

NUMERISCHES LÖSEN GROSSER DÜNNBESETZTER MATRIXGLEICHUNGEN IN PYTHON

An der Fakultät für Mathematik
der Otto-von-Guericke-Universität Magdeburg
zur Erlangung des akademischen Grades
Bachelor of Science
angefertigte

BACHELORARBEIT

vorgelegt von
BJÖRN BARAN
geboren am 08.12.1989 in Flensburg,
Studiengang Mathematik,
Studienrichtung Mathematik.

6. Dezember 2013

Betreut am Max-Planck-Institut
für Dynamik komplexer technischer Systeme von
DR. JENS SAAK

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
Tabellenverzeichnis	VI
Algorithmenverzeichnis	VII
Symbolverzeichnis	VIII
1. Einleitung	1
1.1. Motivation	1
1.2. Einführendes Beispiel	1
1.3. Kurzfassung	3
2. Numerische Verfahren zur Lösung der Lyapunov-Gleichungen	4
2.1. Das ADI Verfahren	4
2.2. Das ADI Verfahren für Lyapunov-Gleichungen	5
3. Konvergenz des ADI Algorithmus	7
4. Niedrigrangfaktoren	9
4.1. Niedrigrangfaktor der Lösung	9
4.2. Vermeidung komplexer Lösungsfaktoren	10
4.3. Niedrigrangfaktor des Residuums	13
4.4. Vermeidung komplexer Arithmetik beim Niedrigrangfaktor des Residuums	15
5. Verallgemeinerte Lyapunov-Gleichungen	16
6. Implementierung in Python	18
6.1. Direkte Implementierung in Python	18
6.2. Interface zwischen Python und C-M.E.S.S.	21
7. Experimenteller Vergleich zu C-M.E.S.S.	22
7.1. Sequentielle Tests	22
7.2. Parallele Tests	27
8. Zusammenfassung	34
9. Ausblick	35
A. Testergebnisse	36
A.1. Sequentielle Tests	36

A.1.1. fdm_2d_unsym	36
A.1.2. MSD_TripleChain	37
A.2. Parallele Tests	37
A.2.1. fdm_2d_sym	37
A.2.2. fdm_2d_unsym	40
A.2.3. MSD_TripleChain	44
A.3. fdm_2d_sym ohne numactl	48
A.4. Datenträger	50
Literaturverzeichnis	52

Abbildungsverzeichnis

1.1. Gebietsaufteilung der instationären Wärmeleitung	2
6.1. Beispielcode zur Demonstration des Fehlers im Zusammenhang mit dem shape Attribut	20
6.2. Ausgabe von 6.1	20
7.1. Laufzeit von sequentieller Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_sym</code> : Python vs. C	23
7.2. Verhältnis der Laufzeit von Python zu C für <code>fdm_2d_sym</code> (sequentiell) .	23
7.3. Laufzeit von sequentieller Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_unsym</code> : Python vs. C	25
7.4. Verhältnis der Laufzeit von Python zu C für <code>fdm_2d_unsym</code> (sequentiell)	26
7.5. Laufzeit von sequentieller Shift-Parameter Berechnung und LRCF_ADI für <code>MSD_TripleChain</code> : Python vs. C	26
7.6. Verhältnis der Laufzeit von Python zu C für <code>MSD_TripleChain</code> (se- quentiell)	27
7.7. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_sym</code> mit dem C-M.E.S.S. Interface	28
7.8. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_sym</code> mit dem C-M.E.S.S. Interface (logarithmische Achsenska- lierung)	29
7.9. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_sym</code> mit Python	30
7.10. Verhältnis der Laufzeit von Python zu C für <code>fdm_2d_sym</code> mit mehreren Threads	30
7.11. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_sym</code> mit Python mit und ohne <code>numactl</code>	32
7.12. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_sym</code> mit dem C-M.E.S.S. Interface mit und ohne <code>numactl</code> . .	33
A.1. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_unsym</code> mit dem C-M.E.S.S. Interface	40
A.2. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>fdm_2d_unsym</code> mit Python	40
A.3. Verhältnis der Laufzeit von Python zu C für <code>fdm_2d_unsym</code> mit mehreren Threads	41
A.4. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>MSD_TripleChain</code> mit dem C-M.E.S.S. Interface	44
A.5. Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für <code>MSD_TripleChain</code> mit Python	44

A.6. Verhältnis der Laufzeit von Python zu C für MSD_TripleChain mit mehreren Threads 45

Tabellenverzeichnis

6.1. Software Versionen	18
7.1. Laufzeiten für <code>fdm_2d_sym</code> mit einem Thread	24
A.1. Laufzeiten für <code>fdm_2d_unsym</code> mit einem Thread	36
A.2. Laufzeiten für <code>MSD_TripleChain</code> mit einem Thread	37
A.3. Laufzeiten für <code>fdm_2d_sym</code> mit 2 Threads	37
A.4. Laufzeiten für <code>fdm_2d_sym</code> mit 4 Threads	38
A.5. Laufzeiten für <code>fdm_2d_sym</code> mit 8 Threads	38
A.6. Laufzeiten für <code>fdm_2d_sym</code> mit 12 Threads	39
A.7. Laufzeiten für <code>fdm_2d_sym</code> mit 16 Threads	39
A.8. Laufzeiten für <code>fdm_2d_unsym</code> mit 2 Threads	41
A.9. Laufzeiten für <code>fdm_2d_unsym</code> mit 4 Threads	42
A.10. Laufzeiten für <code>fdm_2d_unsym</code> mit 8 Threads	42
A.11. Laufzeiten für <code>fdm_2d_unsym</code> mit 12 Threads	43
A.12. Laufzeiten für <code>fdm_2d_unsym</code> mit 16 Threads	43
A.13. Laufzeiten für <code>MSD_TripleChain</code> mit 2 Threads	45
A.14. Laufzeiten für <code>MSD_TripleChain</code> mit 4 Threads	46
A.15. Laufzeiten für <code>MSD_TripleChain</code> mit 8 Threads	46
A.16. Laufzeiten für <code>MSD_TripleChain</code> mit 12 Threads	47
A.17. Laufzeiten für <code>MSD_TripleChain</code> mit 16 Threads	47
A.18. Laufzeiten für <code>fdm_2d_sym</code> mit einem Thread ohne <code>numactl</code>	48
A.19. Laufzeiten für <code>fdm_2d_sym</code> mit 2 Threads ohne <code>numactl</code>	48
A.20. Laufzeiten für <code>fdm_2d_sym</code> mit 4 Threads ohne <code>numactl</code>	49
A.21. Laufzeiten für <code>fdm_2d_sym</code> mit 8 Threads ohne <code>numactl</code>	49

Algorithmenverzeichnis

1.	Low-Rank Cholesky factor ADI iteration (LRCF-ADI-v1)	10
2.	Low-Rank Cholesky factor ADI iteration (LRCF-ADI-v2)	13
3.	Low-Rank Cholesky factor ADI iteration (LRCF-ADI-v3)	15
4.	Generalized Low-Rank Cholesky factor ADI iteration (G-LRCF-ADI) . .	17

Symbolverzeichnis

\mathbb{R}	Die Menge der reellen Zahlen
\mathbb{C}	Die Menge der komplexen Zahlen
\mathbb{R}_-	Die offene linke reelle Halbachse $\{x \in \mathbb{R} : x < 0\}$
\mathbb{C}_-	Die offene linke komplexe Halbebene $\{x \in \mathbb{C} : \operatorname{Re}(x) < 0\}$
\mathbb{R}^n	Der n -dimensionale Vektorraum über \mathbb{R}
\mathbb{C}^n	Der n -dimensionale Vektorraum über \mathbb{C}
$\mathbb{R}^{n \times m}$	Der Vektorraum der $n \times m$ Matrizen über \mathbb{R}
$\mathbb{C}^{n \times m}$	Der Vektorraum der $n \times m$ Matrizen über \mathbb{C}
\bar{x}	Das komplex Konjugierte von $x \in \mathbb{C}$
\bar{A}	Das komplex Konjugierte der Matrix $A \in \mathbb{C}^{n \times m}$
$\operatorname{Re}(x)$	Der Realteil von $x \in \mathbb{C}$
$\operatorname{Im}(x)$	Der Imaginärteil von $x \in \mathbb{C}$
I	Die Einheitsmatrix
0	Die Nullmatrix
$\mathbb{1}$	Der Einsvektor
$\det(A)$	Die Determinante der Matrix A
$\Lambda(A)$	Das Spektrum der Matrix A $\{\lambda \in \mathbb{C} : \det(A - \lambda I) = 0\}$
A^T	Das Transponierte der Matrix A
A^H	Das komplex konjugierte Transponierte der Matrix A
A^{-1}	Das Inverse der Matrix A
$\ x\ _2$	Die euklidische Norm des Vektors x
$\ A\ _2$	Die Spektralnorm der Matrix A
$\operatorname{cond}(A)$	Die Kondition $\ A\ _2 \ A^{-1}\ _2$ der Matrix A
$A \otimes B$	Das Kronecker-Produkt der Matrizen A und B

Danksagung

An erster Stelle möchte ich mich bei Dr. Jens Saak für seine ganze Zeit und den Aufwand bedanken. Er stand mir bei jeder Frage geduldig zur Seite und hat durch eine sehr gute Betreuung diese Bachelorarbeit sowie meine Tätigkeit am Max-Planck-Institut erst ermöglicht. Dank gilt in diesem Zusammenhang auch Prof. Dr. Peter Benner, welcher außerdem Zweitgutachter dieser Arbeit ist.

Sehr viel Unterstützung habe ich auch von Dipl.-Math. Martin Köhler erhalten, welcher dafür gesorgt hat, dass die Software wie C-M.E.S.S., das Interface und Python zusammen funktioniert haben. Ohne seinen Rat wäre mir manches Verhalten der Implementierungen unerklärlich geblieben.

Das Interface zwischen C-M.E.S.S. und Python ist das Werk von Nitin Prasad und bildet die Grundlage der Gegenüberstellung. Er hat zusammen mit den anderen Praktikanten den Alltag im Büro vergangenen Sommer um einiges abwechslungsreicher gemacht.

Sehr viel Dank gebührt auch meiner Familie, die mich auf jede erdenkliche Weise während des gesamten Studiums unterstützt und dieses überhaupt erst möglich gemacht hat.

Ganz besonders dankbar bin ich Kirsten Vilbusch. Sie ist nicht nur größte seelische und moralische Stütze und Motivationsquell, sondern hat auch tatkräftig beim Schreiben und vor allem beim Korrekturlesen geholfen. Diese Arbeit hat wohl niemand sonst so oft gelesen wie sie.

1. Einleitung

1.1. Motivation

In vielen Bereichen, wie zum Beispiel der Modell-Ordnungs-Reduktion großer linearer dynamischer Systeme und der linear-quadratischen optimalen Steuerung von parabolischen partiellen Differentialgleichungen [20], finden Matrixgleichungen Anwendung. Wichtig sind hier vor allem die Lyapunov-Gleichung

$$FX + XF^T = -GG^T \quad (1.1)$$

und die verallgemeinerte Lyapunov-Gleichung

$$FXM^T + MXF^T = -GG^T. \quad (1.2)$$

Es sind verschiedene Lösungsverfahren für diese Gleichungen bekannt. Eins davon ist das ADI Verfahren. Hierfür gibt es inzwischen einige Verbesserungen, die es ermöglichen, auch mit großen dünn besetzten Matrizen F und M eine Lösung X zu approximieren, obwohl dieses X im Allgemeinen voll besetzt ist und damit zu großen Speicherbedarf hat, um es explizit zu ermitteln und abzuspeichern. Auch komplexe Arithmetik lässt sich weitestgehend vermeiden.

Es existieren bereits die Implementierungen M.E.S.S. in MATLAB[®] [16] und C-M.E.S.S. in C. Die verhältnismäßig junge Programmiersprache Python gewinnt zunehmend an Bedeutung und wurde mit den Paketen NumPy [17] und SciPy [13] für das wissenschaftliche Rechnen zugänglich gemacht. Damit ist es wünschenswert ein Werkzeug zur Verfügung zu haben, mit dem Lyapunov-Gleichungen auch in Python gelöst werden können. Ziel dieser Bachelorarbeit ist es zum einen das ADI Verfahren mithilfe von NumPy und SciPy zu implementieren. Zum anderen soll diese Implementierung mit einem Interface zwischen Python und C-M.E.S.S. verglichen werden. Dieses Interface stellt die Alternative dar, den ADI Algorithmus in Python zur Verfügung zu stellen.

1.2. Einführendes Beispiel

Ein typisches Beispiel einer parabolischen partiellen Differentialgleichung ist die instationäre Wärmeleitungsgleichung [18, 20]. Sie beschreibt die Temperaturveränderung in einem Körper in Abhängigkeit von der Zeit. In diesem Fall wird das Einheitsquadrat $\Omega = (0, 1) \times (0, 1)$ betrachtet.

$$\begin{aligned} \Delta u - \frac{\partial u}{\partial t} &= 0 \text{ in } \Omega \\ u &= 0 \text{ auf } \partial\Omega \end{aligned} \quad (1.3)$$

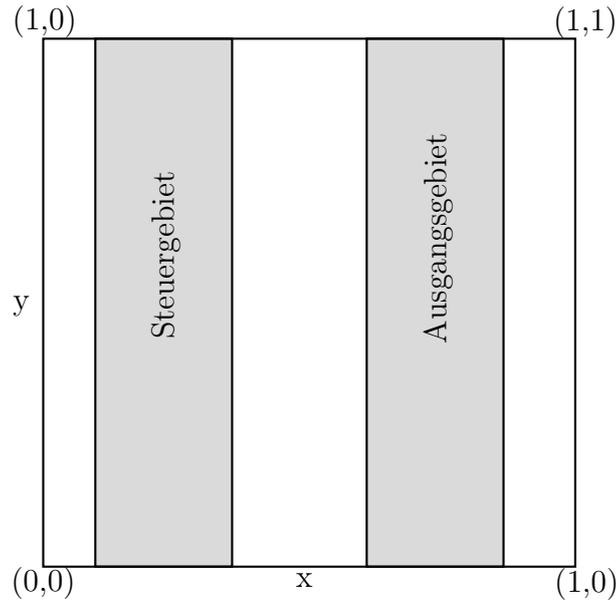


Abbildung 1.1.: Gebietsaufteilung der instationären Wärmeleitung

Mithilfe der Finite-Differenzen-Methode bezüglich des Ortes wird die partielle Differentialgleichung (1.3) mit n Gitterpunkten in jeder Dimension diskretisiert. Wie in Abbildung 1.1 skizziert soll auf dem Bereich $[0.1, 0.3] \times (0, 1)$ die Steuerung gleichmäßig durch eine Steuerfunktion $u(t)$ erfolgen. Für den Ausgang wird die Summe der Funktionswerte auf dem Teilgebiet $[0.7, 0.9] \times (0, 1)$ verwendet. Auf diese Weise entsteht eine dünn besetzte Multi-Diagonalmatrix der Dimension n^2 als Systemmatrix. Diese Semi-Diskretisierung führt zu einem dünn besetzten, linearen zeitinvarianten dynamischen System ohne Massematrix (1.4).

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cu(t) \end{aligned} \tag{1.4}$$

Beim balancierten Abschneiden in der Modellreduktion sind die Lyapunov-Gleichungen mit $F = A, G = B$ sowie mit $F = A^T, G = C^T$ von großen Interesse [8].

$$FX + XF^T = -GG^T$$

Die exakte, eindeutige, konvergente Lösung der Lyapunov-Gleichung ist unter der Voraussetzung, dass $\Lambda(F) \subset \mathbb{C}_-$ [10],

$$X = \lim_{T \rightarrow \infty} \int_0^T e^{Ft} GG^T e^{F^T t} dt.$$

Eine sehr effiziente Methode, diese zu approximieren, ist das ADI Verfahren, welches in dieser Arbeit behandelt wird.

1.3. Kurzfassung

Die vorliegende Bachelorarbeit “Numerisches Lösen großer dünn besetzter Matrixgleichungen in Python” befasst sich mit dem ADI Verfahren zum Lösen der Lyapunov-Gleichung und dessen Implementierung in der Programmiersprache Python.

Nach der Motivation und einem einführenden Beispiel in **Kapitel 1** werden in **Kapitel 2** die Grundlagen des ADI Algorithmus erarbeitet. Bedingungen für die Konvergenz des Verfahrens sind in **Kapitel 3** nachzulesen.

In der Anwendung sind die verwendeten Matrizen häufig sehr groß und dünn besetzt. Die gesuchte Lösung allerdings ist im Allgemeinen voll besetzt. Deshalb sollte vermieden werden diese explizit zu berechnen und zu speichern, welches ab einer gewissen Größe der Eingabematrizen nicht mehr möglich ist. Gleiches gilt für das Residuum der Lyapunov-Gleichung, welches für die Abbruchbedingung des ADI Verfahrens benötigt wird. In **Kapitel 4** werden Verbesserungen des Verfahrens wiederholt, mit denen die Lösung und das Residuum durch Niedrigrangfaktoren approximiert werden können. Außerdem wird eine Möglichkeit aufgezeigt, komplexe Arithmetik zu vermeiden. Komplexe Daten benötigen doppelten Speicherplatz und Operationen mit diesen sind mindestens doppelt so teuer. Diese Verbesserungen werden in **Algorithmus 3** kombiniert.

Die Resultate aus **Kapitel 4** werden in **Kapitel 5** auf die verallgemeinerte Lyapunov-Gleichung übertragen und in **Algorithmus 4** zusammengefasst.

Kapitel 6 beschreibt sowohl die Implementierung der **Algorithmen 3** und **4** in Python als auch das Interface zwischen Python und C-M.E.S.S., der Implementierung in C.

Diese beiden Varianten, den ADI Algorithmus in Python verfügbar zu machen, werden in **Kapitel 7** auf ihre Leistungsfähigkeit untersucht. Dabei wird auch auf die Parallelisierbarkeit eingegangen.

Den Abschluss der Arbeit bilden die Zusammenfassung in **Kapitel 8** und der Ausblick in **Kapitel 9**. Im **Anhang** finden sich die Ergebnisse zu den einzelnen Testdurchläufen aus **Kapitel 7** sowie ein Datenträger mit den Implementierungen und Testdaten.

2. Numerische Verfahren zur Lösung der Lyapunov-Gleichungen

2.1. Das ADI Verfahren

D. Peaceman und H. Rachford führten das ADI Verfahren (Alternating Directions Implicit iteration method) ein, um elliptische und parabolische Differenzialgleichungen zu lösen. [20]

Seien nun $A \in \mathbb{R}^{n \times n}$ eine symmetrische positiv definite Matrix und der Vektor $b \in \mathbb{R}^n$ bekannt.

Gesucht ist ein Vektor $x \in \mathbb{R}^n$ mit

$$Ax = b. \tag{2.1}$$

Angenommen, es gibt Matrizen $H, V \in \mathbb{R}^{n \times n}$ symmetrisch, positiv definit und kommutierend ($HV = VH$) mit $A = H + V$. Das System

$$\begin{aligned} (H + pI)v &= r \\ (V + pI)w &= t \end{aligned}$$

habe für Vektoren $r, t \in \mathbb{R}^n$ und ein geeignetes $p \in \mathbb{R}$ eine Lösung $v, w \in \mathbb{R}^n$, welche sich effizient berechnen lässt.

Die Gleichung (2.1) wird für das ADI Verfahren folgendermaßen umformuliert.

$$\begin{aligned} (H + pI)x &= (pI - V)x + b, \\ (V + pI)x &= (pI - H)x + b \end{aligned} \tag{2.2}$$

Hieraus ergibt sich mit geeigneten Parametern $p_j \in \mathbb{C}$ für $j = 1, 2, \dots$ die Zwei-Schritt-Iterationsvorschrift für das ADI Verfahren.

$$\begin{aligned} (H + p_j I)x_{j-\frac{1}{2}} &= (p_j I - V)x_{j-1} + b, \\ (V + p_j I)x_j &= (p_j I - H)x_{j-\frac{1}{2}} + b \end{aligned} \tag{2.3}$$

Wie schon von Eugene L. Wachspress beschrieben [24], lässt sich dieses Verfahren auch auf die Lyapunov-Gleichungen (1.1) und (1.2) anwenden.

2.2. Das ADI Verfahren für Lyapunov-Gleichungen

Seien $F \in \mathbb{R}^{n \times n}$ und $G \in \mathbb{R}^{n \times m}$. Dann ist die Lyapunov-Gleichung definiert als

$$FX + XF^T = -GG^T \quad (2.4)$$

mit der Lösung $X \in \mathbb{R}^{n \times n}$. (2.4) besitzt eine eindeutige, reelle, symmetrische und positiv semidefinite Lösung, wenn $\lambda(F) \subset \mathbb{C}_-$. Diese ist sogar positiv definit, falls (F, G) steuerbar ist [7].

Auch hier lässt sich die lineare Operation

$$\begin{aligned} \mathcal{A} : \mathbb{R}^{n \times n} &\rightarrow \mathbb{R}^{n \times n} \\ X &\mapsto FX + XF^T \end{aligned}$$

als Summe zweier linearer Operatoren

$$\begin{aligned} \mathcal{H} : \mathbb{R}^{n \times n} &\rightarrow \mathbb{R}^{n \times n} \\ X &\mapsto FX \\ \mathcal{V} : \mathbb{R}^{n \times n} &\rightarrow \mathbb{R}^{n \times n} \\ X &\mapsto XF^T \end{aligned}$$

schreiben, welche kommutieren. [20]

$$\mathcal{H}\mathcal{V}X = \mathcal{H}XF^T = FXF^T = \mathcal{V}FX = \mathcal{V}\mathcal{H}X$$

(2.4) lässt sich analog zu (2.2) in

$$(F + pI)X = -GG^T - X(F^T - pI) \quad (2.5)$$

und

$$X(F^T + pI) = -GG^T - (F - pI)X \quad (2.6)$$

umformen und (2.6) durch komplexes Konjugieren und Transponieren in

$$(F + \bar{p}I)X^T = -GG^T - X^T(F^T - \bar{p}I),$$

da $F \in \mathbb{R}^{n \times n}$, $G \in \mathbb{R}^{n \times m}$, $X \in \mathbb{R}^{n \times n}$ und $p \in \mathbb{C}$.

Daraus folgt die Zwei-Schritt-Iterationsvorschrift, wie in (2.3) als

$$\begin{aligned} (F + p_j I)X_{j-\frac{1}{2}} &= -GG^T - X_{j-1}(F^T - p_j I), \\ (F + \bar{p}_j I)X_j^T &= -GG^T - X_{j-1}^T(F^T - \bar{p}_j I), \end{aligned} \quad (2.7)$$

mit $p_j \in \mathbb{C}_-$ für $j = 1, 2, \dots$

Weiterhin kann (2.7) als Ein-Schritt-Iteration geschrieben werden.

$$\begin{aligned} X_j &= -2 \operatorname{Re}(p_j)(F + p_j I)^{-1}GG^T(F + \bar{p}_j I)^{-T} \\ &\quad + (F^T - \bar{p}_j I)^T(F + p_j I)^{-1}X_{j-1}(F^T - p_j I)(F + \bar{p}_j I)^{-T} \end{aligned} \quad (2.8)$$

Da $(F \pm p_j I)^{\pm 1}$ mit $(F \pm p_i I)^{\pm 1}$ kommutiert, ist (2.8) äquivalent zu

$$\begin{aligned} X_j = & -2 \operatorname{Re}(p_j)(F + p_j I)^{-1} G G^T (F + \bar{p}_j I)^{-T} \\ & + (F + p_j I)^{-1} (F - \bar{p}_j I) X_{j-1} (F - p_j I)^T (F + \bar{p}_j I)^{-T}. \end{aligned} \quad (2.9)$$

An (2.9) ist die Symmetrie von $X_j \in \mathbb{R}^{n \times n}$ abzulesen, wie in [8]. Die Konvergenz dieses Verfahrens hängt von der Wahl der Shift-Parameter $p_j \in \mathbb{C}_-$ ab. Diese Abhängigkeit wird im folgenden Kapitel näher untersucht.

3. Konvergenz des ADI Algorithmus

Bei geeigneter Wahl der Shift-Parameter p_j in (2.9) konvergiert das ADI Verfahren $\lim_{j \rightarrow \infty} X_j = X$ superlinear, wie auch in [20] beziehungsweise [8] nachzulesen ist. Der Fehler im j -ten Iterationsschritt ist definiert als

$$e_j := X_j - X.$$

Wird e_{j-1} nach X umgeformt, in die Lyapunov-Gleichung (2.4) eingesetzt

$$F(X_{j-1} - e_{j-1}) + (X_{j-1} - e_{j-1})F^T = -GG^T$$

und diese dann in die Ein-Schritt-Iterationsvorschrift (2.9), führt dies zu

$$\begin{aligned} X_j = & 2 \operatorname{Re}(p_j)(F + p_j I)^{-1}(F(X_{j-1} - e_{j-1}) + (X_{j-1} - e_{j-1})F^T)(F + \bar{p}_j I)^{-T} \\ & + (F + p_j I)^{-1}(F - \bar{p}_j I)X_{j-1}(F - p_j I)^T(F + \bar{p}_j I)^{-T}. \end{aligned}$$

Hieraus folgt dann mit $2 \operatorname{Re}(p_j) = p_j + \bar{p}_j$, indem die Inversen links und rechts eliminiert werden,

$$\begin{aligned} (F + p_j I)X_j(F + \bar{p}_j I)^T &= \bar{p}_j F X_{j-1} - p_j X_{j-1} F^T + F X_{j-1} F^T + p_j \bar{p}_j X_{j-1} \\ &\quad - (p_j + \bar{p}_j)(F e_{j-1} + e_{j-1} F^T) \\ &= (F + p_j I)(X_{j-1} - e_{j-1})(F + \bar{p}_j I)^T + (F - \bar{p}_j I)e_{j-1}(F - p_j I)^T \\ &\Leftrightarrow \\ e_j &= X_j - X \\ &= X_j - X_{j-1} + e_{j-1} \\ &= (F + p_j I)^{-1}(F - \bar{p}_j I)e_{j-1}(F - p_j I)^T(F + \bar{p}_j I)^{-T} \\ &= R_j e_{j-1} R_j^H \end{aligned}$$

mit

$$R_j := (F + p_j I)^{-1}(F - \bar{p}_j I).$$

Mithilfe dieser für alle j kommutativen R_j kann der Fehler rekursiv dargestellt werden als

$$\begin{aligned} e_j &= \prod_{i=1}^j R_i e_0 \left(\prod_{i=1}^j R_i \right)^H \\ &= \hat{W}_j e_0 \hat{W}_j^H \end{aligned}$$

mit

$$\hat{W}_j := \prod_{i=1}^j R_i.$$

Der Fehler wird mithilfe von $\kappa(\mathcal{P}_j) := \left\| \hat{W}_j \right\|_2$, $\mathcal{P}_j := \{p_1, p_2, \dots, p_j\}$ abgeschätzt.

$$\begin{aligned} \|e_j\|_2 &\leq \left\| \hat{W}_j \right\|_2 \|e_0\|_2 \left\| \hat{W}_j^T \right\|_2 \\ &= \kappa(\mathcal{P}_j)^2 \|e_0\|_2 \end{aligned}$$

Sei F diagonalisierbar mit den Eigenwerten $\Lambda(F) = \{\lambda_1, \dots, \lambda_n\}$ und zugehörigen Eigenvektoren als Spalten von $V \in \mathbb{C}^{n \times n}$. Aus $F = V\Lambda V^{-1}$ mit $\Lambda = \text{diag}\{\lambda_1, \dots, \lambda_n\}$ folgt

$$\begin{aligned} \kappa(\mathcal{P}_j) &= \left\| \prod_{i=1}^j (F + p_j I)^{-1} (F - \bar{p}_j I) \right\|_2 \\ &\leq \|V\|_2 \left\| \prod_{i=1}^j (\Lambda + p_j I)^{-1} (\Lambda - \bar{p}_j I) \right\|_2 \|V^{-1}\|_2 \\ &= \text{cond}(V) \max_{\lambda \in \Lambda(F)} \left| \prod_{i=1}^j \frac{\lambda - \bar{p}_i}{\lambda + p_i} \right| \\ &\leq \max_{\lambda \in \Lambda(F)} \left| \prod_{i=1}^j \frac{\lambda - \bar{p}_i}{\lambda + p_i} \right|. \end{aligned}$$

Das Konvergenzverhalten des ADI Verfahrens ist abhängig von der Wahl der Shift-Parameter. Hierfür muss das folgende Min-Max-Problem gelöst werden.

$$\min_{p_i \in \mathbb{C} : i=1, \dots, j} \kappa(\mathcal{P}_j)$$

In der Regel ist das Spektrum $\Lambda(F)$ nicht bekannt und lässt sich auch nicht effektiv berechnen, wodurch es sehr schwierig ist optimale Parameter p_j zu finden. Alternativ können die Eigenwerte von F mit einer Heuristik approximiert werden. Dies geschieht, indem mit dem Arnoldi-Verfahren zum einen die zu F gehörenden Ritz-Werte berechnet werden, welche nahe der betragsmäßig großen Eigenwerte liegen. Zum anderen sind Approximationen an die Eigenwerte in der Nähe des Ursprungs zu berechnen. Dafür werden die Reziproken der Ritz-Werte benutzt, welche zu F^{-1} gehören. Eine Heuristik wählt die Shift-Parameter aus den beiden Mengen an Ritz-Werten und reziproken Ritz-Werten und ordnet diese dann so, dass Parameter, die in einem ADI Iterationsschritt den Fehler am stärksten reduzieren, möglichst am Anfang stehen. Genaueres zu dieser Heuristik und der entsprechende Algorithmus sind in [18] und [20] zu finden.

Zusätzlich zu der Wahl guter Shift-Parameter sind weitere Verbesserungen des ADI Verfahrens möglich, welche in dem folgenden Kapitel 4 näher betrachtet werden.

4. Niedrigrangfaktoren

4.1. Niedrigrangfaktor der Lösung

Häufig ist die Matrix $F \in \mathbb{R}^{n \times n}$ nur dünn besetzt, wobei n sehr groß ist. Weiterhin gilt oft $G \in \mathbb{R}^{n \times m}$ mit $m \ll n$. Die Lösung der Lyapunov-Gleichung (2.4) $X \in \mathbb{R}^{n \times n}$ sowie die Iterationslösungen $X_j \in \mathbb{R}^{n \times n}$ sind im Allgemeinen voll besetzt. Daher soll vermieden werden, X beziehungsweise X_j explizit zu speichern und zu benutzen. Wie auch in [20] wird in Kapitel 2.2 beobachtet, dass X_j symmetrisch ist. Mithilfe von rechteckigen Matrizen $Z_j \in \mathbb{C}^{n \times j \cdot m}$ als Niedrigrangfaktoren von $X_j = Z_j Z_j^H$ lässt sich der explizite Umgang mit X_j vermeiden. Unter der Annahme $X_0 = \mathbb{0}_{n \times n}$ ergibt sich eine rekursive Iterationsvorschrift, wie in [15].

$$\begin{aligned} Z_0 &= \mathbb{0}_{n \times 1} \\ Z_1 &= \sqrt{-2 \operatorname{Re}(p_1)} (F + p_1 I)^{-1} G \\ Z_j &= \left[\sqrt{-2 \operatorname{Re}(p_j)} (F + p_j I)^{-1} G, (F + p_j I)^{-1} (F - \bar{p}_j I) Z_{j-1} \right], j \geq 2 \end{aligned} \quad (4.1)$$

Hier wird in jedem Iterationsschritt der vorherige Faktor $Z_{j-1} \in \mathbb{C}^{n \times (j-1) \cdot m}$ mit $(F + p_j I)^{-1} (F - \bar{p}_j I)$ multipliziert. Z_j bekommt pro Iterationsschritt m Spalten dazu, wodurch die Berechnung der Iterationslösungen mit jedem Schritt teurer wird. Bei genauer Betrachtung von (4.1) ist zu beobachten, dass es ausreicht, in jedem Iterationsschritt m Spalten hinzuzufügen, ohne Z_{j-1} zu modifizieren.

Seien nun $J \in \mathbb{N}$ die Anzahl der Shift-Parameter und für $j = 1, \dots, J$

$$\begin{aligned} S_j &:= (F + p_j I)^{-1}, \\ T_j &:= (F - \bar{p}_j I), \\ z_j &:= \sqrt{-2 \operatorname{Re}(p_j)} (F + p_j I)^{-1} G, \\ \gamma_j &:= \sqrt{-2 \operatorname{Re}(p_j)}, \\ P_j &:= \frac{\gamma_j}{\gamma_{j+1}} T_{j+1} S_j \\ &= \frac{\sqrt{-2 \operatorname{Re}(p_j)}}{\sqrt{-2 \operatorname{Re}(p_{j+1})}} (F - \bar{p}_{j+1} I) (F + p_j I)^{-1} \\ &= \frac{\sqrt{-2 \operatorname{Re}(p_j)}}{\sqrt{-2 \operatorname{Re}(p_{j+1})}} \left[I - (p_j + \bar{p}_{j+1}) (F + p_j I)^{-1} \right], \end{aligned}$$

wobei T_j und S_r für alle $j, r = 1, \dots, J$ kommutieren. (4.1) kann folgendermaßen umformuliert werden.

$$\begin{aligned} Z_J &= [\gamma_J S_J G, S_J T_J \gamma_{J-1} S_{J-1} G, S_J T_J S_{J-1} T_{J-1} \gamma_{J-2} S_{J-2} G, \dots, \\ &\quad S_J T_J S_{J-1} T_{J-1} \cdots S_2 T_2 \gamma_1 S_1 G] \\ &= [z_J, P_{J-1} z_J, P_{J-2} P_{J-1} z_J, \dots, P_1 P_2 \cdots P_{J-1} z_J] \end{aligned}$$

Da die Reihenfolge der Shift-Parameter keine Rolle spielt, können diese mit umgekehrter Nummerierung angeordnet werden. (4.2) macht deutlich, dass sich das Update der Iterationslösung lediglich auf die letzten m Spalten auswirkt und die ersten $(J-1) \cdot m$ Spalten unverändert bleiben. Somit genügt es, in jeder Iteration Z_j um die neuen m Spalten zu erweitern.

$$\begin{aligned} \hat{P}_j &:= \frac{\gamma_j}{\gamma_{j-1}} S_j T_{j-1} \\ Z_j &= [z_1, \hat{P}_1 z_1, \hat{P}_2 \hat{P}_1 z_1, \dots, \hat{P}_{j-1} \hat{P}_{j-2} \cdots \hat{P}_2 \hat{P}_1 z_1] \\ &= [Z_1, \hat{P}_1 Z_1, \hat{P}_2 Z_2, \dots, \hat{P}_{j-1} Z_{j-1}] \end{aligned} \tag{4.2}$$

Mithilfe von (4.1) und (4.2) kann nun das ADI Verfahren in einen Algorithmus gefasst werden.

Algorithmus 1: Low-Rank Cholesky factor ADI iteration (LRCF-ADI-v1)

Eingabe : $F \in \mathbb{R}^{n \times n}, G \in \mathbb{R}^{n \times m}$, welche $FX + XF^T = -GG^T$ definieren und

Shift-Parameter $\{p_1, \dots, p_{J_{max}}\} \subset \mathbb{C}_-$

Ausgabe : $Z = Z_{J_{max}} \in \mathbb{C}^{n \times J_{max} \cdot m}$ mit $ZZ^H \approx X$

- 1 $V_1 = (F + p_1 I)^{-1} G$
 - 2 $Z_1 = \sqrt{-2 \operatorname{Re}(p_1)} V_1$
 - 3 **for** $j = 2$ **to** J_{max} **do**
 - 4 $V_j = V_{j-1} - (p_j + \overline{p_{j-1}})(F + p_j I)^{-1} V_{j-1}$
 - 5 $Z_j = [Z_{j-1}, \sqrt{-2 \operatorname{Re}(p_j)} V_j]$
 - 6 **end**
-

In diesem Algorithmus ist es nicht nötig, die Iterationslösung $X_j \in \mathbb{R}^{n \times n}$ zu speichern. Stattdessen wird lediglich $Z_j \in \mathbb{C}^{n \times j \cdot m}$ gespeichert, welches weniger aufwändig ist, da $m \ll n$. Um den Speicheraufwand noch weiter zu senken, kann das Verfahren zusätzlich verbessert werden, indem sichergestellt wird, dass die Niedrigrangfaktoren $Z_j \in \mathbb{R}^{n \times j \cdot m}$ ausschließlich reell sind.

4.2. Vermeidung komplexer Lösungsfaktoren

Wie schon in Kapitel 2.2 erwähnt, besitzt die Lyapunov-Gleichung (2.4) eine reelle Lösung. Allerdings können die Shift-Parameter des ADI Verfahrens auch komplex sein. Dies hat zur Folge, dass für ein $p_j \in \mathbb{C}_-$ in **Algorithmus 1** sowohl V_j als auch Z_j komplexwertig sind. Zum einen muss ein System mit $(F + p_j I)^{-1} \in \mathbb{C}^{n \times n}$ in komplexer Arithmetik gelöst werden, was zusätzlichen Aufwand bedeutet. Zum anderen haben V_j

und Z_j doppelten Speicherbedarf gegenüber einer reellen Lösung.

Um dies zu vermeiden, sind weitere Voraussetzungen an die Shift-Parameter $p_j \in \mathbb{C}_-$ nötig. Diese heißen **geeignet**, falls für alle $j = 1, \dots, J_{max}$ entweder $p_j \in \mathbb{R}_-$ oder komplexe Shifts nur in Paaren mit $p_j \in \mathbb{C}_-$, $p_{j+1} = \overline{p_j}$ auftreten.

Ist solch ein Paar komplexer Shifts gegeben, gilt mit $\beta_j := \frac{\operatorname{Re}(p_j)}{\operatorname{Im}(p_j)} \in \mathbb{R}$ für $\hat{V}_j := \sqrt{-2 \operatorname{Re}(p_j)} V_j \in \mathbb{C}^{n \times m}$ und $\hat{V}_{j+1} := \sqrt{-2 \operatorname{Re}(p_{j+1})} V_{j+1} \in \mathbb{C}^{n \times m}$

$$\hat{V}_{j+1} = \overline{\hat{V}_j} + 2\beta_j \operatorname{Im}(\hat{V}_j). \quad (4.3)$$

Die folgenden Resultate sowie der Beweis von (4.3) finden sich auch in [6].

Sei nun

$$Z_{j+1} = [\hat{V}_1, \dots, \hat{V}_{j-1}, \hat{V}_j, \hat{V}_{j+1}]$$

der Lösungsfaktor der $(j+1)$ -ten Iteration und

$$\hat{Z} := [\hat{V}_j, \hat{V}_{j+1}] = [\hat{V}_j, \overline{\hat{V}_j} + 2\beta_j \operatorname{Im}(\hat{V}_j)] \in \mathbb{C}^{n \times 2m}.$$

\hat{Z} lässt sich mithilfe von

$$\hat{T} := \begin{bmatrix} I & I \\ iI & (2\beta_j - i)I \end{bmatrix} \in \mathbb{C}^{2m \times 2m}$$

darstellen als

$$\hat{Z} = [\operatorname{Re}(\hat{V}_j) \quad \operatorname{Im}(\hat{V}_j)] \cdot \hat{T}.$$

Sei

$$\hat{F} := \begin{bmatrix} 2 & 2\beta_j \\ 2\beta_j & 2 + 4\beta_j^2 \end{bmatrix} \in \mathbb{R}^{2 \times 2}.$$

Um die Matrix $\hat{T}\hat{T}^H$ zu untersuchen, kann diese als Kronecker-Produkt geschrieben werden.

$$\hat{T}\hat{T}^H = \begin{bmatrix} 2I & 2\beta_j I \\ 2\beta_j I & (2 + 4\beta_j^2)I \end{bmatrix} = \hat{F} \otimes I \in \mathbb{R}^{2m \times 2m}$$

Die Eigenwerte von \hat{F} sind

$$\Lambda(\hat{F}) = \left\{ 2 \left(\beta_j^2 + 1 \pm \beta_j \sqrt{\beta_j^2 + 1} \right) \right\},$$

welche echt größer als Null sind, da

$$(\beta_j^2 + 1)^2 = \beta_j^2(\beta_j^2 + 1) + \underbrace{\beta_j^2 + 1}_{>0} > \left(\beta_j \sqrt{\beta_j^2 + 1} \right)^2.$$

Folglich ist \hat{F} reell, symmetrisch, positiv definit. Aufgrund der Eigenschaften des Kronecker-Produkts gilt das Gleiche für $\hat{T}\hat{T}^H$. Damit hat $\hat{T}\hat{T}^H$ eine eindeutige Cholesky-

Zerlegung.

$$L := \sqrt{2} \begin{bmatrix} I & 0_{m \times m} \\ \beta_j I & \sqrt{\beta_j^2 + 1} I \end{bmatrix} \in \mathbb{R}^{2m \times 2m}$$

$$\hat{T} \hat{T}^H = LL^T \in \mathbb{R}^{2m \times 2m}$$

Letzteres zeigt, dass

$$\begin{aligned} \hat{Z} \hat{Z}^H &= [\operatorname{Re}(\hat{V}_j), \operatorname{Im}(\hat{V}_j)] \cdot \hat{T} \hat{T}^H \cdot [\operatorname{Re}(\hat{V}_j), \operatorname{Im}(\hat{V}_j)]^H \\ &= [\operatorname{Re}(\hat{V}_j), \operatorname{Im}(\hat{V}_j)] \cdot LL^T \cdot [\operatorname{Re}(\hat{V}_j), \operatorname{Im}(\hat{V}_j)]^T \in \mathbb{R}^{n \times n} \end{aligned}$$

reell ist und auch mit reellen Faktoren berechnet werden kann.

$$\begin{aligned} \alpha_j &:= 2\sqrt{-\operatorname{Re}(p_j)} \\ \tilde{Z} &:= [\operatorname{Re}(\hat{V}_j), \operatorname{Im}(\hat{V}_j)] \cdot L \\ &= \sqrt{2} \left[\operatorname{Re}(\hat{V}_j) + \beta_j \operatorname{Im}(\hat{V}_j), \sqrt{\beta_j^2 + 1} \operatorname{Im}(\hat{V}_j) \right] \\ &= \alpha_j \left[\operatorname{Re}(V_j) + \beta_j \operatorname{Im}(V_j), \sqrt{\beta_j^2 + 1} \operatorname{Im}(V_j) \right] \in \mathbb{R}^{n \times 2m} \end{aligned} \tag{4.4}$$

Mit (4.4) kann **Algorithmus 1** so umformuliert werden, dass Z_j in jedem Iterationsschritt reell ist.

Dadurch wird der Speicheraufwand für den Lösungsfaktor halbiert. Für komplexe Shift-Parameter muss zwar weiterhin ein komplexes System mit $(F + p_j I)^{-1} \in \mathbb{C}^{n \times n}$ gelöst werden. Mit dem daraus resultierenden V_j kann jedoch in einem Doppelschritt V_{j+1} berechnet werden, ohne ein weiteres Mal lösen zu müssen, wodurch der Rechenaufwand hierfür eingespart wird.

Ein Problem ist, dass für den folgenden Iterationsschritt das eventuell komplexe V_{j+1} aus Gleichung (4.3) benötigt wird. Somit ist auch das nächste zu lösende System komplex, selbst wenn der Shift-Parameter reell ist. Obwohl die Matrix V_j in Zeile 6 von **Algorithmus 2** theoretisch reell ist, kann es passieren, dass dieser durch numerische Fehler einen Imaginärteil hat, welcher betragsmäßig nahe der Maschinengenauigkeit ist. Deswegen wird in dieser Zeile sichergestellt, dass V_j eine reelle Matrix ist. Auch diese Schwierigkeiten können mithilfe von Niedrigrangfaktoren des Residuums umgangen werden, wie im folgenden Abschnitt 4.3 zu sehen ist.

Algorithmus 2: Low-Rank Cholesky factor ADI iteration (LRCF-ADI-v2)

Eingabe : $F \in \mathbb{R}^{n \times n}$, $G \in \mathbb{R}^{n \times m}$, welche $FX + XF^T = -GG^T$ definieren und geeignete Shift-Parameter $\{p_1, \dots, p_{J_{max}}\} \subset \mathbb{C}_-$

Ausgabe : $Z = Z_{J_{max}} \in \mathbb{R}^{n \times J_{max} \cdot m}$ mit $ZZ^T \approx X$

```

1  $V_1 = (F + p_1 I)^{-1} G$ 
2  $Z = \sqrt{-2 \operatorname{Re}(p_1)} V_1$ 
3 for  $j = 2$  to  $J_{max}$  do
4    $V_j = V_{j-1} - (p_j + \overline{p_{j-1}})(F + p_j I)^{-1} V_{j-1}$ 
5   if  $\operatorname{Im}(p_j) = 0$  then
6      $V_j = \operatorname{Re}(V_j)$ 
7      $Z = [Z, \sqrt{-2 \operatorname{Re}(p_j)} V_j]$ 
8   else
9      $\alpha = 2\sqrt{-\operatorname{Re}(p_j)}$ 
10     $\beta = \frac{\operatorname{Re}(p_j)}{\operatorname{Im}(p_j)}$ 
11     $V_{j+1} = \overline{V_j} + 2\beta \operatorname{Im}(V_j)$ 
12     $Z = [Z, \alpha (\operatorname{Re}(V_j) + \beta \operatorname{Im}(V_j)), \alpha \sqrt{\beta^2 + 1} \operatorname{Im}(V_j)]$ 
13     $j = j + 1$ 
14  end
15 end

```

4.3. Niedrigrangfaktor des Residuums

Als Abbruchkriterium für das ADI Verfahren kann die Norm des Residuums (4.5) der Lyapunov-Gleichung (2.4) benutzt werden.

$$\mathcal{L}(X_j) := FX_j + X_j F^T + GG^T \in \mathbb{R}^{n \times n} \quad (4.5)$$

Wenn diese kleiner als eine vorher definierte Toleranz ist, wird der Algorithmus abgebrochen. Wie schon in Kapitel 4.1 festgestellt, sind die Iterationslösungen X_j im Allgemeinen voll besetzt und somit auch die Residuen $\mathcal{L}(X_j)$. Alleine der Rechen- und Speicheraufwand dafür wäre schon zu groß. Wie in [5], nur mit komplexen Shift-Parametern $p_j \in \mathbb{C}_-$, ist zu zeigen, dass auch $\mathcal{L}(X_j)$ symmetrisch ist und durch einen Niedrigrangfaktor deutlich effizienter berechnet werden kann.

In Kapitel 3 wird bereits gezeigt, dass

$$e_j = \hat{W}_j e_0 \hat{W}_j^H = \hat{W}_j (X_0 - X) \hat{W}_j^H, \quad (4.6)$$

mit den Bezeichnungen

$$\begin{aligned} e_j &:= X_j - X, \\ R_j &:= (F + p_j I)^{-1}(F - \overline{p}_j I), \\ \hat{W}_j &:= \prod_{i=1}^j (F + p_i I)^{-1}(F - \overline{p}_i I). \end{aligned}$$

Das Einsetzen von (4.6) in (4.5) ergibt

$$\mathcal{L}(X_j) = F(X_j - X) + (X_j - X)F^T = F\hat{W}_j e_0 \hat{W}_j^H + \hat{W}_j e_0 \hat{W}_j^H F^T.$$

Zusammen damit, dass F und \hat{W}_j kommutieren und $X_0 = \mathbb{0}_{n \times n}$, folgt

$$\mathcal{L}(X_j) = \hat{W}_j G G^T \hat{W}_j^H.$$

Somit ist mit $W_0 := G$

$$W_j := (F + p_j I)^{-1}(F - \overline{p}_j I)W_{j-1} \in \mathbb{C}^{n \times m} \quad (4.7)$$

ein Niedrigrangfaktor des Residuums definiert, welcher sich rekursiv berechnen lässt. Mit diesem lässt sich über ein inneres Produkt die Norm des Residuums sehr effizient ermitteln. Zu beachten ist $m \ll n$.

$$\|\mathcal{L}(X_j)\|_2 = \|W_j W_j^H\|_2 = \underbrace{\|W_j^H W_j\|_2}_{\in \mathbb{C}^{m \times m}} \quad (4.8)$$

Dieser Niedrigrangfaktor kann sogar genutzt werden, um V_j in **Algorithmus 1** zu berechnen.

$$\begin{aligned} V_j &= [I - (p_j + \overline{p}_{j-1})(F + p_j I)^{-1}] V_{j-1} \\ &= (F - \overline{p}_{j-1} I)(F + p_j I)^{-1} V_{j-1} \\ &= (F + p_j I)^{-1}(F - \overline{p}_{j-1} I)(F + p_{j-1} I)^{-1}(F - \overline{p}_{j-2} I) V_{j-1} \\ &= (F + p_j I)^{-1} R_{j-1}(F - \overline{p}_{j-2} I) V_{j-1} \\ &= (F + p_j I)^{-1} \left(\prod_{i=1}^{j-1} (F + p_i I)^{-1}(F - \overline{p}_i I) \right) G \\ &= (F + p_j I)^{-1} W_{j-1} \end{aligned} \quad (4.9)$$

Weiterhin kann W_j effizient mithilfe von V_j aufdatiert werden.

$$\begin{aligned} W_j &= (F - \overline{p}_j I)(F + p_j I)^{-1} W_{j-1} \\ &= [I - (p_j + \overline{p}_j)(F + p_j I)^{-1}] W_{j-1} \\ &= W_{j-1} - 2 \operatorname{Re}(p_j) V_j \end{aligned} \quad (4.10)$$

Wie schon in Kapitel 4.2 kann der Shift-Parameter $p_j \in \mathbb{C}_-$ komplex sein. Auch hier lässt sich vermeiden, dass der Niedrigrangfaktor W_j als komplexe Matrix gespeichert

und komplexe Arithmetik angewandt werden muss.

4.4. Vermeidung komplexer Arithmetik beim Niedrigrangfaktor des Residuums

Der Doppelschritt wie in Kapitel 4.2 kann auch für den Niedrigrangfaktor des Residuums durchgeführt werden [5]. Wird Gleichung (4.3) in die Berechnungsformel (4.10) für W_{j+1} eingesetzt, folgt

$$\begin{aligned} W_{j+1} &= W_j - 2 \operatorname{Re}(p_j) V_{j+1} \\ &= W_{j-1} - 2 \operatorname{Re}(p_j) (\operatorname{Re}(V_j) + i \operatorname{Im}(V_j) + \operatorname{Re}(V_j) - i \operatorname{Im}(V_j) + 2\beta_j \operatorname{Im}(V_j)) \quad (4.11) \\ &= W_{j-1} - 4 \operatorname{Re}(p_j) (\operatorname{Re}(V_j) + \beta_j \operatorname{Im}(V_j)). \end{aligned}$$

Mithilfe von (4.11) lässt sich **Algorithmus 2** erweitern.

Algorithmus 3: Low-Rank Cholesky factor ADI iteration (LRCF-ADI-v3)

Eingabe : $F \in \mathbb{R}^{n \times n}$, $G \in \mathbb{R}^{n \times m}$, welche $FX + XF^T = -GG^T$ definieren und geeignete Shift-Parameter $\{p_1, \dots, p_{J_{\max}}\} \subset \mathbb{C}_-$

Ausgabe : $Z = Z_{J_{\max}} \in \mathbb{R}^{n \times J_{\max} \cdot m}$ mit $ZZ^T \approx X$

```

1   $Z = []$ 
2   $W_0 = G$ 
3   $j = 1$ 
4  while  $j \leq J_{\max}$  and  $\|W_{j-1}^T W_{j-1}\|_2 > tol$  do
5       $V_j = (F + p_j I)^{-1} W_{j-1}$ 
6      if  $\operatorname{Im}(p_j) = 0$  then
7           $Z = [Z, \sqrt{-2 \operatorname{Re}(p_j)} V_j]$ 
8           $W_j = W_{j-1} - 2 \operatorname{Re}(p_j) V_j$ 
9           $j = j + 1$ 
10     else
11          $\alpha := 2\sqrt{-\operatorname{Re}(p_j)}$ 
12          $\beta = \frac{\operatorname{Re}(p_j)}{\operatorname{Im}(p_j)}$ 
13          $Z = [Z, \alpha (\operatorname{Re}(V_j) + \beta \operatorname{Im}(V_j)), \alpha \sqrt{\beta^2 + 1} \operatorname{Im}(V_j)]$ 
14          $W_{j+1} = W_{j-1} + \alpha^2 (\operatorname{Re}(V_j) + \beta \operatorname{Im}(V_j))$ 
15          $j = j + 2$ 
16     end
17 end

```

Hier wird über den Niedrigrangfaktor des Residuums effizient ein Abbruchkriterium ermittelt. Im Gegensatz zu **Algorithmus 2** muss nach einem Doppelschritt kein komplexes System mehr gelöst werden, falls der zugehörige Shift-Parameter reell ist.

5. Verallgemeinerte Lyapunov-Gleichungen

Algorithmus 3 kann erweitert werden, um verallgemeinerte Lyapunov-Gleichungen zu lösen [5].

$$FXM^T + MXF^T = -GG^T \quad (5.1)$$

Gleichung (5.1) ist äquivalent zu

$$\begin{aligned} M^{-1}FX + XF^T M^{-T} &= -M^{-1}GG^T M^{-T} \\ \Leftrightarrow \tilde{F}X + X\tilde{F}^T &= -\tilde{G}\tilde{G}^T. \\ \tilde{F} &:= M^{-1}F \\ \tilde{G} &:= M^{-1}G \end{aligned}$$

In **Algorithmus 1** ergibt sich

$$\begin{aligned} V_j &= V_{j-1} - (p_j + \overline{p_{j-1}})(\tilde{F} + p_j I)^{-1}V_{j-1} \\ &= V_{j-1} - (p_j + \overline{p_{j-1}})(M^{-1}(F + p_j M))^{-1}V_{j-1} \\ &= V_{j-1} - (p_j + \overline{p_{j-1}})(F + p_j M)^{-1}MV_{j-1}, \end{aligned}$$

beziehungsweise für **Algorithmus 3** mit $\tilde{W}_j := MW_j$

$$\begin{aligned} V_j &= (\tilde{F} + p_j I)^{-1}W_{j-1} = (F + p_j M)^{-1}\tilde{W}_{j-1} \\ \tilde{W}_j &= MW_{j-1} - 2\operatorname{Re}(p_j)MV_j = \tilde{W}_{j-1} - 2\operatorname{Re}(p_j)MV_j. \end{aligned} \quad (5.2)$$

Mithilfe von (5.2) kann **Algorithmus 3** für verallgemeinerte Lyapunov-Gleichungen umformuliert werden.

Algorithmus 4: Generalized Low-Rank Cholesky factor ADI iteration (G-LRCF-ADI)

Eingabe : $F \in \mathbb{R}^{n \times n}$, $M \in \mathbb{R}^{n \times n}$, $G \in \mathbb{R}^{n \times m}$, welche $FXM^T + MXF^T = -GG^T$ definieren und geeignete Shift-Parameter $\{p_1, \dots, p_{J_{max}}\} \subset \mathbb{C}_-$

Ausgabe : $Z = Z_{J_{max}} \in \mathbb{R}^{n \times J_{max} \cdot m}$ mit $ZZ^T \approx X$

```

1  $Z = []$ 
2  $W_0 = G$ 
3  $j = 1$ 
4 while  $j \leq J_{max}$  and  $\|W_{j-1}^T W_{j-1}\|_2 > tol$  do
5    $V_j = (F + p_j M)^{-1} W_{j-1}$ 
6   if  $\text{Im}(p_j) = 0$  then
7      $Z = [Z, \sqrt{-2 \text{Re}(p_j)} V_j]$ 
8      $W_j = W_{j-1} - 2 \text{Re}(p_j) M V_j$ 
9      $j = j + 1$ 
10  else
11     $\alpha := 2\sqrt{-\text{Re}(p_j)}$ 
12     $\beta = \frac{\text{Re}(p_j)}{\text{Im}(p_j)}$ 
13     $Z = [Z, \alpha (\text{Re}(V_j) + \beta \text{Im}(V_j)), \alpha \sqrt{\beta^2 + 1} \text{Im}(V_j)]$ 
14     $W_{j+1} = W_{j-1} + \alpha^2 M (\text{Re}(V_j) + \beta \text{Im}(V_j))$ 
15     $j = j + 2$ 
16  end
17 end

```

6. Implementierung in Python

Es bestehen im wesentlichen zwei Möglichkeiten, das ADI Verfahren in Python umzusetzen. Zum einen können alle wichtigen Routinen direkt in Python neu geschrieben werden (Python-M.E.S.S.), wie im folgenden Kapitel 6.1 beschrieben. Zum anderen können die bereits in C-M.E.S.S. bestehenden Routinen mithilfe eines Interface genutzt werden, welches in Kapitel 6.2 dargelegt ist.

6.1. Direkte Implementierung in Python

Ein Ziel dieser Bachelorarbeit ist es das ADI Verfahren in der Programmiersprache Python [21] zu implementieren. Hierfür wird hauptsächlich das Modul SciPy [13] verwendet.

SciPy basiert auf dem Modul NumPy [17] und stellt einen schnellen Array-Datentyp, Matrix-Datentypen für voll und dünn besetzte Matrizen sowie alle benötigten Lineare Algebra Operationen zur Verfügung. Dafür greift es unter anderem auf BLAS [1], LAPACK [2] und UMFPACK [9] zurück.

Die **Algorithmen 3** und **4** werden so implementiert, dass sie sowohl Python 2 als auch Python 3 unterstützen. In Tabelle 6.1 sind die verwendeten Versionen aufgelistet (kompiliert mit gcc 4.6.3).

Name	Version
Python 2	2.7.5
Python 3	3.2.3
NumPy(Python 2)	1.7.0
NumPy(Python 3)	1.6.1
SciPy(Python 2)	0.13.0
SciPy(Python 3)	0.9.0
BLAS	Intel [®] MKL 11.0
LAPACK	Intel [®] MKL 11.0
UMFPACK	5.5.2
M.E.S.S. Revision	3902

Tabelle 6.1.: Software Versionen

Der Aufbau und die Benennung von Variablen und Funktionen der Implementierung orientieren sich an den bestehenden Versionen C-M.E.S.S. in der Programmiersprache C und hauptsächlich an MATLAB-M.E.S.S..

Um Ressourcen zu sparen, werden, anders als üblich, in keiner der Python-Funktionen ganze Module wie SciPy oder Submodule von SciPy geladen. Stattdessen werden nur

die Elemente importiert, die in der jeweiligen Funktion benötigt werden, um den Namespace¹ so klein wie möglich zu halten.

Beim Lösen dünn besetzter Systeme mit mehreren rechten Seiten, wie in **Algorithmus 3**, Zeile 5, stellt sich das Problem, dass das Submodul `scipy.sparse` keine passende Funktion anbietet. Mehrere rechte Seiten und eine dünn besetzte Matrix kann `scipy.sparse.linalg.spsolve` zwar handhaben, erwartet aber auch, dass die Rechte-Seite-Matrix sowie die Lösung dünn besetzt sind [22]. Da dies beim ADI Verfahren nicht der Fall ist, wird direkt **UMFPACK** benutzt. `scipy.sparse.linalg.dsolve.factorized` ist ein Wrapper um `scipy.sparse.linalg.umfpack.UmfpackContext`. Es berechnet eine LU-Zerlegung, speichert diese und hat als Rückgabewert eine Funktion, die diese Zerlegung zum Lösen des Systems benutzt. In **Python-M.E.S.S.** wird einheitlich `UmfpackContext` direkt benutzt ohne `factorized`. Dies bietet den Vorteil, dass das System auch in transponierter Form gelöst werden kann, welches sich in Tests als schneller erwiesen hat. `factorized` ist dazu nicht in der Lage, weshalb die Matrix vorher transponiert werden müsste, welches zusätzlichen Aufwand bedeutet. Mithilfe der zurückgegebenen Löserfunktion kann über die einzelnen rechten Seiten iteriert werden.

Ein weiterer zu beachtender Aspekt sind die verschiedenen Speicherformate für Matrizen, die SciPy bereitstellt. Bei voll besetzten Matrizen wird das Fortran-übliche “column-major”-Speicherformat genutzt, da den Lineare Algebra Funktionen auf Fortran basierende Routinen wie **BLAS** und **LAPACK** zugrunde liegen. Auch dies hat sich in Tests als schneller erwiesen als das C-typische “row-major”-Format. Hierfür wird einheitlich die bereitgestellte Matrix-Klasse `matrix` von SciPy verwendet. Bei anderen Klassen, wie dem mehrdimensionalen Array `ndarray`, treten Fehler mit dem `shape`-Attribut auf, welches alle Arrayklassen besitzen.

Der Zugriff auf eine einzelne Zeile oder Spalte einer als `ndarray` gespeicherten Matrix liefert einen eindimensionalen Vektor (`shape = (n,)`). Weitere Berechnungen, beispielsweise ein Matrix-Vektor-Produkt mit diesem sind fehlerhaft. Bei einer als `matrix` gespeicherten Matrix hingegen liefert der Zugriff auf eine einzelne Zeile oder Spalte einen zweidimensionalen Vektor (`shape = (n, 1)` oder `shape = (1, n)`). Weitere Operationen mit diesem produzieren keine solchen Fehler. Vergleiche auch mit dem Beispiel in Abbildung 6.1 beziehungsweise 6.2.

Andererseits muss die Löserfunktion, die `UmfpackContext` zurück gibt, einen eindimensionalen Vektor(`ndarray`) als rechte Seite bekommen und gibt auch einen solchen als Lösung aus. Dafür ist jeweils eine Änderung des `shape`-Attributes notwendig. `matrix` ist eine Unterklasse von `ndarray`. Trotzdem verwenden beide nicht die selbe `dot`-Funktion für das Matrix-Matrix- oder Matrix-Vektor-Produkt. Ein Problem dieser `dot`-Funktionen ist, dass sie nicht zuverlässig die bereitgestellten **BLAS**-Routinen nutzen, vor allem wenn mehrere Threads zur Verfügung stehen. Aber auch das Interface auf die **BLAS**-Routinen in `scipy.linalg.blas.fblas` ist nicht leistungsfähiger.

¹Ein mapping aller “Namen” von Variablen, Funktionen, Modulen, ... zu den entsprechenden Objekten

```

from scipy import rand, mat
from scipy.sparse import rand as sprand
from scipy.linalg import norm

n = 100
d = .05
# zufaellige duenn besetzte 100 x 100 Matrix
F = sprand(n, n, d)
# zufaelliger Vektor der Laenge 100 als ndarray
b = rand(n,1)
print "b: = " , b.shape
print "b[ : , 0].shape = " , b[ : , 0].shape

# b von sich selbst subtrahieren
print "||b - b[ : , 0]|| =" , norm(b - b[ : ,0])

# Multiplikation mit einer Matrix
print "||F * b - F * b[ : , 0]|| =" ,
print norm(F.dot(b) - F.dot(b[ : , 0]))

# gleicher Vektor als matrix
b = mat(rand(n,1))
print "b: = " , b.shape
print "b[ : , 0].shape = " , b[ : , 0].shape

# b von sich selbst subtrahieren
print "||b - b[ : , 0]|| =" , norm(b - b[ : ,0])

# Multiplikation mit einer Matrix
print "||F * b - F * b[ : , 0]|| =" ,
print norm(F.dot(b) - F.dot(b[ : , 0]))

```

Abbildung 6.1.: Beispielcode zur Demonstration des Fehlers im Zusammenhang mit dem shape Attribut

```

b: = (100, 1)
b[ : , 0].shape = (100,)
||b - b[ : , 0]|| = 45.257903656
||F * b - F * b[ : , 0]|| = 98.0907729406
b: = (100, 1)
b[ : , 0].shape = (100, 1)
||b - b[ : , 0]|| = 0.0
||F * b - F * b[ : , 0]|| = 0.0

```

Abbildung 6.2.: Ausgabe von 6.1

Dünn besetzte Matrizen werden im “Compressed Sparse Column”-Format gespeichert, da `UMFPACK` dieses benötigt.

Die benötigten Shift-Parameter werden mithilfe der heuristischen Methode berechnet, welche in Kapitel 3 vorgeschlagen wird. Dabei muss mit F beziehungsweise im Fall der verallgemeinerten Lyapunov-Gleichung (5.1) auch mit M gelöst werden. Hierfür werden die LU-Zerlegungen vorab berechnet, gespeichert und bei jedem Lösen wiederverwendet.

Ein Testdurchlauf von **Algorithmus 4** findet sich in `DEMOS/MSD_TripleChainNaive.py`. Diesem Beispiel liegt das Triple Chain Oscillator Modell aus [23] zugrunde, welches auch für die numerischen Tests in Kapitel 7 verwendet wird.

6.2. Interface zwischen Python und C-M.E.S.S.

Eine weitere Möglichkeit, das ADI Verfahren in Python zu implementieren, ist ein Interface zu nutzen, um die bestehende Version von C-M.E.S.S. in der Programmiersprache C auszuführen.

Python stellt eine Schnittstelle zur Verfügung, mit der Module um C-Programme erweiterbar sind [11]. Es können aus Python heraus C-Funktionen gerufen, Python-Objekte in C-Datentypen konvertiert werden und anders herum. Ein solches Interface hält Funktionen bereit, um alle benötigten Datentypen wie Matrizen, Vektoren und Strukturen, die die Optionen für das ADI Verfahren enthalten, hin und zurück zu konvertieren sowie die C-M.E.S.S. Methoden zu starten. Bei diesem Konvertieren von Python zu C müssen allerdings Kopien aller Daten angelegt werden. Es bietet eine einheitliche Schnittstelle, mit der der ADI Algorithmus analog zu der direkten Implementierung in Python und MATLAB-M.E.S.S. angewandt werden kann. Für eine genaue Beschreibung siehe [3].

Dieses Interface soll nun mit der direkten Implementierung aus Kapitel 6.1 verglichen werden. Dazu wird sichergestellt, dass beide Versionen genau die selben Bibliotheken wie BLAS, LAPACK und UMFPACK im Hintergrund verwenden.

7. Experimenteller Vergleich zu C-M.E.S.S.

Die beiden Implementierungsvarianten aus Kapitel 6.1 und 6.2 werden mit verschiedenen Eingabedaten auf folgendem System getestet.

- 2 Intel[®]Xeon[®]E5-2690 CPUs mit 2.9 GHz Taktung. (8 Kerne pro CPU, Sandy-Bridge-EP Mikroarchitektur)
 - 32 KB L1 Cache (jeweils für Daten und Instruktionen)
 - 256 KB L2 Cache
 - 20 MB L3 Cache
- 32 GB DDR3(1.6 GHz) Hauptspeicher, partitioniert in zwei ccNUMA Knoten, 16 GB pro CPU
- Ubuntu 12.04 64bit, Server Edition

Alle Testdurchläufe werden sowohl auf einem einzelnen Kern als auch auf mehreren Kernen der CPUs durchgeführt.

7.1. Sequentielle Tests

In dem folgenden Test wird die Lösung der Lyapunov-Gleichung (2.4) approximiert. Die symmetrische Matrix $F \in \mathbb{R}^n$ wird mithilfe der zweidimensionalen FDM Diskretisierung aus Kapitel 1.2 in den Größen $n = 400$ bis maximal 250 000 generiert.

Die dafür benötigte Funktion `generate_demo_sym` findet sich in der Datei `generate_fdm_2d_demo.py` und stammt aus [18]. Das ADI Verfahren wird abgebrochen, wenn die euklidische Norm des Residuums kleiner als 10^{-10} ist oder nach maximal 100 Iterationen. Es werden 25 Shift-Parameter durch die heuristische Methode [18] aus Kapitel 3 berechnet, die 50 Arnoldi-Schritte mit F und 25 mit F^{-1} durchführt. Um zuverlässige Resultate zu bekommen, wird für jede Matrix F mit $n < 10\,000$ der Durchlauf 20 mal wiederholt. Bei einer Größe von $n > 10\,000$ gibt es 10 Wiederholungen und für $n > 40\,000$ sind es 5.

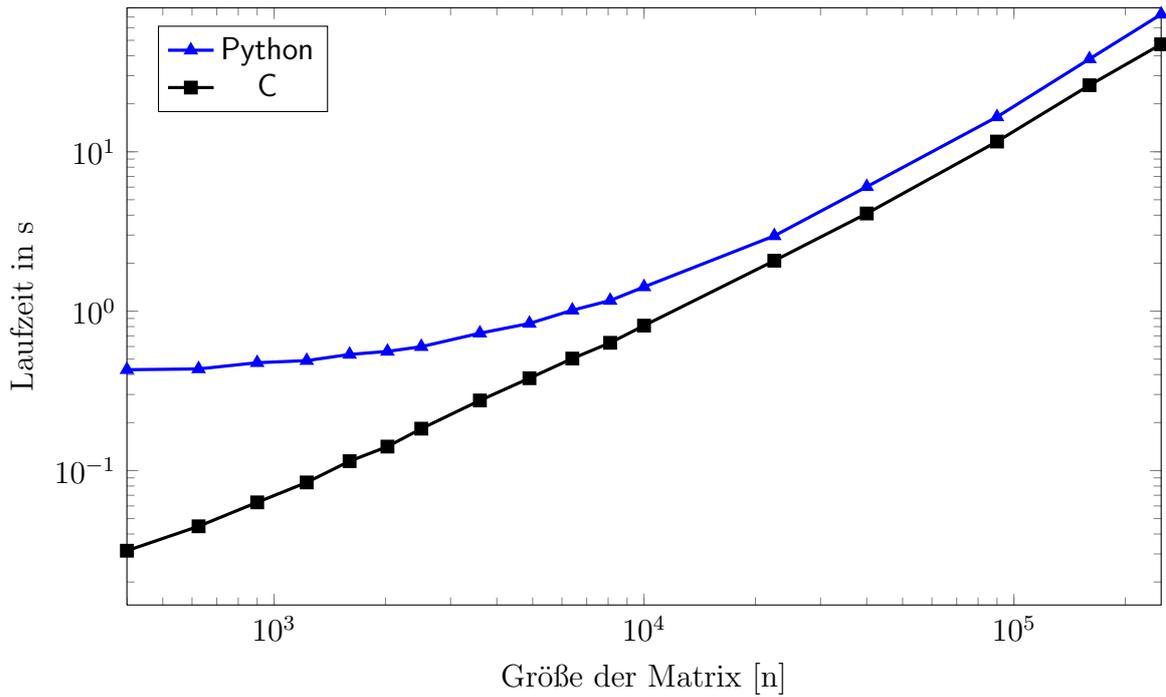


Abbildung 7.1.: Laufzeit von sequentieller Shift-Parameter Berechnung und LRCF ADI für `fdm_2d_sym`: Python vs. C

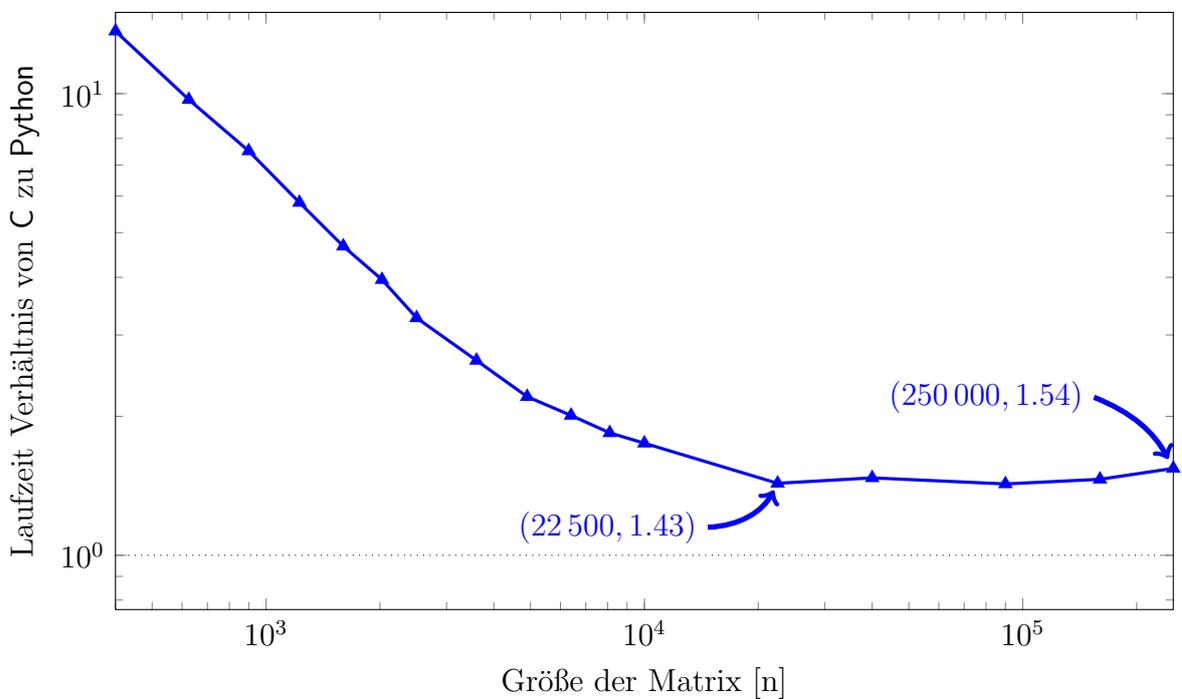


Abbildung 7.2.: Verhältnis der Laufzeit von Python zu C für `fdm_2d_sym` (sequentiell)

Wie in Abbildung 7.1 abzulesen ist, benötigt Python signifikant mehr Zeit, um die Shift-Parameter zu berechnen und das ADI Verfahren auszuführen. Bei $n = 250\,000$ terminiert der Algorithmus in Python nach 72.8 und in C nach 47.2 Sekunden. Abbildung 7.2 ist zu entnehmen, dass C 4 bis 14 mal weniger Zeit braucht bei $n \leq 2025$. In Python wird bei dem Importieren eines Moduls dieses komplett ausgeführt. Die Zeit, die dabei vergeht, macht sich vor Allem bei größeren Modulen wie SciPy bemerkbar. Python braucht ungefähr 145 Millisekunden, um aus diesem Modul ein Objekt zu importieren. Wird die Matrix F größer, verschwindet der Einfluss dieses Effektes im Verhältnis zu der Gesamtlaufzeit. Python ist bei einem $n = 22\,500$ ca. 43.2% langsamer als C. Für $n = 250\,000$ sind es hingegen wieder 54.2%. Vergleiche auch Tabelle 7.1.

Tabelle 7.1.: Laufzeiten für `fdm_2d_sym` mit einem Thread

Größe der Matrix F	Laufzeit in C [s]	Laufzeit in Python [s]	ratio	ADI Iterationen
400	0.03	0.43	13.66	15
625	0.04	0.43	9.71	15
900	0.06	0.47	7.51	16
1 225	0.08	0.49	5.81	17
1 600	0.11	0.54	4.67	18
2 025	0.14	0.56	3.95	18
2 500	0.18	0.60	3.27	19
3 600	0.28	0.73	2.64	21
4 900	0.38	0.84	2.21	21
6 400	0.50	1.01	2.01	22
8 100	0.63	1.17	1.84	22
10 000	0.81	1.42	1.75	23
22 500	2.07	2.97	1.43	25
40 000	4.10	6.03	1.47	29
90 000	11.59	16.54	1.43	30
160 000	26.14	38.19	1.46	32
250 000	47.21	72.83	1.54	34

Ein wesentlicher Unterschied zwischen C-M.E.S.S. und Python-M.E.S.S. ist, dass in ersterem nicht nur die LU-Zerlegung von F , sondern auch die der geshifteten Matrizen $(F + p_j I)$, $j = 1 \dots 25$, vorab berechnet und gespeichert werden. Mit 25 Shift-Parametern wiederholen sich diese geshifteten Matrizen ab dem 26. ADI Iterationsschritt, wodurch C-M.E.S.S. die LU-Zerlegungen wiederverwenden kann. Das Lösen der geshifteten Systeme $(F + p_j I)^{-1} W_{j-1}$ ist der aufwändigste Teil des Verfahrens. Bei $n = 22\,500$ führt der ADI Algorithmus genau 25 Iterationen aus. Mit steigendem n werden auch die Iterationsschritte mehr, wodurch es 34 bei $n = 250\,000$ sind. Python muss für dieses n 9 LU-Zerlegungen mehr berechnen als C, weshalb Python-M.E.S.S. im Verhältnis wieder langsamer wird.

Bekommt der Algorithmus eine unsymmetrische F Matrix aus der FDM Diskre-

tisierung mit Konvektion $f = [10, 100]^T [18, 20]$ und ansonsten die gleichen Daten wie oben als Eingabe, ist die Laufzeit etwas länger. Bei $n = 250\,000$ benötigt C 50.6 und Python 91.9 Sekunden (Abbildung 7.3, Tabelle A.1). Hier ist Python 56% bis 82% langsamer als C (Abbildung 7.4). Das ADI Verfahren führt an dieser Stelle bis zu 46 Iterationsschritte aus. Dadurch kann C-M.E.S.S. noch mehr von den zuvor ermittelten LU-Zerlegungen profitieren.

Die Spitze bei $n = 8\,100$ in Abbildung 7.4 lässt sich durch den gleichen Effekt erklären. Der ADI Algorithmus benötigt 33 Iterationen bei $n = 6\,400$ und 34 bei $n = 10\,000$, aber bei $n = 8\,100$ sind es 37. Dadurch kann die Implementierung in C ihren Vorteil durch die gespeicherten Zerlegungen für $n = 8\,100$ stärker ausnutzen als für $n = 6\,400$ und $n = 10\,000$. Das sonstige Verhalten ist analog zu dem obigen Beispiel.

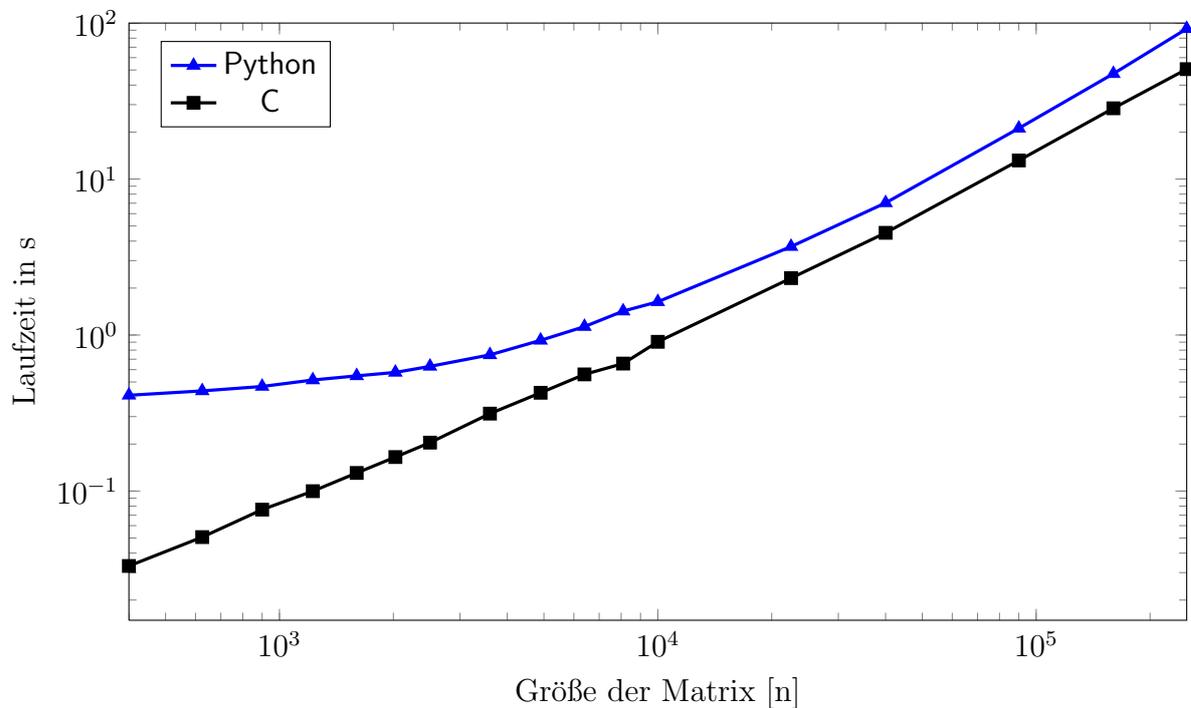


Abbildung 7.3.: Laufzeit von sequentieller Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_unsym`: Python vs. C

Dem folgenden Test liegt das Triple Chain Oscillator Modell zugrunde, welches sich an [23] orientiert. Die F Matrix wird in Größen von $n = 602$ bis $240\,002$ mit der Funktion `triplechain_MSD` generiert, welche sich in `triplechain_MSD.py` findet. Die Eingaben dieser sind ein $n_1 \in \mathbb{N}$ sowie $\alpha, \beta, v \in \mathbb{R}$ und ihre Rückgaben die Matrizen $W, D, K \in \mathbb{R}^{(3 \cdot n_1 + 1) \times (3 \cdot n_1 + 1)}$. Aus letzteren werden dann $F, M \in \mathbb{R}^{n \times n}$ sowie $G \in \mathbb{R}^n$ zusammengesetzt.

$$F := \begin{bmatrix} 0 & -K \\ -K & -D \end{bmatrix}, \quad M := \begin{bmatrix} -K & 0 \\ 0 & W \end{bmatrix}, \quad G := \begin{bmatrix} 0 \\ \mathbf{1} \end{bmatrix}, \quad n = 6 \cdot n_1 + 2$$

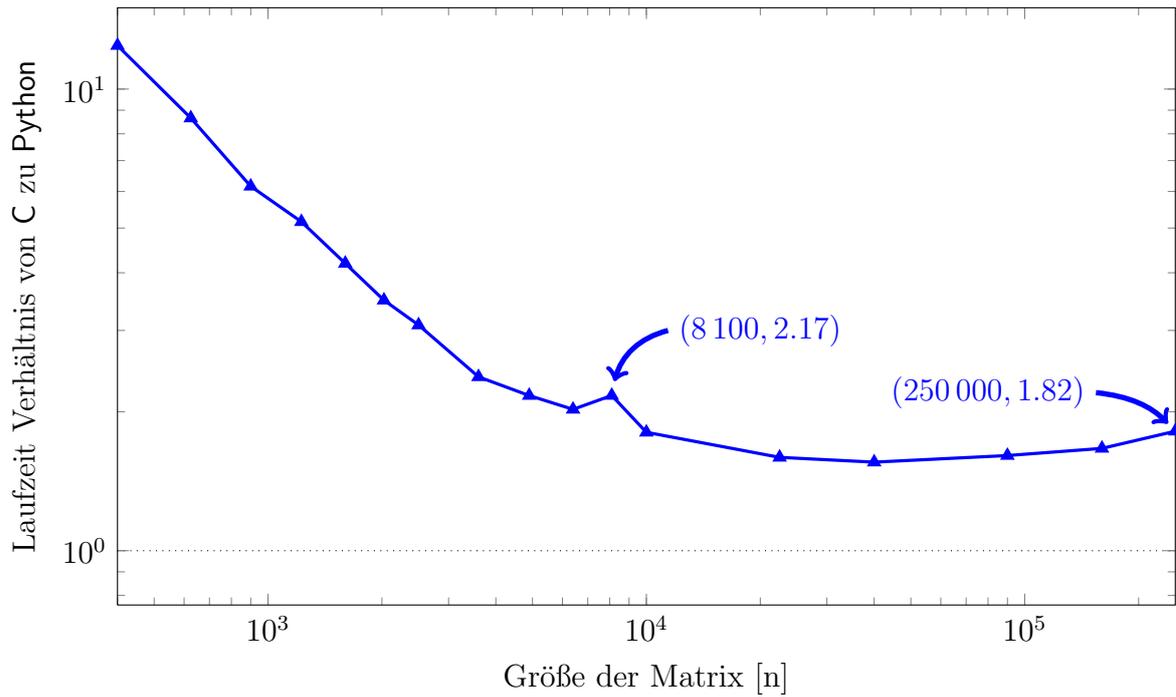


Abbildung 7.4.: Verhältnis der Laufzeit von Python zu C für `fdm_2d_unsym` (sequentiell)

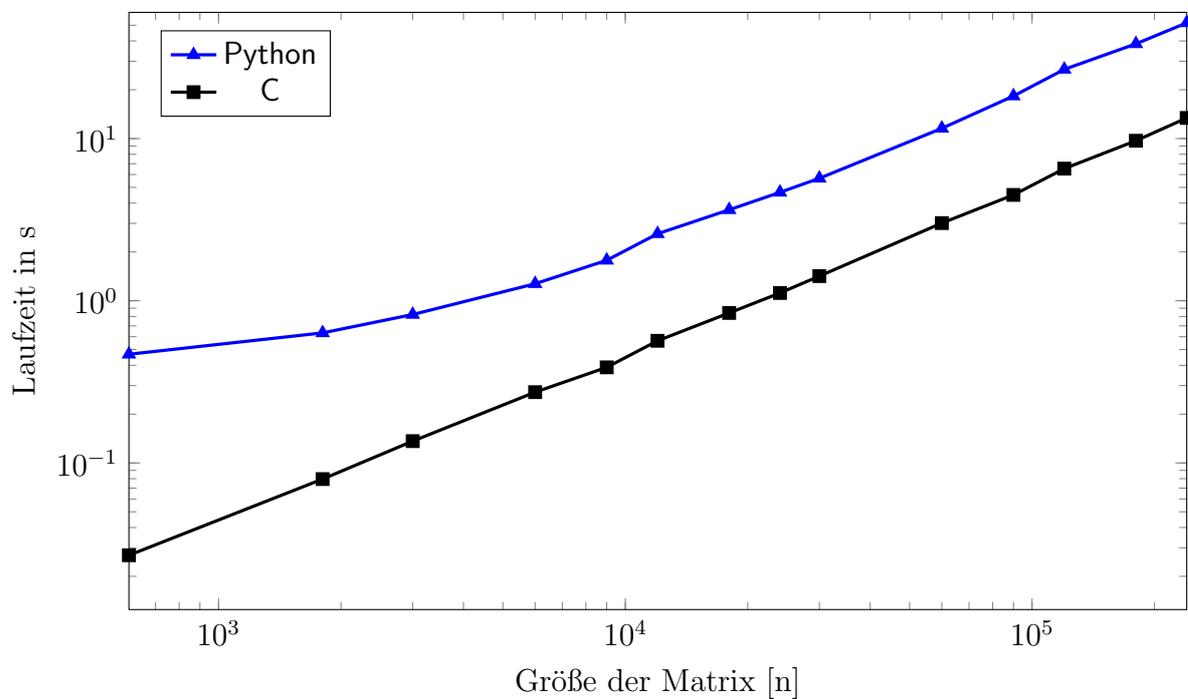


Abbildung 7.5.: Laufzeit von sequentieller Shift-Parameter Berechnung und LRCF_ADI für `MSD_TripleChain`: Python vs. C

Für diese Matrizen wird die Lösung der verallgemeinerten Lyapunov-Gleichung (5.1) mithilfe des ADI Verfahrens approximiert.

Als Eingabe der Funktion `triplechain_MSD` werden neben der Größe die Parameter $\alpha = 2$, $\beta = 5$ und $v = 5$ verwendet. Bei einem n bis einschließlich 24 002 werden die Durchläufe 20 mal wiederholt, 10 mal bis 60 002 und darüber hinaus 5 mal. Alle sonstigen Optionen für das ADI Verfahren sind die gleichen wie in den vorhergehenden Tests.

Haben die Matrizen eine Größe von $n = 240\,002$, benötigt C 13.4 und Python 51.7 Sekunden, um den ADI Algorithmus auszuführen (Abbildung 7.5). Damit ist C in diesem Fall ungefähr 4 mal so schnell wie Python. Ab einer Größe von $n = 12\,002$ werden 100 Iterationsschritte durchgeführt. Bei 25 Shift-Parametern kann C-M.E.S.S. jede LU-Zerlegung einer geschifteten Matrix 4 mal verwenden. Hieraus resultieren die höheren Faktoren in Abbildung 7.6 beziehungsweise Tabelle A.2. Abgesehen davon ist das Verhalten analog zu den vorherigen Tests.

Weitere Schwächen von Python treten beim Parallelisieren des Verfahrens auf und werden im folgenden Kapitel 7.2 behandelt.

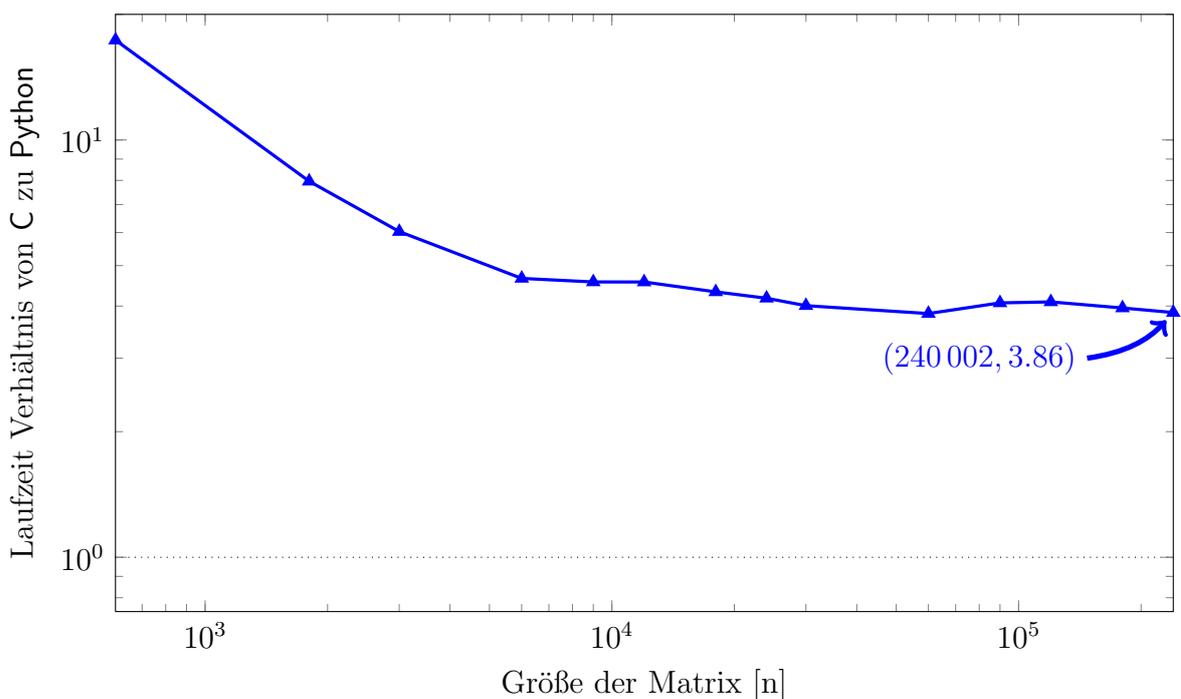


Abbildung 7.6.: Verhältnis der Laufzeit von Python zu C für `MSD_TripleChain` (sequentiell)

7.2. Parallele Tests

Analog zu den sequentiellen Testdurchläufen werden die beiden Implementierungen mit 2, 4, 8, 12 und 16 parallelen Threads ausgeführt, gesteuert durch die Variable `OMP_NUM_THREADS=t` mit $t = 1, 2, 4, 8, 12, 16$. Dabei wird mit dem Befehl

`numactl --cpunodebind=0` [14] sichergestellt, dass für 1, 2, 4 und 8 Threads diese nur auf einem der beiden Prozessoren laufen. Die Auswirkungen dieser Einschränkungen sind unter anderem in Abbildung 7.12 zu erkennen und werden im weiteren Verlauf dieses Kapitels anhand des `fdm_2d_sym`-Beispiels exemplarisch diskutiert.

In Python ist das ADI Verfahren rein sequentiell implementiert. Lediglich die im Hintergrund arbeitenden Bibliotheken wie BLAS, LAPACK und UMFPACK sind in der Lage mehrere Threads zu verwenden. Die C-Implementierung hingegen ist zusätzlich an weiteren Stellen parallelisiert. So werden zum Beispiel die vorab berechneten LU-Zerlegungen möglichst von mehreren Threads ermittelt.

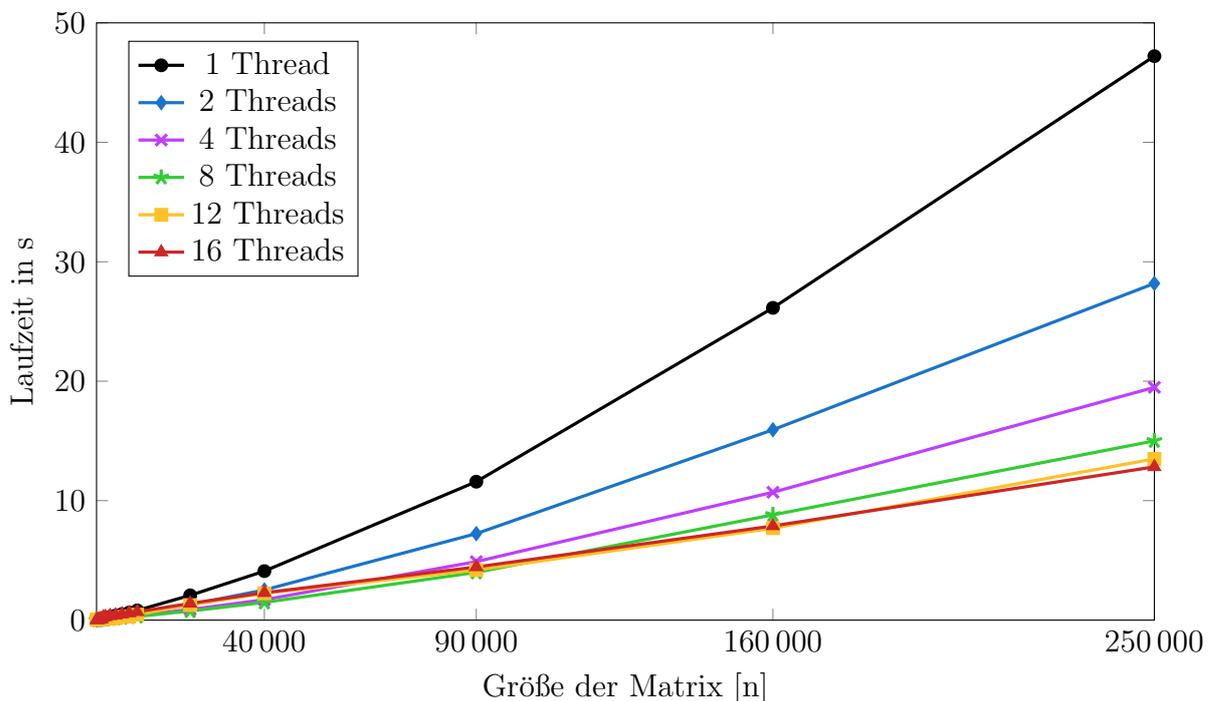


Abbildung 7.7.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_sym` mit dem C-M.E.S.S. Interface

Abbildung 7.7 ist zu entnehmen, dass der ADI Algorithmus in C deutlich von der Parallelisierung profitieren kann. Für $n = 250\,000$ benötigt C-M.E.S.S. mit einem Thread 47.2, mit 2 Threads 28.2 und mit allen 16 nur noch 12.8 Sekunden (Anhang A.2.1). Damit beträgt der parallele Beschleunigungsfaktor¹ ungefähr 3.7. Dies ergibt parallele Effizienzen von 84% mit 2 Threads und 23% mit 16 Threads, welche keine optimalen Speedups sind. Dafür gibt es im Wesentlichen zwei Gründe.

Zum einen ist der ADI Algorithmus nicht vollständig parallelisierbar, wodurch nach dem Amdahlschen Gesetz [19] dieser nur begrenzt beschleunigt werden kann. Zum anderen sind Algorithmen, wie das Lösen mit einer dünn besetzten Matrix, bandbreitenbeschränkt [12]. Das bedeutet, die Prozessoren können nicht schneller rechnen, als die

¹Im folgenden Speedup

Memory Management Unit die dafür benötigten Daten in deren Cache beziehungsweise Register laden kann.

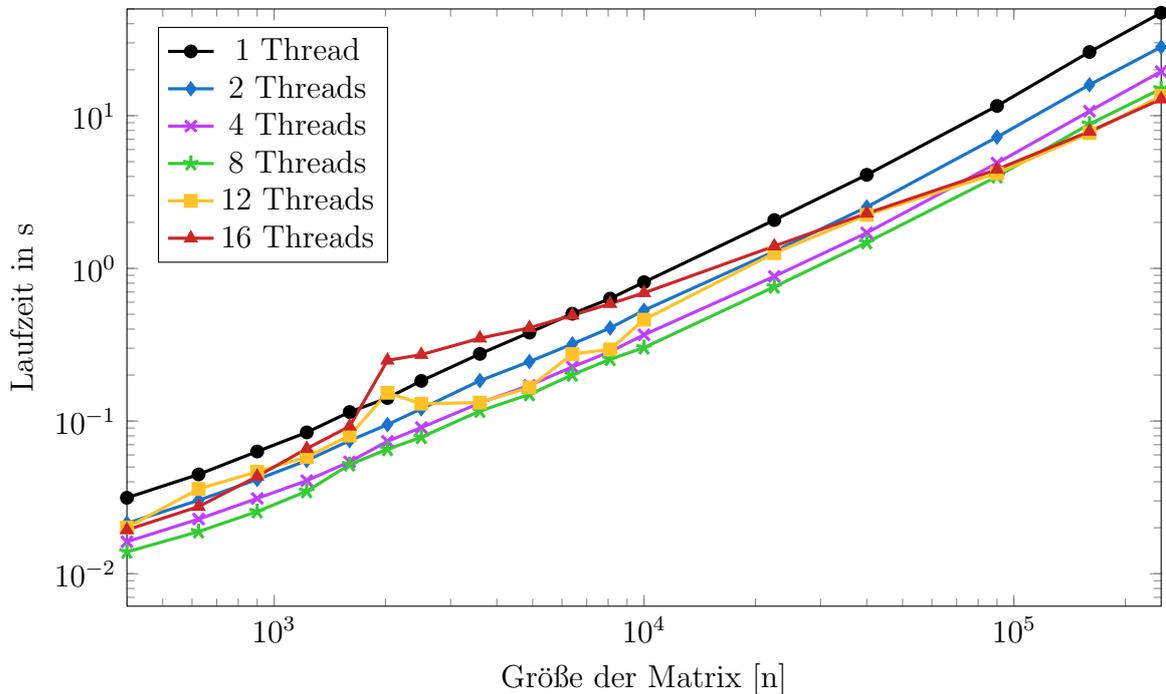


Abbildung 7.8.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_sym` mit dem C-M.E.S.S. Interface (logarithmische Achsenskalierung)

Das Verteilen der Daten auf mehrere Threads lohnt sich erst mit steigender Datengröße, wie in Abbildung 7.8 abzulesen ist. Zum Beispiel wird die Laufzeit mit 16 Threads erst bei $n \geq 40\,000$ kürzer als mit 2 Threads.

Python kann deutlich weniger von der Parallelisierung profitieren als C. In Abbildung 7.9 wird auf eine logarithmische Achsenskalierung verzichtet, da die Unterschiede der Laufzeiten von Python-M.E.S.S. mit 1, 2, 4 und 12 sonst nicht mehr erkennbar wären. Zusätzlich zu dem, was in Abbildung 7.9 sichtbar ist, ließe sich nur noch ausmachen, dass die Ausführung mit 2, 4 und 12 Threads bis zu einer Datengröße von $n \leq 90\,000$ noch langsamer als die sequentielle ist. Während Python-M.E.S.S. bei $n = 250\,000$ mit einem Thread 72.8 Sekunden benötigt, sind es mit 4 Threads 65.2, aber mit 16 Threads 105.1. Die parallele Effizienz beträgt 53% bei 2 und 28% bei 4 Threads und ist merklich schlechter als von C-M.E.S.S.. Es kann nur ein Speedup von 1.1 erreicht werden (4 Threads). Bei 16 Threads verlangsamt sich die Laufzeit sogar um 44% gegenüber der sequentiellen. Damit ist Python mit 16 Threads hier circa 8 mal langsamer als C gegenüber einem Faktor von 1.5 im nicht parallelisierten Fall (Abbildung 7.10). Ein Grund hierfür ist, dass C-M.E.S.S. die LU-Zerlegungen vorab parallel berechnet, Python-M.E.S.S. jedoch nicht, welches ein signifikanter Vorteil in der C Implementierung ist.

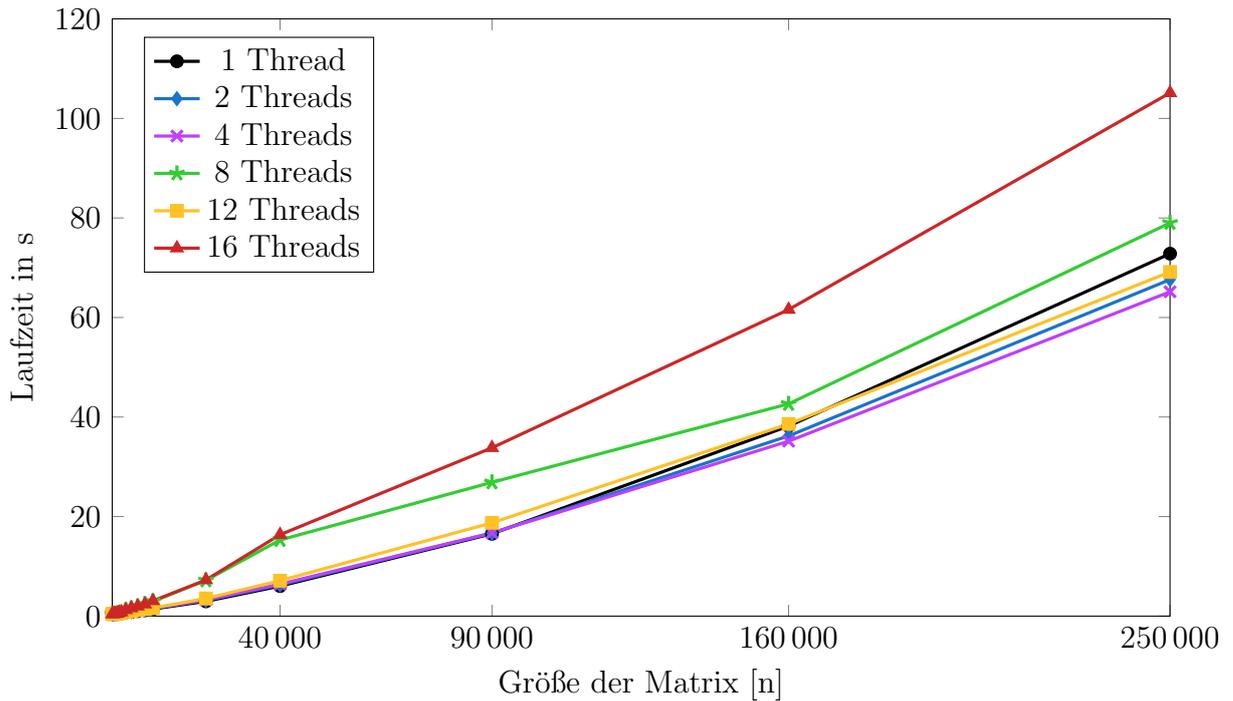


Abbildung 7.9.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF-ADI für `fdm_2d_sym` mit Python

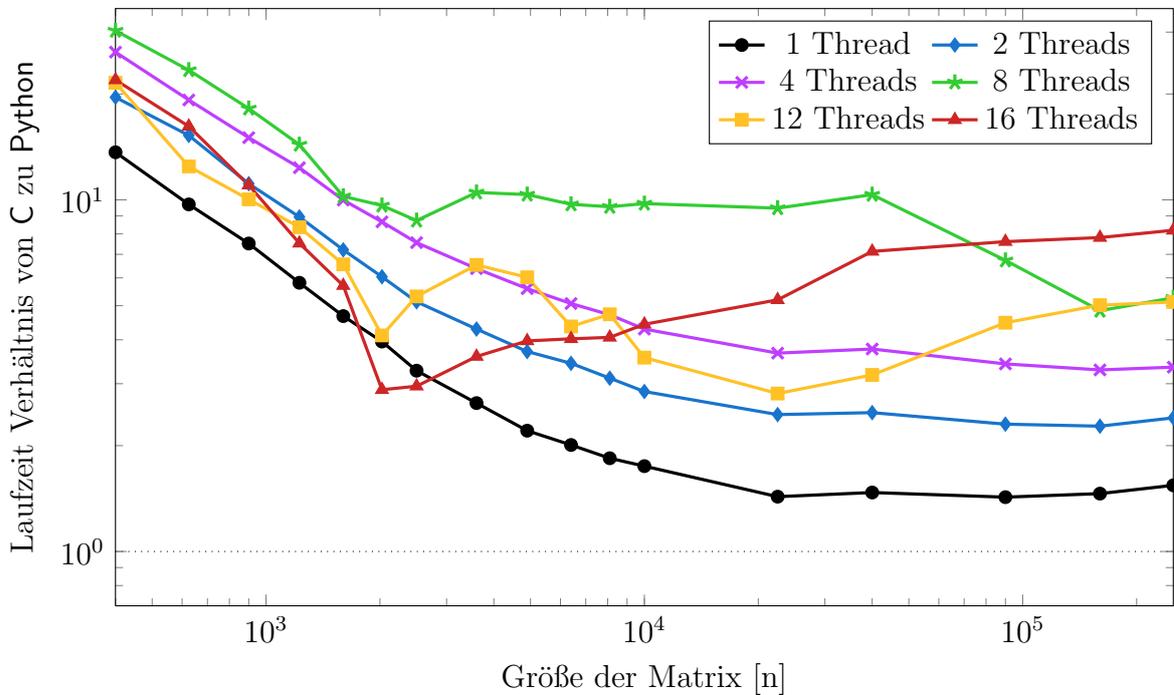


Abbildung 7.10.: Verhältnis der Laufzeit von Python zu C für `fdm_2d_sym` mit mehreren Threads

C-M.E.S.S. initialisiert die zur Verfügung stehenden Threads zu Beginn selber und verwaltet diese. Da Python-M.E.S.S. dies nicht tut, ist eine Vermutung, dass bei jedem Aufruf einer externen Bibliothek, die Threading unterstützt, wie beispielsweise BLAS, alle Threads wieder neu initialisiert werden müssen. Der dadurch zusätzlich erzeugte Aufwand kann von dem geringen Speedup bei Operationen mit dünn besetzten Matrizen nicht aufgewogen werden.

Auffallend ist das Ausbrechen der Graphen für 8 und 16 Threads in Abbildung 7.9. Ist `OMP_NUM_THREADS=16` gesetzt und wird eine externe Routine aufgerufen, die Threading unterstützt, kann diese alle 16 Threads nutzen, die auf alle 16 Kerne verteilt werden. Beispiele hierfür sind BLAS Routinen wie eine Matrix-Matrix-Multiplikation oder UMFPACK Routinen wie das Ermitteln einer LU-Zerlegung. Der Interpreter von Python hat jedoch noch einen eigenen Thread, womit insgesamt 17 Threads vorhanden sind. C-M.E.S.S. verhält sich genauso. Es ist aber sichergestellt, dass der 17. Thread, welcher nicht an der aktuellen Berechnung beteiligt ist, schläft, bis die externe Routine terminiert. In Python-M.E.S.S. kann das Verhalten des überschüssigen Threads nicht kontrolliert werden. Somit ist es möglich, dass dieser nebenbei nicht schläft, sondern sich zum Beispiel mit seinem Speichermanagement (Garbage Collector) befasst. Damit unterbricht er dann allerdings eventuell einen der anderen Threads, da kein Kern mehr frei ist, und verlangsamt so den kompletten Ablauf.

Genau der gleiche Effekt kann bei 8 Threads auftreten, da diese mit `numactl` auf einen Prozessor gebunden sind, welcher genau 8 Kerne zur Verfügung hat. Passend dazu ist das Verhalten, wenn `numactl` nicht benutzt wird. Abbildung 7.11 vergleicht die Laufzeiten von Python mit und ohne den Einsatz von `numactl`. Bis $n = 2500$ sind die Laufzeiten mit beiden Optionen nahezu gleich (Tabelle A.18). Bei $n > 2500$ sind 8 Threads mit `numactl` dann deutlich langsamer als ohne. Ansonsten treten keine nennenswerten Unterschiede auf bis $n = 22500$. Damit die Differenzen für $n \geq 22500$ gut erkennbar sind, beschränkt sich der dargestellte Bereich in der Abbildung auf diese Werte und die Farben der Graphen sind andere als in den vorherigen Abbildungen.

Ohne den Einsatz von `numactl` besteht keinerlei Kontrolle darüber, welchen Kernen die einzelnen Threads von dem Betriebssystem zugewiesen werden. Insbesondere ist die Verteilung auf unterschiedliche Prozessoren möglich. Letzteres geschieht auch schon bei 2 oder 4 Threads relativ häufig. Das Problem hierbei ist, dass dadurch die beiden Prozessoren zusätzlichen Kommunikationsaufwand untereinander haben. Die Daten, mit denen im Ausführungszeitpunkt gerechnet wird, müssen in den beiden Caches immer wieder synchronisiert werden und jeder Prozessor muss auf die Daten im Cache des jeweils anderen Prozessors zugreifen.

Genau dieser zusätzliche Aufwand und die zusätzlichen Speicherzugriffszeiten sollen vermieden werden, indem mit `numactl` sichergestellt wird, dass im Fall von höchstens 8 Threads diese nur auf einem Prozessor ausgeführt werden. An den Laufzeiten von C-M.E.S.S. (Abbildung 7.12) wird für $400 \leq n \leq 10000$ deutlich, dass durch die unterschiedliche Verteilung der Threads auf die Kerne der 2 Prozessoren ohne `numactl` keine zuverlässigen Werte erzielt werden können. Die erhoffte Verbesserung durch den Einsatz von `numactl` ist mit 2, 4 und 8 Threads deutlich erkennbar. Je größer die Daten werden, desto kleiner wird der Unterschied, da die oben erwähnte Bandbreitenbeschränkung der eingesetzten Algorithmen mit dünn besetzten Matrizen zum Tragen

kommt. Bei sequentieller Ausführung gibt es keinen Unterschied.

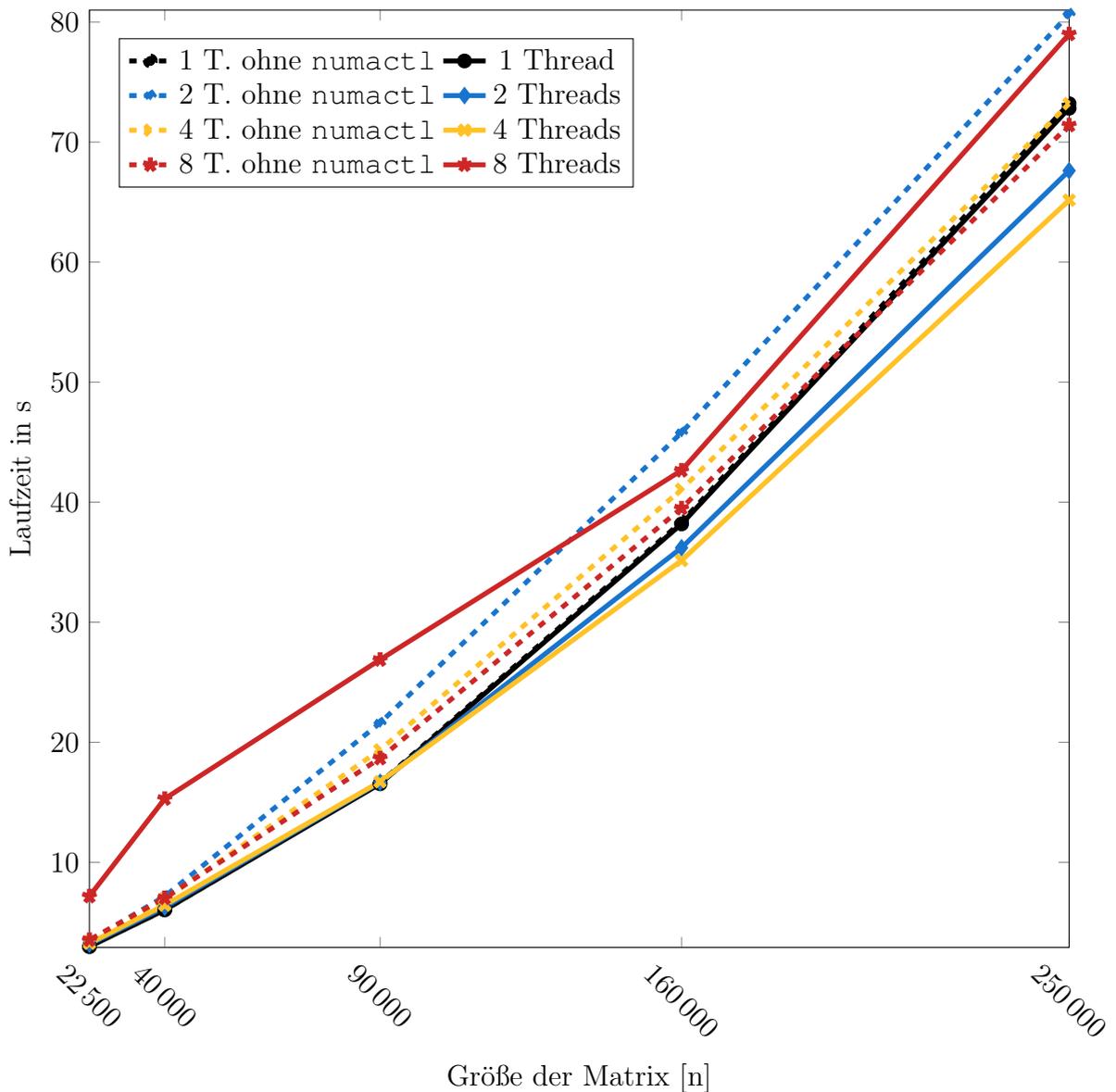


Abbildung 7.11.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_sym` mit Python mit und ohne `numactl`

In Python bringt `numactl` für 2 und 4 Threads auch den erwarteten Vorteil. Nur bei 8 Threads tritt der oben beschriebene negative Effekt ein, den C-M.E.S.S. vermeidet.

Die anderen beiden Testbeispiele (`fdm_2d_unsym` und `MSD_TripleChain`) aus Kapitel 7.1 verhalten sich bei der Parallelisierung sowohl in C als auch in Python analog. Die dazugehörigen Tabellen und Abbildungen finden sich im Anhang A.2.2 und A.2.3.

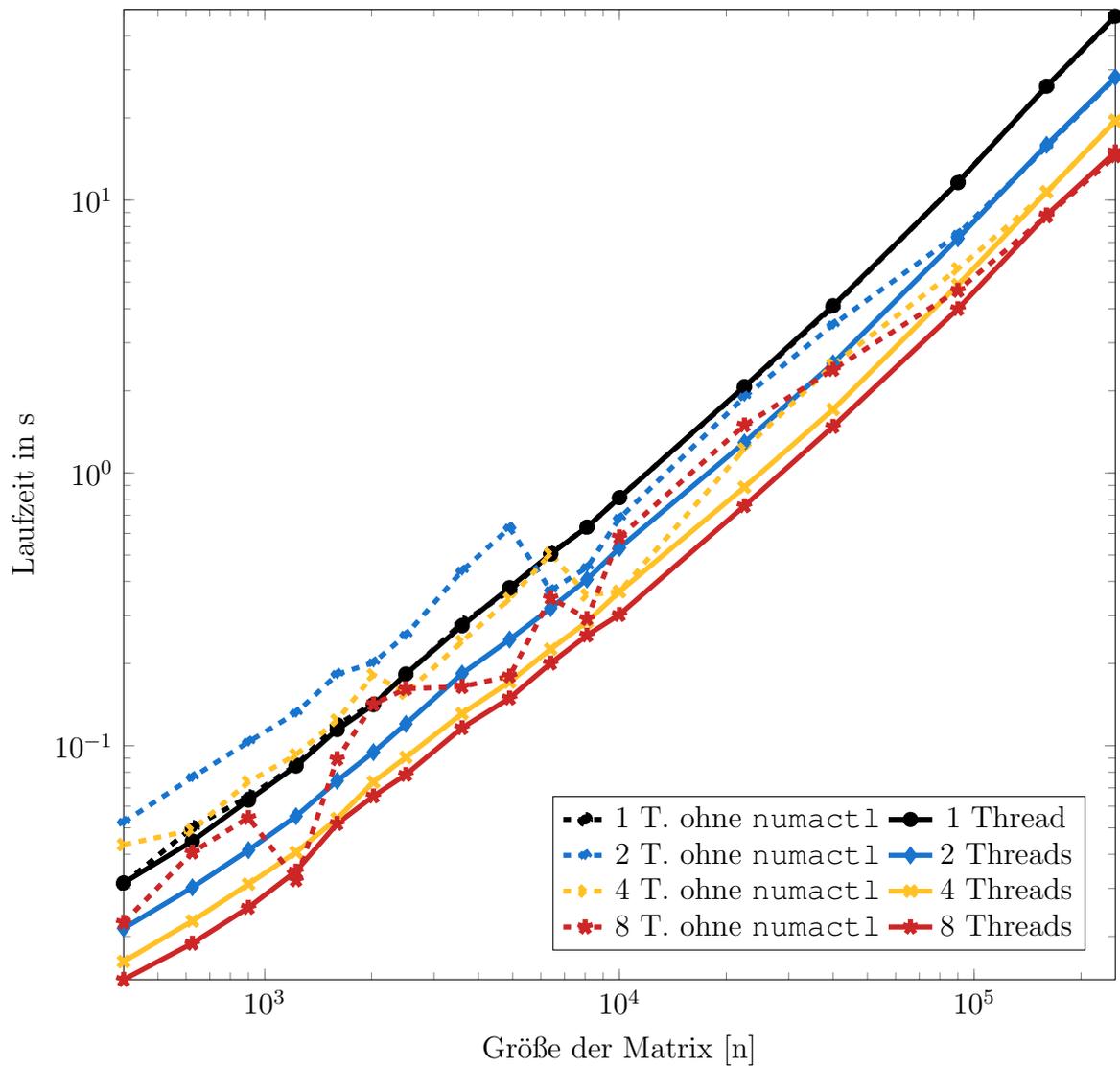


Abbildung 7.12.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_sym` mit dem C-M.E.S.S. Interface mit und ohne `numactl`

8. Zusammenfassung

In dieser Bachelorarbeit wurde gezeigt, wie die Lösung der Lyapunov-Gleichung und der verallgemeinerten Lyapunov-Gleichung mithilfe des ADI Verfahrens effizient approximiert werden kann. Der daraus entstandene Algorithmus lässt sich durch Niedrigrangfaktoren der Lösung und des Residuums auch für große dünn besetzte Matrizen anwendbar machen. Die im Allgemeinen voll besetzte Lösung muss auf diese Weise nicht explizit gespeichert werden. Eine weitere Einsparung von Speicherplatz und Rechenaufwand ist möglich, indem mit komplexen Shift-Parametern Doppelschritte durchgeführt werden. Dadurch kann verhindert werden, dass die Niedrigrangfaktoren der Lösung und des Residuums komplexe Werte enthalten.

Diese Verbesserungen wurden zu implementierungsfähigen Algorithmen sowohl für die Lyapunov-Gleichung als auch für die verallgemeinerte Lyapunov-Gleichung kombiniert.

Im Rahmen der vorliegenden Arbeit wurde das ADI Verfahren samt der genannten Verbesserungen in `Python` implementiert und damit in dieser immer populärer werdenden Programmiersprache verfügbar gemacht. Daraufhin sind die Laufzeiten der Implementierung mit denen ihrer Alternative verglichen worden, einem Interface zwischen `Python` und `C-M.E.S.S.`.

Dabei wurde deutlich, dass der Zugriff auf `C-M.E.S.S.` eine deutlich bessere Performance liefert, obwohl im Hintergrund bei beiden die selben Bibliotheken arbeiten und das Interface zusätzlich Kopien der Eingabedaten machen muss. Bei sequentieller Ausführung kann sich `Python-M.E.S.S.` noch bis auf ungefähr 40% an die Laufzeit in `C` annähern. Doch durch den Einsatz mehrerer Threads wurden drastische Defizite im Zusammenhang mit der Parallelisierbarkeit in `Python` aufgezeigt. `C-M.E.S.S.` ist selbst mit den größten getesteten Systemen bis zu 10 mal schneller.

9. Ausblick

Ein Interface zwischen C-M.E.S.S. und Python hat sich als die wesentlich leistungsfähigere Variante herausgestellt, das ADI Verfahren in Python verfügbar zu machen. Aufgrund dieses eindeutigen Ergebnisses ist nun das Interface auf die weiteren Funktionalitäten von M.E.S.S. zu erweitern. Es stehen zum Beispiel Modifikationen des ADI Algorithmus zur Verfügung, die speziell auf bestimmte Systeme zugeschnitten sind, aus denen die zum Einsatz kommenden Matrizen generiert werden. Die Blockstruktur der Matrizen bei linear quadratischen Regelungsproblemen zweiter Ordnung wie dem verwendeten Modell des Triple Chain Oscillators kann ausgenutzt werden [4]. Die zugrunde liegenden Routinen können an Index-1 Probleme angepasst werden, wobei beide Techniken kombinierbar sind. Weiterhin wird beim Approximieren einer Lösung der algebraischen Riccati-Gleichung

$$C^T C + A^T X + X A - X B R^{-1} B^T X = 0$$

mit einem Newton-Verfahren in jedem Newtonschritt eine Lyapunov-Gleichung gelöst [20]. Auch diese Methode ist in C-M.E.S.S. implementiert und das Interface könnte dahingehend erweitert werden.

Zusätzlich sind noch einige Probleme beim Zusammenspiel von Python und C im Zusammenhang mit dem Speichermanagement zu beheben.

A. Testergebnisse

Abkürzungen:

- n : Größe der Matrix F
- time C in s: Laufzeit in C in Sekunden
- time Py in s: Laufzeit in Python in Sekunden
- ratio: Verhältnis der Laufzeit von Python zu C
- # ADI: Iterationsschritte des ADI Verfahrens

A.1. Sequentielle Tests

A.1.1. `fdm_2d_unsym`

Tabelle A.1.: Laufzeiten für `fdm_2d_unsym` mit einem Thread

n	time C in s	time Py in s	ratio	# ADI
400	0.03	0.41	12.42	22
625	0.05	0.44	8.65	27
900	0.08	0.47	6.16	29
1 225	0.10	0.52	5.16	26
1 600	0.13	0.55	4.19	32
2 025	0.17	0.58	3.49	30
2 500	0.20	0.63	3.08	30
3 600	0.31	0.75	2.38	30
4 900	0.43	0.92	2.17	33
6 400	0.56	1.13	2.02	33
8 100	0.66	1.42	2.17	37
10 000	0.90	1.63	1.81	34
22 500	2.32	3.69	1.59	37
40 000	4.52	7.03	1.56	38
90 000	13.15	21.15	1.61	42
160 000	28.42	47.35	1.67	43
250 000	50.65	91.93	1.82	46

A.1.2. MSD_TripleChain

Tabelle A.2.: Laufzeiten für MSD_TripleChain mit einem Thread

n	time C in s	time Py in s	ratio	# ADI
602	0.03	0.47	17.35	29
1 802	0.08	0.63	7.96	59
3 002	0.14	0.82	6.03	59
6 002	0.27	1.27	4.66	75
9 002	0.39	1.78	4.57	86
12 002	0.57	2.59	4.57	100
18 002	0.84	3.64	4.33	100
24 002	1.12	4.66	4.18	100
30 002	1.42	5.69	4.01	100
60 002	3.01	11.56	3.84	100
90 002	4.49	18.29	4.07	100
120 002	6.53	26.71	4.09	100
180 002	9.70	38.38	3.96	100
240 002	13.40	51.72	3.86	100

A.2. Parallele Tests

A.2.1. `fdm_2d_sym`

Tabelle A.3.: Laufzeiten für `fdm_2d_sym` mit **2** Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.42	19.57	15
625	0.03	0.46	15.25	15
900	0.04	0.46	11.12	16
1 225	0.06	0.49	8.94	17
1 600	0.07	0.54	7.21	18
2 025	0.09	0.57	6.04	18
2 500	0.12	0.62	5.13	19
3 600	0.18	0.79	4.29	21
4 900	0.25	0.91	3.70	21
6 400	0.32	1.10	3.43	22
8 100	0.41	1.27	3.11	22
10 000	0.53	1.51	2.85	23
22 500	1.29	3.17	2.45	25
40 000	2.52	6.26	2.48	29
90 000	7.24	16.66	2.30	30
160 000	15.93	36.20	2.27	32
250 000	28.18	67.62	2.40	34

Tabelle A.4.: Laufzeiten für `fdm_2d_sym` mit 4 Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.43	26.31	15
625	0.02	0.44	19.24	15
900	0.03	0.47	15.03	16
1 225	0.04	0.50	12.35	17
1 600	0.05	0.54	9.99	18
2 025	0.07	0.64	8.66	18
2 500	0.09	0.69	7.56	19
3 600	0.13	0.84	6.38	21
4 900	0.17	0.96	5.59	21
6 400	0.23	1.15	5.07	22
8 100	0.28	1.34	4.71	22
10 000	0.37	1.57	4.29	23
22 500	0.89	3.25	3.67	25
40 000	1.71	6.43	3.77	29
90 000	4.89	16.71	3.42	30
160 000	10.70	35.15	3.28	32
250 000	19.49	65.16	3.34	34

Tabelle A.5.: Laufzeiten für `fdm_2d_sym` mit 8 Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.01	0.42	30.28	15
625	0.02	0.44	23.39	15
900	0.03	0.46	18.19	16
1 225	0.03	0.50	14.36	17
1 600	0.05	0.53	10.23	18
2 025	0.07	0.63	9.64	18
2 500	0.08	0.68	8.73	19
3 600	0.12	1.22	10.50	21
4 900	0.15	1.55	10.35	21
6 400	0.20	1.95	9.71	22
8 100	0.25	2.43	9.56	24
10 000	0.30	2.95	9.76	23
22 500	0.76	7.17	9.47	25
40 000	1.48	15.30	10.35	29
90 000	3.99	26.88	6.73	30
160 000	8.81	42.65	4.84	32
250 000	15.00	79.00	5.27	34

Tabelle A.6.: Laufzeiten für `fdm_2d_sym` mit **12** Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.43	21.53	15
625	0.04	0.45	12.45	15
900	0.05	0.47	10.03	16
1 225	0.06	0.49	8.36	17
1 600	0.08	0.53	6.56	18
2 025	0.15	0.63	4.12	18
2 500	0.13	0.69	5.31	19
3 600	0.13	0.86	6.53	21
4 900	0.17	1.00	6.02	21
6 400	0.27	1.20	4.36	22
8 100	0.29	1.39	4.73	22
10 000	0.46	1.64	3.56	23
22 500	1.26	3.54	2.81	25
40 000	2.24	7.12	3.18	29
90 000	4.19	18.74	4.47	30
160 000	7.69	38.60	5.02	32
250 000	13.49	69.14	5.12	33

Tabelle A.7.: Laufzeiten für `fdm_2d_sym` mit **16** Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.42	21.93	15
625	0.03	0.45	16.18	15
900	0.04	0.48	11.01	16
1 225	0.07	0.50	7.53	17
1 600	0.09	0.53	5.70	18
2 025	0.25	0.72	2.88	18
2 500	0.27	0.80	2.95	19
3 600	0.35	1.25	3.59	21
4 900	0.41	1.62	3.97	21
6 400	0.49	1.98	4.03	22
8 100	0.59	2.38	4.06	22
10 000	0.69	3.06	4.43	23
22 500	1.40	7.26	5.20	25
40 000	2.29	16.35	7.14	29
90 000	4.45	33.82	7.61	30
160 000	7.88	61.59	7.81	32
250 000	12.83	105.14	8.20	34

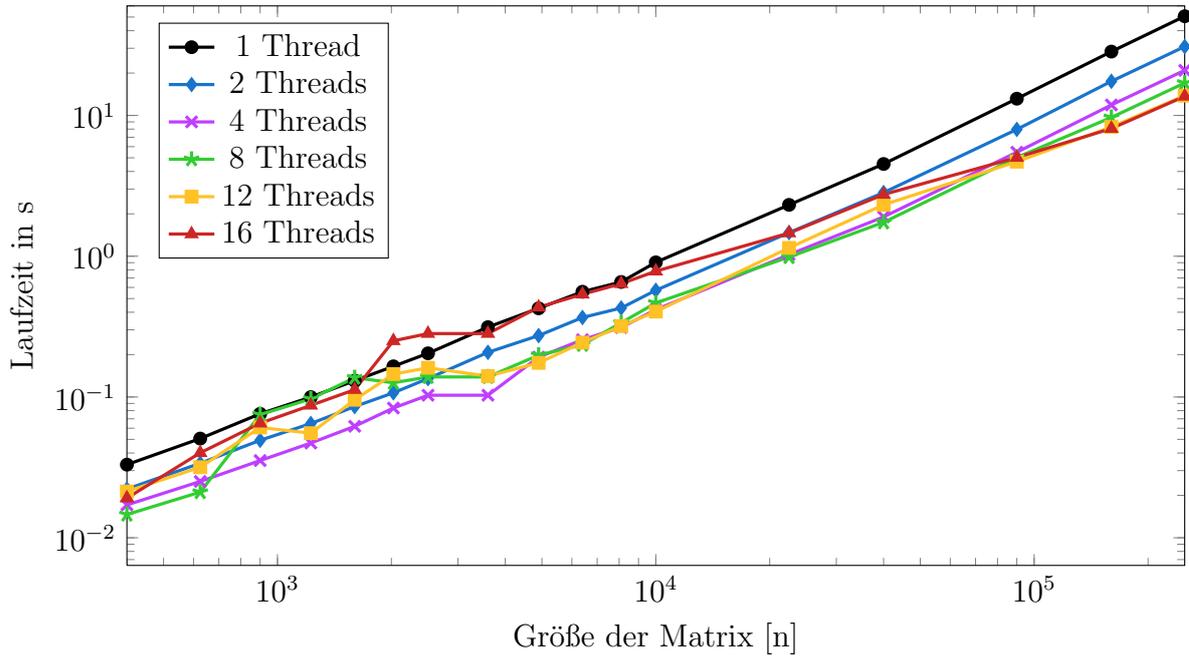
A.2.2. `fdm_2d_unsym`

Abbildung A.1.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_unsym` mit dem C-M.E.S.S. Interface

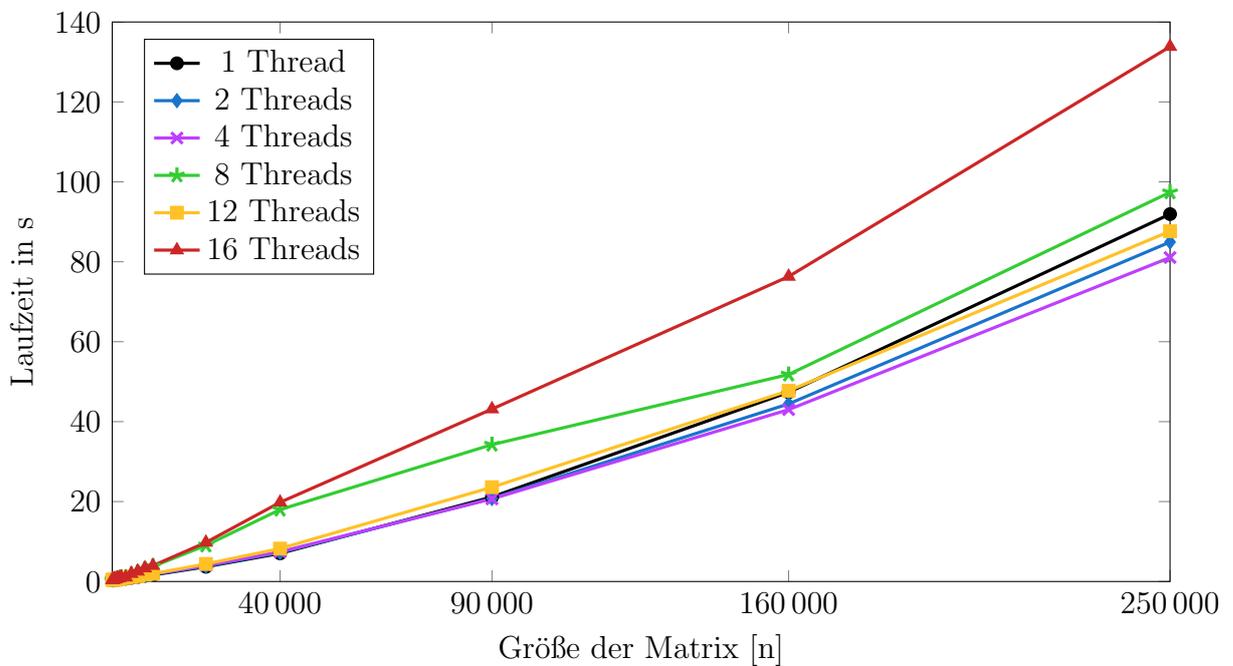


Abbildung A.2.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für `fdm_2d_unsym` mit Python

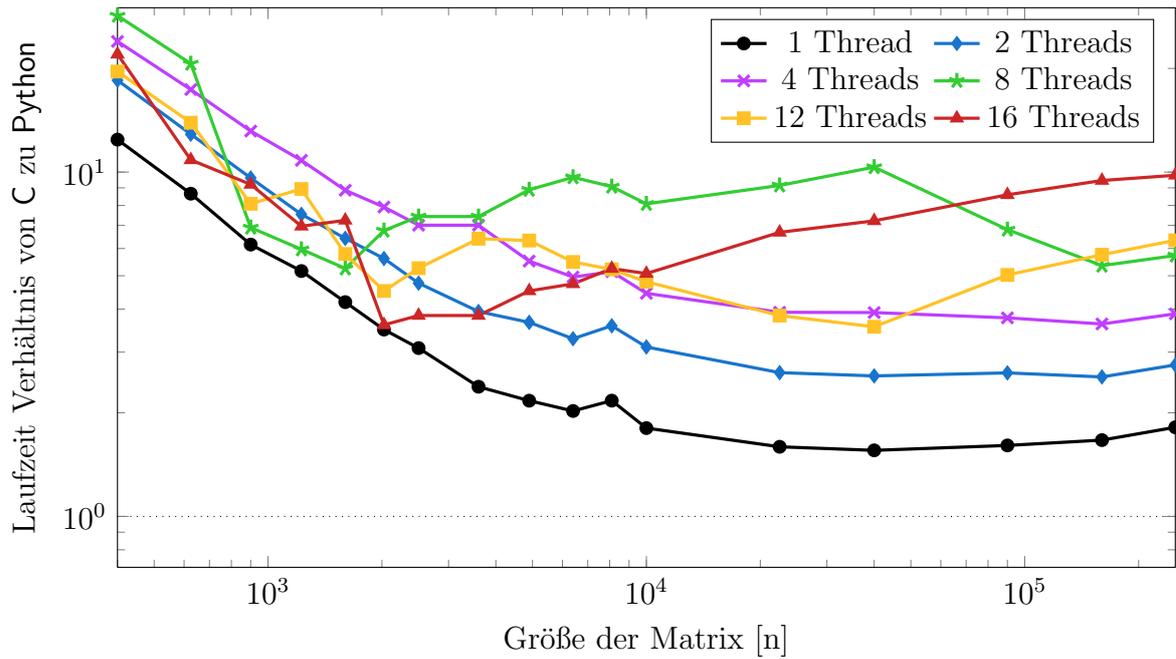


Abbildung A.3.: Verhältnis der Laufzeit von Python zu C für `fdm_2d_unsym` mit mehreren Threads

Tabelle A.8.: Laufzeiten für `fdm_2d_unsym` mit **2** Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.41	18.48	22
625	0.03	0.44	12.88	27
900	0.05	0.47	9.62	29
1 225	0.06	0.49	7.54	26
1 600	0.09	0.55	6.42	32
2 025	0.11	0.60	5.61	30
2 500	0.14	0.64	4.76	30
3 600	0.21	0.82	3.94	30
4 900	0.27	1.00	3.66	33
6 400	0.37	1.20	3.28	33
8 100	0.43	1.53	3.58	37
10 000	0.57	1.78	3.11	34
22 500	1.47	3.85	2.62	37
40 000	2.83	7.25	2.56	38
90 000	7.96	20.78	2.61	42
160 000	17.48	44.44	2.54	43
250 000	30.85	84.95	2.75	46

Tabelle A.9.: Laufzeiten für `fdm_2d_unsym` mit 4 Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.41	23.96	22
625	0.03	0.44	17.37	27
900	0.04	0.47	13.16	29
1 225	0.05	0.51	10.82	26
1 600	0.06	0.55	8.86	32
2 025	0.08	0.66	7.92	30
2 500	0.10	0.72	7.01	30
3 600	0.10	0.72	7.01	30
4 900	0.19	1.06	5.52	33
6 400	0.26	1.27	4.96	33
8 100	0.31	1.60	5.14	37
10 000	0.42	1.85	4.45	34
22 500	1.03	4.02	3.92	37
40 000	1.90	7.44	3.91	38
90 000	5.48	20.67	3.77	42
160 000	11.87	43.00	3.62	43
250 000	20.92	81.07	3.88	46

Tabelle A.10.: Laufzeiten für `fdm_2d_unsym` mit 8 Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.01	0.42	28.46	22
625	0.02	0.44	20.65	27
900	0.07	0.52	6.90	29
1 225	0.10	0.58	5.97	26
1 600	0.14	0.72	5.25	32
2 025	0.13	0.85	6.76	30
2 500	0.14	1.03	7.43	30
3 600	0.14	1.03	7.43	30
4 900	0.20	1.76	8.88	33
6 400	0.23	2.27	9.66	33
8 100	0.34	3.06	9.07	37
10 000	0.47	3.77	8.09	34
22 500	0.99	9.02	9.15	37
40 000	1.74	18.00	10.34	38
90 000	5.03	34.25	6.81	42
160 000	9.68	51.80	5.35	43
250 000	17.01	97.40	5.72	46

Tabelle A.11.: Laufzeiten für `fdm_2d_unsym` mit **12** Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.42	19.59	22
625	0.03	0.44	13.91	27
900	0.06	0.49	8.09	29
1 225	0.06	0.49	8.93	26
1 600	0.10	0.55	5.78	32
2 025	0.15	0.66	4.52	30
2 500	0.16	0.85	5.26	30
3 600	0.14	0.90	6.40	30
4 900	0.17	1.11	6.33	33
6 400	0.24	1.34	5.48	33
8 100	0.32	1.66	5.22	37
10 000	0.40	1.94	4.80	34
22 500	1.15	4.39	3.83	37
40 000	2.32	8.24	3.55	38
90 000	4.69	23.59	5.03	42
160 000	8.29	47.72	5.75	43
250 000	13.83	87.64	6.34	46

Tabelle A.12.: Laufzeiten für `fdm_2d_unsym` mit **16** Threads

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.42	21.99	22
625	0.04	0.44	10.85	27
900	0.07	0.60	9.21	29
1 225	0.09	0.61	6.96	26
1 600	0.11	0.82	7.24	32
2 025	0.25	0.90	3.60	30
2 500	0.28	1.08	3.83	30
3 600	0.28	1.08	3.83	30
4 900	0.43	1.96	4.52	33
6 400	0.54	2.54	4.74	33
8 100	0.64	3.35	5.25	37
10 000	0.78	3.97	5.07	34
22 500	1.46	9.77	6.68	37
40 000	2.74	19.81	7.22	38
90 000	5.02	43.14	8.59	42
160 000	8.07	76.31	9.45	43
250 000	13.67	133.83	9.79	46

A.2.3. MSD_TripleChain

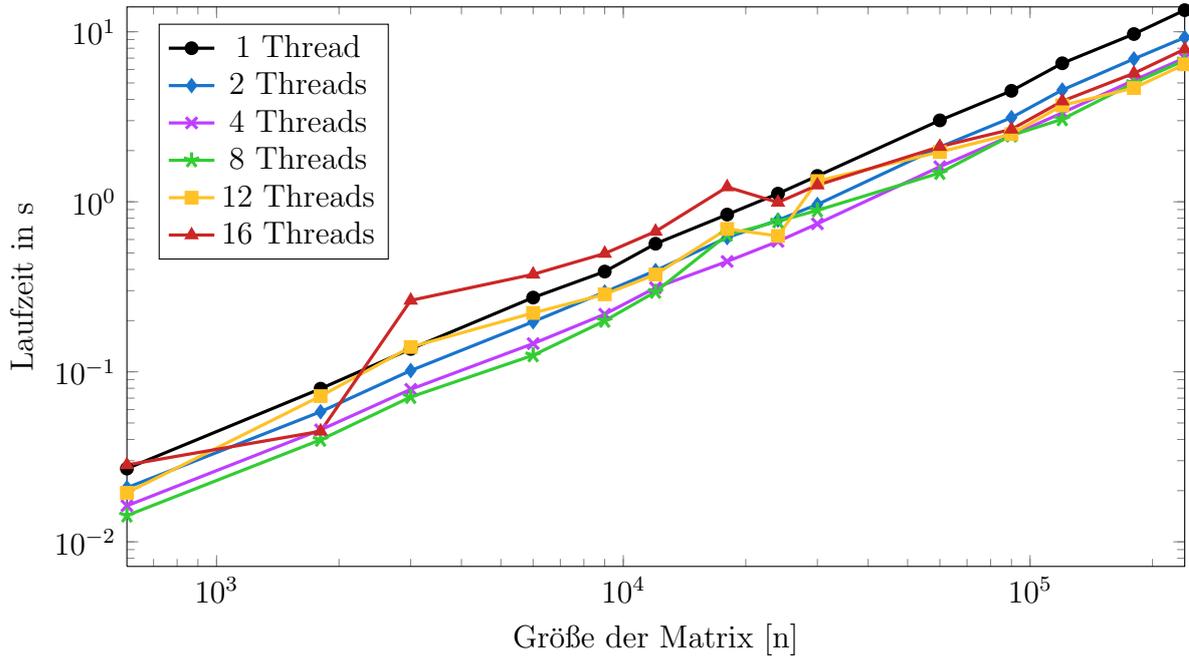


Abbildung A.4.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für MSD_TripleChain mit dem C-M.E.S.S. Interface

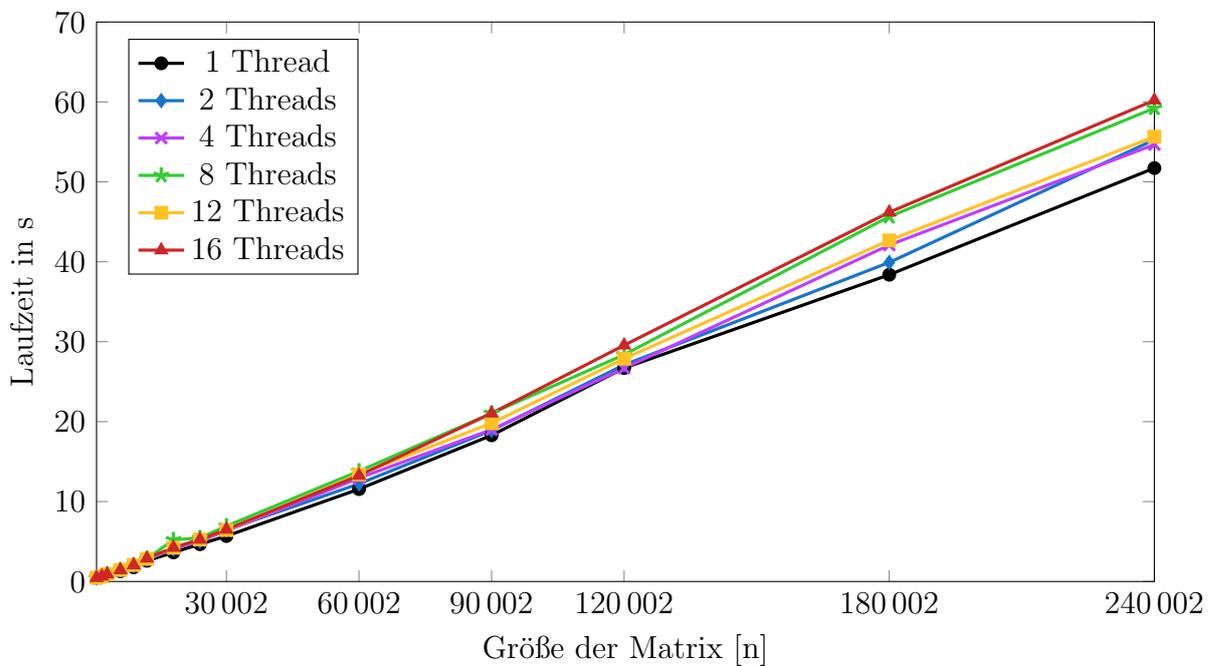


Abbildung A.5.: Laufzeit von paralleler Shift-Parameter Berechnung und LRCF_ADI für MSD_TripleChain mit Python

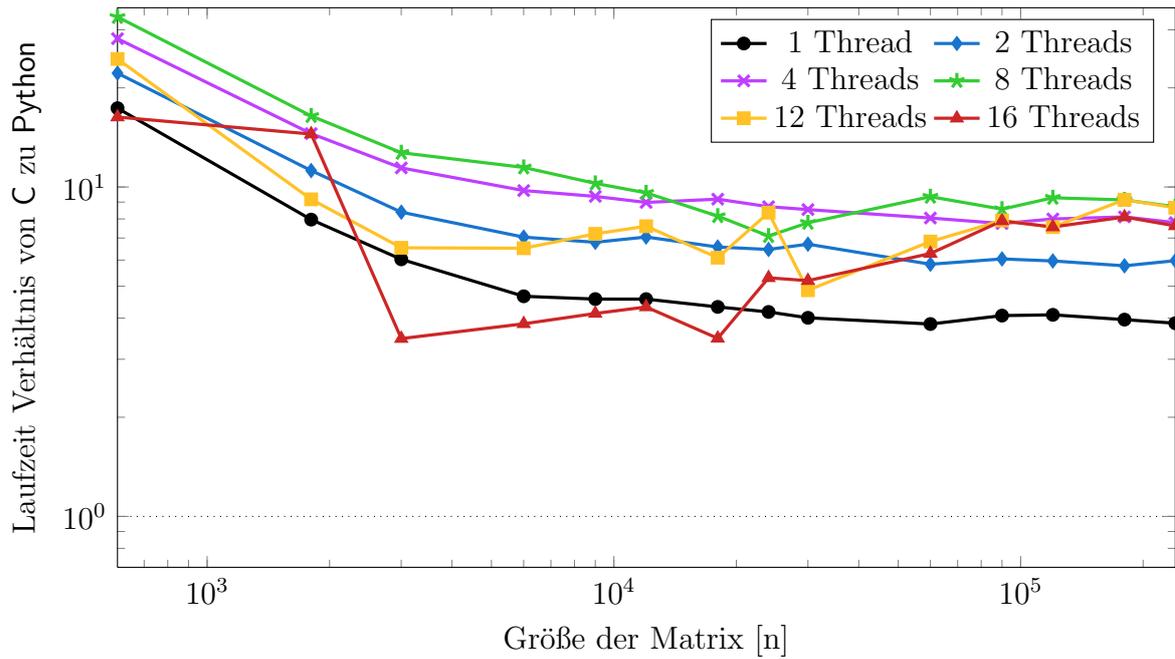


Abbildung A.6.: Verhältnis der Laufzeit von Python zu C für `MSD_TripleChain` mit mehreren Threads

Tabelle A.13.: Laufzeiten für `MSD_TripleChain` mit **2** Threads

n	time C in s	time Py in s	ratio	# ADI
602	0.02	0.46	22.16	29
1 802	0.06	0.65	11.22	59
3 002	0.10	0.85	8.39	59
6 002	0.20	1.39	7.04	75
9 002	0.29	2.00	6.80	86
12 002	0.39	2.78	7.05	100
18 002	0.62	4.05	6.57	100
24 002	0.78	5.04	6.46	100
30 002	0.96	6.47	6.71	100
60 002	2.10	12.23	5.83	100
90 002	3.13	18.90	6.05	100
120 002	4.54	27.10	5.96	100
180 002	6.92	39.92	5.77	100
240 002	9.25	55.27	5.98	100

Tabelle A.14.: Laufzeiten für MSD_TripleChain mit 4 Threads

n	time C in s	time Py in s	ratio	# ADI
602	0.02	0.46	28.21	29
1 802	0.05	0.66	14.52	59
3 002	0.08	0.90	11.43	59
6 002	0.15	1.43	9.75	75
9 002	0.22	2.04	9.36	86
12 002	0.31	2.80	8.98	100
18 002	0.45	4.09	9.19	100
24 002	0.58	5.10	8.72	100
30 002	0.74	6.34	8.54	100
60 002	1.61	12.94	8.05	100
90 002	2.45	19.00	7.75	100
120 002	3.34	26.69	8.00	100
180 002	5.19	42.09	8.11	100
240 002	7.01	54.66	7.80	100

Tabelle A.15.: Laufzeiten für MSD_TripleChain mit 8 Threads

n	time C in s	time Py in s	ratio	# ADI
602	0.01	0.47	32.83	29
1 802	0.04	0.65	16.43	59
3 002	0.07	0.90	12.71	59
6 002	0.13	1.43	11.47	75
9 002	0.20	2.05	10.26	86
12 002	0.30	2.84	9.60	100
18 002	0.64	5.21	8.16	100
24 002	0.77	5.45	7.10	100
30 002	0.89	6.94	7.79	100
60 002	1.48	13.81	9.34	100
90 002	2.46	21.05	8.57	100
120 002	3.06	28.38	9.28	100
180 002	5.00	45.65	9.14	100
240 002	6.80	59.22	8.71	100

Tabelle A.16.: Laufzeiten für MSD_TripleChain mit **12** Threads

n	time C in s	time Py in s	ratio	# ADI
602	0.02	0.48	24.47	29
1 802	0.07	0.66	9.18	59
3 002	0.14	0.92	6.54	59
6 002	0.22	1.45	6.52	75
9 002	0.29	2.06	7.21	86
12 002	0.37	2.84	7.60	100
18 002	0.69	4.23	6.10	100
24 002	0.63	5.26	8.36	100
30 002	1.32	6.44	4.86	100
60 002	1.96	13.40	6.83	100
90 002	2.50	19.81	7.92	100
120 002	3.69	27.87	7.56	100
180 002	4.67	42.68	9.15	100
240 002	6.43	55.66	8.65	100

Tabelle A.17.: Laufzeiten für MSD_TripleChain mit **16** Threads

n	time C in s	time Py in s	ratio	# ADI
602	0.03	0.46	16.31	29
1 802	0.04	0.65	14.48	59
3 002	0.26	0.91	3.47	59
6 002	0.37	1.44	3.84	75
9 002	0.50	2.05	4.13	86
12 002	0.67	2.89	4.32	100
18 002	1.22	4.24	3.47	100
24 002	0.99	5.25	5.30	100
30 002	1.25	6.51	5.20	100
60 002	2.11	13.26	6.28	100
90 002	2.67	21.05	7.89	100
120 002	3.91	29.56	7.56	100
180 002	5.70	46.20	8.11	100
240 002	7.90	60.21	7.62	100

A.3. *fdm_2d_sym ohne numactl*Tabelle A.18.: Laufzeiten für *fdm_2d_sym* mit einem Thread ohne *numactl*

n	time C in s	time Py in s	ratio	# ADI
400	0.03	0.42	13.46	15
625	0.05	0.44	9.86	15
900	0.07	0.46	7.34	16
1225	0.08	0.50	5.88	17
1600	0.12	0.53	4.66	18
2025	0.14	0.57	4.00	18
2500	0.18	0.61	3.31	19
3600	0.28	0.73	2.64	21
4900	0.37	0.84	2.22	21
6400	0.51	1.02	2.01	22
8100	0.63	1.18	1.86	22
10000	0.81	1.43	1.76	23
22500	2.06	3.01	1.45	25
40000	4.05	6.03	1.47	29
90000	11.65	16.63	1.44	30
160000	26.06	38.36	1.47	32
250000	46.90	73.21	1.55	34

Tabelle A.19.: Laufzeiten für *fdm_2d_sym* mit **2** Threads ohne *numactl*

n	time C in s	time Py in s	ratio	# ADI
400	0.05	0.42	19.51	15
625	0.08	0.44	14.52	15
900	0.10	0.46	11.14	16
1225	0.13	0.49	8.83	17
1600	0.18	0.52	7.06	18
2025	0.20	0.56	5.96	18
2500	0.25	0.62	5.12	19
3600	0.44	0.78	4.27	21
4900	0.63	0.94	3.83	21
6400	0.37	1.12	3.51	22
8100	0.45	1.31	3.21	22
10000	0.68	1.58	2.98	23
22500	1.90	3.44	2.66	25
40000	3.49	7.17	2.84	29
90000	7.51	21.60	2.98	30
160000	15.67	45.80	2.87	32
250000	28.00	80.67	2.86	34

Tabelle A.20.: Laufzeiten für *fdm_2d_sym* mit 4 Threads ohne *numactl*

n	time C in s	time Py in s	ratio	# ADI
400	0.04	0.43	26.35	15
625	0.05	0.45	19.56	15
900	0.07	0.47	14.99	16
1225	0.09	0.49	11.93	17
1600	0.12	0.53	9.79	18
2025	0.18	0.63	8.56	18
2500	0.16	0.67	7.44	19
3600	0.24	0.84	6.37	21
4900	0.35	0.98	5.72	21
6400	0.51	1.18	5.22	22
8100	0.36	1.34	4.71	22
10000	0.37	1.60	4.36	23
22500	1.22	3.42	3.86	25
40000	2.49	6.93	4.06	29
90000	5.60	19.36	3.96	30
160000	10.68	41.07	3.84	32
250000	19.60	73.20	3.76	34

Tabelle A.21.: Laufzeiten für *fdm_2d_sym* mit 8 Threads ohne *numactl*

n	time C in s	time Py in s	ratio	# ADI
400	0.02	0.42	30.57	15
625	0.04	0.44	23.30	15
900	0.05	0.47	18.26	16
1225	0.03	0.49	14.10	17
1600	0.09	0.54	10.48	18
2025	0.14	0.63	9.57	18
2500	0.16	0.68	8.72	19
3600	0.16	0.84	7.21	21
4900	0.18	0.98	6.58	21
6400	0.35	1.17	5.84	22
8100	0.29	1.36	5.34	22
10000	0.58	1.61	5.34	23
22500	1.50	3.54	4.67	25
40000	2.39	7.03	4.75	29
90000	4.67	18.66	4.67	30
160000	8.74	39.49	4.48	32
250000	14.52	71.41	4.76	34

A.4. Datenträger

Literaturverzeichnis

- [1] *BLAS (Basic Linear Algebra Subprograms)*. <http://www.netlib.org/blas/>.
- [2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. <http://www.netlib.org/lapack/>.
- [3] B. Baran, M. Köhler, N. Prasad, and J. Saak. Interfacing C-M.E.S.S. with Python. Preprint MPIMD/13-22, Max Planck Institute Magdeburg, November 2013. Available from <http://www.mpi-magdeburg.mpg.de/preprints/>.
- [4] M. Behr. *Second Order LQR-Problems in M.E.S.S.* Bachelorarbeit, Max-Planck-Institut für Dynamik komplexer technischer Systeme, Magdeburg, 2013.
- [5] P. Benner, P. Kürschner, and J. Saak. An improved numerical method for balanced truncation for symmetric second-order systems. *Mathematical and Computer Modelling of Dynamical Systems*, 0(0):1–23, 0. doi:10.1080/13873954.2013.794363.
- [6] P. Benner, P. Kürschner, and J. Saak. Efficient handling of complex shift parameters in the low-rank Cholesky factor ADI method. *Numerical Algorithms*, 62(2):225–251, 2013. doi:10.1007/s11075-012-9569-7.
- [7] P. Benner, J.-R. Li, and T. Penzl. Numerical solution of large-scale Lyapunov equations, Riccati equations, and linear-quadratic optimal control problems. *Numerical Linear Algebra with Applications*, 15(9):755–777, 2008. doi:10.1002/nla.622.
- [8] P. Benner and J. Saak. Numerical Solution of Large and Sparse Continuous Time Algebraic Matrix Riccati and Lyapunov Equations: A State of the Art Survey. 2013. <http://www.mpi-magdeburg.mpg.de/preprints/2013/MPIMD13-07.pdf>.
- [9] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions On Mathematical Software*, 30(2):196–199, June 2004. <http://dx.doi.org/10.1145/992200.992206>.
- [10] T.E. Djaferis and S.K. Mitter. Exact solution to Lyapunov's equation using algebraic methods. In *Decision and Control including the 15th Symposium on Adaptive Processes, 1976 IEEE Conference on*, volume 15, pages 1194–1200, 1976. <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4045776&tag=1>.
- [11] Python Software Foundation. Extending and Embedding the Python Interpreter. <http://docs.python.org/3/extending/>, October 2013.

-
- [12] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010. <http://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/HPC4SE/>.
- [13] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. <http://www.scipy.org/>, 2001–2013.
- [14] Kleen, A. numactl. <http://oss.sgi.com/projects/libnuma/>, 2002,2004. GNU General Public License, v.2 .
- [15] J. Li and J. White. Low Rank Solution of Lyapunov Equations. *SIAM Journal on Matrix Analysis and Applications*, 24(1):260–280, 2002. doi:10.1137/S0895479801384937.
- [16] MATLAB. *Version 8.0.0.783 (R2012b)*. The MathWorks Inc., Natick, Massachusetts, 2012.
- [17] T. E. Oliphant. *Guide to NumPy*. Trelgol Publishing, 2006. <http://numpy.scipy.org>.
- [18] T. Penzl. LYAPACK users guide. *Technical Report SFB393/00-33, Sonderforschungsbereich 393 Numerische Simulation auf massiv parallelen Rechnern, TU Chemnitz*, 2000. <http://www.tu-chemnitz.de/sfb393/sfb00pr.html>.
- [19] T. Rauber and G. Rünger. Performance Analysis of Parallel Programs. In *Parallel Programming*, pages 151–196. Springer Berlin Heidelberg, 2010.
- [20] J. Saak. *Efficient Numerical Solution of Large Scale Algebraic Matrix Equations in PDE Control and Model Order Reduction*. PhD thesis, Technische Universität Chemnitz, 2009.
- [21] Stichting Mathematisch Centrum, Amsterdam, The Netherlands. Python. <http://www.python.org>, 1991-1995.
- [22] The SciPy community. NumPy and SciPy Documentation: `scipy.sparse.linalg.spsolve`. <http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html>, 2013.
- [23] N. Truhar and K. Veselić. An Efficient Method for Estimating the Optimal Dampers’ Viscosity for Linear Vibrating Systems Using Lyapunov Equation. *SIAM Journal on Matrix Analysis and Applications*, 31(1):18–39, 2009. doi:10.1137/070683052.
- [24] Eugene Wachspress. In *The ADI Model Problem*. Springer New York, 2013. http://dx.doi.org/10.1007/978-1-4614-5122-8_5.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort, Datum, Unterschrift