])[2,1]*I
148             X[R.index[[i]],R.index[[j]]] ←
149             solve(tUii+tUjj,as.vector(S[R.index[[i]],R.index[[j]]]-sumU))
150         }
151     }
152 }
153
154 ##-----STEP 3: Reverse the Schur Decomposition-----
155 #Reverse the Schur Decomposition
156 X ← Q%*%X%*%solve(Q)
157 X
158 }

```

B.3.2 Wurzelfunktion für die Logarithmusfunktion

61

```

1 ##-----OVERVIEW-----
2
3 #Input: UT; nxn upper triangular block matrix (real Schur decomposition)
4 #Output: root of matrix UT, nxn upper triangular Matrix
5
6
7 #Function for calculation of UT^(1/2), which is used for the logarithm function
8
9 # Step 0: Analyze block structure
10 # Step 1: Calculate diagonal elements/blocks
11 # Step 2: Calculate superdiagonal elements/blocks
12
13 #R-Implementation of Higham's Algorithm from the Book
14 # "Functions of Matrices – Theory and Computation", Chapter 6, Algorithm 6.7
15
16 ##-----CODE-----
17 roots ← function(UT){
18     #Generate Basic information of Matrix UT
19     n ← dim(UT)[1]
20     X ← matrix(0,n,n)
21     S ← as.matrix(UT)
22     I ← diag(1,2)
23     k ← 0
24
25     ##-----STEP 0: Analyze block structure-----
26     #Count 2x2 blocks (as Schur(A) is the real Schur Decompostion)
27     for (i in seq_len(n-1)){
28         if (S[i+1,i]!=0) k ← k+1
29     }
30
31     #Generate Blockstructure and save it as R.index
32     R.index ← vector("list",n-k)
33     i ← 1
34     l ← 1
35     while(i<n){
36         if (S[i+1,i]==0) {
37             R.index[[l]] ← i
38         }
39         else {
40             R.index[[l]] ← (i:(i+1))
41             i ← i+1
42         }
43         i ← i+1
44         l ← l+1
45     }
46     if (is.null(R.index[[n-k]])) R.index[[n-k]] ← n
47
48     ##-----STEP 1: Calculate diagonal elements/blocks-----
49     #Calculate the root of the diagonal blocks of the Schur Decompostion S
50     for (j in seq_len(n-k)){
51
52         if (length(R.index[[j]])==1) {
53             X[R.index[[j]],R.index[[j]]] ← sqrt(S[R.index[[j]],R.index[[j]])
54         }
55         else {
56             ev ← eigen(S[R.index[[j]],R.index[[j]])$values
57             Re.ev ← Re(ev[1])
58             Re.sqev ← Re(sqrt(ev[1]))
59             X[R.index[[j]],R.index[[j]]] ←
60             Re.sqev*I+1/(2*Re.sqev)*(S[R.index[[j]],R.index[[j]]]-Re.ev*I)
61         }
62     }
63     ##-----STEP 1: Calculate superdiagonal elements/blocks-----
64     #Calculate the remaining, not-diagonal blocks
65     if (n-k>1){
66         for (j in 2:(n-k)){
67             for (i in (j-1):1){
68                 sumU ← 0

```

```

69
70     #Calculation for 1x1 Blocks
71     if (length(R.index[[j]])==1 & length(R.index[[i]])==1 ){
72         if (j-i>1){
73             for (l in (i+1):(j-1)) {
74                 if (length(R.index[[l]])==2 ){
75                     sumU ←
76 sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
77                     else {sumU ←
78 sumU+X[R.index[[i]],R.index[[l]]]*X[R.index[[l]],R.index[[j]]]}
79                 }
80             }
81             X[R.index[[i]],R.index[[j]]] ←
82 solve(X[R.index[[i]],R.index[[i]]+
83 X[R.index[[j]],R.index[[j]]],S[R.index[[i]],R.index[[j]]]-sumU)
84         }
85     }
86     #Calculation for 1x2 Blocks
87     else if (length(R.index[[j]])==2 & length(R.index[[i]])==1 ){
88         if (j-i>1){
89             for (l in (i+1):(j-1)) {
90                 if (length(R.index[[l]])==2 ){
91                     sumU ←
92 sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
93                     else {sumU ←
94 sumU+X[R.index[[i]],R.index[[l]]]*X[R.index[[l]],R.index[[j]]]}
95                 }
96             }
97             X[R.index[[i]],R.index[[j]]] ←
98 solve(t(X[R.index[[i]],R.index[[i]]*I+
99 X[R.index[[j]],R.index[[j]]]),as.vector(S[R.index[[i]],R.index[[j]]]-sumU))
100         }
101     }
102     #Calculation for 2x1 Blocks
103     else if (length(R.index[[j]])==1 & length(R.index[[i]])==2 ){
104         if (j-i>1){
105             for (l in (i+1):(j-1)) {
106                 if (length(R.index[[l]])==2 ){
107                     sumU ←
108 sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
109                     else {sumU ←
110 sumU+X[R.index[[i]],R.index[[l]]]*X[R.index[[l]],R.index[[j]]]}
111                 }
112             }
113             X[R.index[[i]],R.index[[j]]] ←
114 solve(X[R.index[[i]],R.index[[i]]+
115 X[R.index[[j]],R.index[[j]]]*I,S[R.index[[i]],R.index[[j]]]-sumU)
116         }
117     }
118     #Calculation for 2x2 Blocks with special equation for solver
119     else if (length(R.index[[j]])==2 & length(R.index[[i]])==2 ){
120         if (j-i>1){
121             for (l in (i+1):(j-1)) {
122                 if (length(R.index[[l]])==2 ){
123                     sumU ←
124 sumU+X[R.index[[i]],R.index[[l]]]*%*%X[R.index[[l]],R.index[[j]]] }
125                     else {sumU ←
126 sumU+X[R.index[[i]],R.index[[l]]]*%*%t(X[R.index[[l]],R.index[[j]]]}
127                 }
128             }
129             tUii ← matrix(0,4,4)
130             tUii[1:2,1:2] ← X[R.index[[i]],R.index[[i]]]
131             tUii[3:4,3:4] ← X[R.index[[i]],R.index[[i]]]
132             tUjj ← matrix(0,4,4)
133             tUjj[1:2,1:2] ← t(X[R.index[[j]],R.index[[j]])[1,1]*I
134             tUjj[3:4,3:4] ← t(X[R.index[[j]],R.index[[j]])[2,2]*I
135             tUjj[1:2,3:4] ← t(X[R.index[[j]],R.index[[j]])[1,2]*I
136             tUjj[3:4,1:2] ← t(X[R.index[[j]],R.index[[j]])[2,1]*I
137             X[R.index[[i]],R.index[[j]]] ←
138 solve(tUii+tUjj,as.vector(S[R.index[[i]],R.index[[j]])-sumU))
139         }
140     }
141 }
142 }

```

B.3.3 Logarithmusfunktion

```

1  ##-----OVERVIEW-----
2
3 #Input: A; nxn Matrix, no eigenvalues <=0, not singular
4 #Output: log(A); Matrixlogarithm; nxn Matrix
5
6
7 #Function for Calculation of log(A) with the Inverse Scaling&Squaring Method
8
9 # Step 0: Schur Decomposition Tr
10 # Step 1: Scaling (root of Tr)
11 # Step 2: Padé-Approximation
12 # Step 3: Squaring
13 # Step 4: Reverse Schur Decomposition
14
15 #R-Implementation of Higham's Algorithm from the Book
16 # "Functions of Matrices – Theory and Computation", Chapter 11, Algorithm 11.9
17
18 ##-----CODE-----
19 logm2 ← function(A){
20
21     A ← as.matrix(A)
22     d ← dim(A)
23     if(length(d) != 2 || d[1] != d[2]) stop("'A' must be a quadratic matrix")
24     if(det(A)==0) stop("'A' is singular")
25     n ← d[1]
26     ##-----Step 0: Schur Decomposition-----
27     stopifnot(require("Matrix")) # just for this:
28     Schur.A ← Schur(Matrix(A))
29     ev ← Schur.A@EValues
30     if(!all(Arg(ev)!=pi)) stop("'A' has negative real eigenvalues!")
31     Tr ← as.matrix(Schur.A@T)
32     Q ← as.matrix(Schur.A@Q)
33     I ← diag(1,n)
34     k ← 0
35     p ← 0
36     theta ← c(0.0162,0.0539, 0.114,0.187, 0.264)
37     ##-----Step 1: Scaling-----
38     repeat{

```

```

39     t ← norm(Tr-I,"1")
40     if (t<0.264){
41         p ← p+1
42         j1 ← which.max( t <= theta)
43         j2 ← which.max( (t/2) <= theta)
44         if ((j1-j2<=1) | (p==2)) {
45             m ← j1
46             break
47         }
48     }
49     Tr ← rootS(Tr)
50     k ← k+1
51 }
52 ### Step 2: Padé-Approximation
53 r ← rbind(c( 5003999585967230*2^(-54), 8006399337547537*2^(-54),
54             5/18,0,0,0,0),
55           c( 5640779706068081*2^(-51), 8899746432686114*2^(-53),
56             8767290225458872*2^(-54), 6733946100265013*2^(-55)
57             0,0,0),
58           c( 5686538473148996*2^(-51), 4670441098084653*2^(-52),
59             5124095576030447*2^(-53), 5604406634440294*2^(-54),
60             8956332917077493*2^(-56), 0, 0),
61           c( 5712804453675980*2^(-51), 4795663223967718*2^(-52),
62             5535461316768070*2^(-53), 6805310445892841*2^(-54),
63             7824302940658783*2^(-55), 6388318485698934*2^(-56),
64             0),
65           c( 5729264333934497*2^(-51), 4873628951352824*2^(-52),
66             5788422587681293*2^(-53), 7529283295392226*2^(-54),
67             4892742764696865*2^(-54), 5786545115272933*2^(-55),

```

```

68             4786997716777457*2^(-56)))
69
70 p ← rbind(c( -7992072898328873*2^(-53), -1/2, -8121010851296995*2^(-56),
71             0,0,0,0),
72           c( -8107950463991866*2^(-49), -6823439817291852*2^(-51),
73             -6721885580294475*2^(-52), -4839623620596807*2^(-52),
74             0,0,0),
75           c( -6000309411699298*2^(-48), -4878981751356277*2^(-50),
76             -2^1, -5854649940415304*2^(-52), -4725262033344781*2^(-52),
77             0,0),
78           c( -8336234321115872*2^(-48), -6646582649377394*2^(-50),
79             -5915042177386279*2^(-51), -7271968136730531*2^(-52),
80             -5422073417188307*2^(-52), -4660978705505908*2^(-52),0),
81           c( -5530820008925390*2^(-47), -8712075454469181*2^(-50),
82             -7579841581383744*2^(-51), -4503599627370617*2^(-51),
83             -6406963985981958*2^(-52), -5171999978649488*2^(-52),
84             -4621190647118544*2^(-52)))
85
86 X ← 0
87 Tr ← Tr-I
88 for (s in 1:(m+2)) {
89     X ← X+r[m,s]*solve(Tr-p[m,s]*I,Tr)
90 }
91 ### Step 3 & 4: Squaring & reverse Schur Decomposition
92
93 X ← 2^k*Q%*X%*solve(Q)
94 X
95 }

```

Literatur

- [1] Awad H. Al-Mohy and Nicholas J. Higham. Computing the Fréchet derivative of the matrix exponential, with an application to condition number estimation. MIMS EPrint 2008.26, Manchester Institute for Mathematical Sciences, University of Manchester, Feb 2008.
- [2] David W. Boyd. The power method for ℓ^p norms. *Linear Algebra Appl.*, 9:95–101, 1974.
- [3] Philip J. Davis and Philip Rabinowitz. *Methods of Numerical Integration*. Academic Press, London, second edition, 1984.
- [4] Walter Gautschi. Algorithm 726: ORTHPOL—A package of routines for generating orthogonal polynomials and Gauss-type quadrature rules. *ACM Trans. Math. Software*, 20(1):21–62, 1994.
- [5] Walter Gautschi. *Numerical Analysis: An Introduction*. Birkhäuser, Boston, MA, USA, 1997.
- [6] Walter Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, 2004.
- [7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.
- [8] Nicholas J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation (Algorithm 674). *ACM Trans. Math. Software*, 14(4):381–396, December 1988.
- [9] Nicholas J. Higham. Experience with a matrix norm estimator. *SIAM J. Sci. Statist. Comput.*, 11(4):804–809, July 1990.
- [10] Nicholas J. Higham. Evaluating Padé approximants of the matrix logarithm. *SIAM J. Matrix Anal. Appl.*, 22(4):1126–1135, 2001.
- [11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
- [12] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.
- [13] George A. Baker Jr. *Essentials of Padé Approximants*. Academic Press, New York, 1975.
- [14] Cleve B. Moler and Charles F. Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Rev.*, 45(1):3–49, 2003.
- [15] B.N. Parlett and C.Reinsch. Balancing a matrix for calculation of eigenvalues and eigenvectors. *Numer. Math.*, 13(4):293–304, 1969.

- [16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.
- [17] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.
- [18] Robert C. Ward. Numerical computation of the matrix exponential with accuracy estimate. *SIAM J. Numer. Anal.*, 14(4):600–610, 1977.
- [19] David S. Watkins. A case where balancing is harmful. *Electron. Trans. Numer. Anal.*, 23:1–4, 2006.