Software zur hocheffizienten Lösung von Intervallgleichungssystemen mit C-XSC

Vom Fachbereich C der Bergischen Universität Wuppertal
angenommene Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

von

Michael Zimmer

Februar 2013

Bergische Universität Wuppertal
Wissenschaftliches Rechnen / Softwaretechnologie

Referent:	Prof. Dr. Walter Krämer
Korreferent:	Prof. Dr. Andreas Frommer
Eingereicht am:	29. November 2012
Tag der mündlichen Prüfung:	13. Februar 2013

Danksagung: Mein Dank gilt allen, die mich bei der Erstellung dieser Arbeit, in welcher Form auch immer, unterstützt haben! Danke an meine Familie und insbesondere meine Eltern, die mir mein Studium überhaupt erst ermöglicht haben. Danke an Prof. Dr. Walter Krämer für die wunderbare Betreuung dieser Arbeit und an meine Kollegen Dr. Werner Hofschuster und Gabriele Birkenfeld für ihre Unterstützung und die angenehme Arbeitsatmosphäre. Danke auch an Prof. Dr. Andreas Frommer für die Zweitbegutachtung dieser Arbeit. Und schließlich gilt mein besonderer Dank Anna, dafür dass sie immer für mich da war.

Für meinen Vater, in memoriam.

Inhaltsverzeichnis

1. Einleitung

Das Lösen linearer Gleichungssysteme ist eines der am häufigsten auftretenden Probleme in der Numerik, welches in vielen praktischen Problemstellungen auftaucht. Es existiert eine Vielzahl von Methoden für die Lösung linearer Gleichungssysteme, von direkten Methoden wie der LU- und Cholesky-Zerlegung [35] bis zu iterativen Methoden wie z.B. Mehrgitterverfahren und Krylov-Unterraum-Verfahren [93].

Durch den enormen Anstieg der Rechenleistung moderner Rechner und das Aufkommen paralleler Computer, zunächst vor allem in Form von in einem Netzwerk zusammengeschlossenen Rechnern (Cluster- oder auch Supercomputern), in den letzten Jahren zunehmend auch in Form von Mehrkernprozessoren und GPUs (*Graphics Processing Unit*) in handelsüblichen PCs, können heutzutage auch sehr große Gleichungssysteme vergleichsweise schnell gelöst werden. Dafür nötig sind performante Implementierungen geeigneter Verfahren, welche durch eine effiziente Parallelisierung die vorhandenen Ressourcen optimal ausnutzen.

Darüber hinaus haben auch spezielle Methoden für dünn besetzte lineare Gleichungssysteme, also Systeme, bei denen die Systemmatrix zu einem großen Teil mit Nullen gefüllt ist, immens an Bedeutung gewonnen. Solche Systeme treten vielfach in der Praxis auf, insbesondere bei der Diskretisierung partieller Differentialgleichungen. Spezielle Algorithmen und ihre Implementierungen, welche die Struktur solcher Systeme ausnutzen, können den Speicher- und Rechenzeitbedarf zur Berechnung einer Lösung um mehrere Größenordnungen reduzieren und somit die Arbeit mit enorm großen Systemen ermöglichen.

Auf dem Rechner treten dabei naturgemäß durch die Verwendung von Gleitkommazahlen Rundungsfehler auf. Vielfach sind Abschätzungen für die bei den jeweiligen Verfahren entstehenden Fehler bekannt, die jedoch oft recht grob oder nicht sicher sind. Das Teilgebiet der Verifikationsnumerik beschäftigt sich mit der verifizierten Einschließung von Lösungen, d.h. der Bestimmung von Ober- und Unterschranken der auftretenden Fehler auf der Maschine, deren Korrektheit automatisch verifiziert - also durch den Rechner mathematisch bewiesen - wird. Bei linearen Gleichungssystemen schließt die Bestimmung einer solchen verifizierten Lösungseinschließung auch den Beweis der Regularität der Systemmatrix mit ein.

Ein grundlegendes Werkzeug der Verifikationsnumerik ist die Intervallarithmetik. Mit ihrer Hilfe können mathematische Problemstellungen so formuliert werden, dass es sich bei den Eingangsdaten (also bei linearen Gleichungssystemen der Systemmatrix und der rechten Seite) um Intervalle handelt. Das Problem wird dann simultan für alle durch die Intervalleinträge beschriebenen Punktprobleme gelöst. Auf diese Weise können auch Unsicherheiten in den Eingangsdaten modelliert werden, welche z.B. durch Messfehler oder durch die Konvertierung reeller Zahlen in binäre Gleitkommazahlen entstehen.

Diese Arbeit beschäftigt sich mit der verifizierten Lösung linearer (Intervall-) Gleichungssysteme, also Systemen der Form

$$Ax = b$$

mit bekannten $A \in \mathbb{K}^{n\times n}$ und $b \in \mathbb{K}^n$, $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}, \mathbb{IR}, \mathbb{IC}\}$, sowie der zu bestimmenden Einschließung $x \in \mathbb{IK}^n$, $\mathbb{IK} \in \{\mathbb{IR}, \mathbb{IC}\}$, der exakten Lösung (nähere Erläuterungen zur Notation und den mathematischen Grundlagen folgen in Abschnitt 2.1). Als Spezialfall werden auch parameterabhängige Systeme betrachtet, bei denen A und b affin-linear von Parametern abhängen, die innerhalb eines vorgegebenen Intervalls variieren.

Die berechneten Einschließungen haben eine andere Qualität als übliche Näherungslösungen, was naturgemäß mit einem höheren Rechenaufwand einhergeht. Daher steht in dieser Arbeit insbesondere eine effiziente Implementierung moderner Methoden zur Lösung dieser Problemstellung im Vordergrund. Solche Implementierungen werden in der Folge auch kurz als Löser bezeichnet. Ein weiterer Fokus ist eine möglichst hohe Qualität der berechneten Einschließungen, so dass bei Punktsystemen nahezu alle Ziffern der berechneten Unter- und Oberschranke übereinstimmen.

Als Grundlage dient dabei die C++ Bibliothek C-XSC (*eXtended Scientific Computing*) [59, 75], welche eine Vielzahl von Werkzeugen aus der Verifikationsnumerik zur Verfügung stellt. Aufgrund ihres Alters und einer ursprünglich anderen Zielsetzung für die Bibliothek, gibt es viele Bereiche, in denen C-XSC für einen effektiven Einsatz im High-Performance-Computing deutlich erweitert und überarbeitet werden muss. Die notwendigen Änderungen an der C-XSC Bibliothek sind daher neben der Implementierung von Verfahren zur verifizierten Lösung linearer Gleichungssysteme ein weiterer Kernpunkt dieser Arbeit.

1.1. Bisherige Arbeiten

Bevor der Fokus dieser Arbeit genauer eingegrenzt wird, soll im Folgenden zunächst ein kurzer Überblick zu einigen bisherigen Arbeiten im Bereich der verifizierten Lösung linearer Gleichungssysteme, vor allem hinsichtlich der Implementierung entsprechender Methoden gegeben werden. Besonderes Augenmerk wird dabei auf Arbeiten, in welchen C-XSC genutzt wird bzw. welche Erweiterungen der C-XSC Bibliothek behandeln, gelegt. Die wesentlichen theoretischen Grundlagen aus diesen Arbeiten sowie Vergleiche mit den hier angesprochenen Implementierungen werden in den folgenden Kapiteln genauer beschrieben.

Das auch heute noch gebräuchliche Verfahren für die verifizierte Lösung allgemeiner dicht besetzter (Intervall-)Gleichungssysteme wurde von Rump in seiner Dissertation [123] erarbeitet. Es basiert auf der Einschließung des Fehlers einer zuvor berechneten Näherungslösung (auch Defekt genannt) des betrachteten Systems mit Hilfe des Krawczyk-Operators [85]. Die so berechnete Lösung ist eine Außeneinschließung der Intervallhülle der exakten Lösungsmenge, deren Bestimmung für Intervall-Gleichungssysteme NP-schwierig ist [86]. Allgemeine theoretische Grundlagen zu Intervallgleichungssystemen fasst Neumaier [102] zusammen.

Für dünn besetzte Gleichungssysteme ist die Benutzung spezieller Methoden sinnvoll. Eine Anpassung der Methode für dicht besetzte Systeme, wie sie u.a. Cordes [30,31] verwendet, ist zwar möglich, ihre Anwendbarkeit ist aber, wie auch im Verlauf dieser Arbeit noch gezeigt wird, recht stark eingeschränkt. Eine alternative Methode stammt wiederum von Rump [126]. Sie basiert auf einer normweisen Abschätzung des Defekts über die Bestimmung einer unteren Schranke des kleinsten Singulärwertes der Systemmatrix. Dieses Vorgehen ist vor allem für symmetrisch positiv definite Systeme gut geeignet.

Rump hat die von ihm erarbeiteten Methoden in seinem Softwarepaket Intlab [130] implementiert. Intlab ist ein weit verbreitetes Zusatzpaket für Matlab zur Arbeit mit Intervallen und bietet eine Vielzahl weiterer Werkzeuge aus der Verifikationsnumerik. Es ist darüber hinaus durch die Nutzung optimierter Bibliotheken (BLAS und LAPACK), welche hier erstmals im Rahmen der Verifikationsnumerik verwendet wurden, sehr schnell.

Von Arbeitsgruppen an den brasilianischen Universitäten in Porto Alegre und Passo Fundo wurden einige Arbeiten zur Implementierung paralleler verifizierender Löser für Cluster-Computer veröffentlicht. Die wesentlichen Ergebnisse fasst Kolberg [77] in ihrer Dissertation zusammen.

Auch für C-XSC bzw. den Vorgänger Pascal-XSC existieren bereits einige Implementierungen von Rumps Methode für dicht besetzte Gleichungssysteme:

- Ein Löser für reelle Punktsysteme wurde von Hammer et al. im Rahmen der sogenannten *Toolbox for Verified Computing I* [53] erstellt.
- Eine Intervallversion dieses Lösers mit einigen zusätzliche Funktionalitäten wurde von Carlos Hölbig [60] implementiert.
- Eine Anpassung dieser Methode für (dünn besetzte) Bandsysteme wurde im Rahmen des zweiten Toolbox-Bandes [83] für Pascal-XSC erstellt. Das dort beschriebene Vorgehen ist nur wenig bekannt und wird im Rahmen dieser Arbeit neu implementiert.
- Für parameterabhängige Gleichungssysteme wurde eine Anpassung der Rumpschen Methode von Popova implementiert [114, 115].

Die hier aufgezählten Löser wurden allerdings alle für ältere C-XSC Versionen implementiert und verwenden keine der in dieser Arbeit beschriebenen Optimierungen von C-XSC. Sie sind daher im Allgemeinen deutlich langsamer als vergleichbare Implementierungen (soweit vorhanden) und auch in ihrem Funktionsumfang meist eingeschränkt.

Verschiedene Arbeiten haben sich bereits mit der Parallelisierung von verifizierenden Algorithmen unter Verwendung von C-XSC und Pascal-XSC bzw. den in den XSC-Sprachen verwendeten Konzepten beschäftigt. Dazu zählen zum einen die Arbeiten von Davidenkoff [33] und Kersten [72], welche sich mit eher allgemeinen Aspekten der Parallelisierung verifizierender Berechnungen auseinandersetzen, sowie die Implementierung paralleler Programme für konkrete Problemstellungen, wie z.B. die Arbeiten zur globalen Optimierung von Berner [18] und Wiethoff [142].

Einige Erweiterungen von C-XSC im Hinblick auf das High Performance Computing wurden von Grimmer in seiner Dissertation erarbeitet [50]. Dabei wurde auch erstmalig

ein paralleler verifizierender Löser für lineare Intervallgleichungssysteme für Cluster-Computer implementiert. Der Autor erstellte im Rahmen seiner Master Thesis [146] erstmals laufzeiteffiziente Löser für dicht besetzte Gleichungssysteme für C-XSC (sowohl seriell als auch parallel), deren Performance an Intlab heran reicht. Einige der dort verwendeten Methoden werden im Rahmen dieser Arbeit in die C-XSC Bibliothek übernommen und erweitert. Ebenso dienen die dort beschriebenen Löser als Grundlage für die erweiterten und verbesserten Versionen der dicht besetzten Löser in dieser Arbeit.

1.2. Themen dieser Arbeit

Im Folgenden werden die im Rahmen dieser Arbeit untersuchten Bereiche genauer erläutert. Zwei grundlegende Kernpunkte sind dabei zu unterscheiden: Zum einen Erweiterungen der Kernfunktionalität der C-XSC Bibliothek selbst, zum anderen auf diesen Erweiterungen aufbauende Löser zur verifizierten Lösung linearer Gleichungssysteme.

1.2.1. Erweiterungen von C-XSC

Allgemeines Ziel bei den Erweiterungen von C-XSC im Rahmen dieser Arbeit ist, die Bibliothek um für einen Einsatz im High Performance Computing attraktive Funktionalitäten zu ergänzen. Diese Erweiterungen werden zwar insbesondere für die späteren Implementierungen der Gleichungssystem-Löser benötigt, können aber alle auch allgemein wertvolle Werkzeuge für vielfältige andere Aufgabenstellungen sein. Das Augenmerk liegt besonders auf folgenden Punkten:

- Die flexiblere Berechnung von Skalarprodukten und Skalarproduktausdrücken durch wählbare Genauigkeit. Hierzu zählt die Unterstützung einfacher Gleitkommarechnung und insbesondere hochoptimierter BLAS-Routinen.
- Spezielle Datentypen für dünn besetzte Vektoren und Matrizen mit entsprechenden Grundoperationen.
- Anpassungen für den Einsatz in der parallelen Programmierung, sowohl mit geteiltem als auch verteiltem Speicher.

Bei all diesen Veränderungen steht dabei im Vordergrund, dass sämtliche Erweiterungen optional nutzbar sind, also das Standardverhalten der Bibliothek nicht verändern. Dadurch können alte C-XSC Programme weiterhin ohne Anpassung kompiliert werden und liefern identische Ergebnisse zu vorherigen Versionen.

1.2.2. Löser für lineare Gleichungssysteme

Mit Hilfe der Erweiterungen von C-XSC sollen laufzeiteffiziente Implementierungen von Algorithmen zur verifizierten Lösung linearer Gleichungssysteme erstellt werden. Alle Löser sollen dabei die vier C-XSC Grunddatentypen (reell, komplex und entsprechende Intervalle) unterstützen und möglichst hohe Genauigkeit und Geschwindigkeit bieten.

Weiterhin sollen durch eine effiziente Parallelisierung die Ressourcen moderner Rechner möglichst gut ausgenutzt werden. Sofern bereits andere Implementierungen der verwendeten Algorithmen verfügbar sind, ist das Ziel, entweder bessere oder zumindest gleichwertige Geschwindigkeit bei höherer Genauigkeit und Flexibilität zu erreichen.

Im Rahmen dieser Arbeit werden dabei folgende Klassen von Gleichungssystemen betrachtet:

- Allgemeine dicht besetzte Systeme.
- Dich besetzte Systeme, deren Einträge affin-linear von Intervallparametern abhängen.
- Allgemeine dünn besetzte Systeme.

Genauere Erläuterungen und Literaturhinweise folgen im Verlauf dieser Arbeit in den jeweiligen Kapiteln.

1.3. Aufbau dieser Arbeit

Die Strukturierung dieser Arbeit orientiert sich im Wesentlichen an den in Abschnitt 1.2 genannten zentralen Themen. Für einen besseren Überblick wird hier eine Kurzbeschreibung der folgenden Kapitel angegeben:

- In Kapitel 2 werden zunächst die wichtigsten Grundlagen für diese Arbeit kurz besprochen. Dazu zählen insbesondere mathematische Grundlagen zur Intervallrechnung und zur Verifikationsnumerik allgemein sowie die Einführung der zugehörigen Notationselemente. Neben den mathematischen Grundlagen werden weiterhin einige Begriffe aus der parallelen Programmierung eingeführt und es wird ein Überblick über C-XSC und andere verwendete numerische Bibliotheken gegeben. Außerdem wird kurz auf Compileroptimierungen und das Laufzeitverhalten von Programmen aus dem Bereich der Verifikationsnumerik eingegangen.
- Nach Besprechung der Grundlagen folgt in Kapitel 3 der erste große Teil dieser Arbeit, die Erweiterung der C-XSC Bibliothek um für die spätere Erstellung der verifizierenden Löser wichtige Werkzeuge. Dazu zählen insbesondere Skalarprodukte in K-facher Arbeitsgenauigkeit, BLAS- und LAPACK-Unterstützung, sowie Datentypen für dünn besetzte Matrizen und Vektoren. Weiterhin wird auf einige für eine sinnvolle Parallelisierung von C-XSC Programmen nötige Änderungen und Erweiterungen eingegangen.
- Im Anschluss folgt die Erläuterung der im Rahmen dieser Arbeit implementierten verifizierenden Löser für dicht besetzte Gleichungssysteme in Kapitel 4. Darunter fallen zum einen Löser für allgemeine dicht besetze Systeme, die sowohl in einer seriellen bzw. mit Multi-Threading parallelisierten Version als auch in einer angepassten Version für Systeme mit verteiltem Speicher vorliegen. Weiterhin werden zwei Löser für parameterabhängige Systeme mit verschiedenen Lösungsansätzen

beschrieben. Dabei wird jeweils auf die grundlegenden theoretischen Überlegungen sowie insbesondere auf die Implementierung der Löser eingegangen. Ebenso werden ausführliche Tests mit entsprechenden Zeitmessungen und der Auswertung der numerischen Ergebnisse durchgeführt.

- In Kapitel 5 werden dann spezielle Löser für dünn besetzte Systeme beschrieben. Einer dieser Löser verwendet eine Modifikation des Ansatzes für allgemeine dicht besetzte Systeme, während ein anderer Löser auf einer normweisen Abschätzung des Defekts basiert. Sowohl die theoretischen Überlegungen zu den verwendeten Methoden als auch die eigentliche Implementierung der beiden Löser werden wiederum ausführlich dargelegt. In detaillierten Tests werden auch hier Laufzeitverhalten und Ergebnisqualität der Löser für dünn besetzte Systeme untersucht.

- Im abschließenden Kapitel 6 wird ein allgemeines Fazit dieser Arbeit gezogen. Zusätzlich wird nochmals eine kurze Zusammenfassung der Arbeit angegeben und es wird auf einige Erweiterungsmöglichkeiten und für die Zukunft geplante Änderungen von C-XSC und den verschiedenen Lösern eingegangen.

2. Grundlagen

In diesem Kapitel werden die wichtigsten Grundlagen aus einigen für diese Arbeit bedeutsamen Bereichen zusammengefasst. Dabei werden die in den folgenden Kapiteln verwendeten Fachbegriffe definiert und eine einheitliche Notation für diese Arbeit eingeführt. Neben mathematischen Grundlagen werden auch eher technische Themen, z.B. zur parallelen Programmierung und den verwendeten Bibliotheken, behandelt.

2.1. Mathematische Grundlagen und Notation

Im Folgenden werden die wichtigsten mathematischen Grundlagen für diese Arbeit erläutert. Zunächst wird allgemein auf die Gleitkommarechnung und insbesondere auf den IEEE 754 Standard [14] eingegangen. Danach wird die Intervallarithmetik allgemein eingeführt. Lineare Intervall-Gleichungssysteme werden, da sie das zentrale Thema dieser Arbeit sind, in einem eigenen Abschnitt behandelt. Schließlich werden einige Grundprinzipien der Verifikationsnumerik beschrieben. In allen Abschnitten wird jeweils auf weiterführende Literatur verwiesen.

Zunächst noch einige Anmerkungen zur Notation: Matrizen werden allgemein mit Großbuchstaben bezeichnet (z.B. A), Vektoren und Skalare mit Kleinbuchstaben (z.B. x), wobei für Skalare teilweise griechische Buchstaben verwendet werden (z.B. α). Die Menge aller reellen Vektoren der Dimension n mit Elementen aus einer Menge M wird mit M^n notiert, die Menge aller $m \times n$ Matrizen mit Elementen aus einer Menge M wird mit $M^{m \times n}$ notiert. Die Einheitsmatrix wird mit I angegeben. Das i-te Element eines Vektors x wird mit x_i notiert, das Element in Zeile i und Spalte j einer Matrix A mit a_{ij} oder, falls zur Klarheit erforderlich (bei Verwendung von Formeln für die Indizes), in der Form $a_{i+1,2j}$. Ein Ausschnitt aus der Matrix wird über Indizes mit Doppelpunkten notiert, z.B. steht $A_{1:10,1:10}$ für den linken oberen 10×10 Block der Matrix A. Ganze Zeilen oder Spalten einer Matrix werden mittels $A_{\bullet j}$ für die j-te Spalte bzw. $A_{i\bullet}$ für die i-te Zeile bezeichnet. Fortlaufende Indizes für Vektoren und Matrizen, z.B. bei Iterationen, werden in der Regel hochgestellt und in Klammern angefügt, also z.B. $A^{(k)}$.

2.1.1. Gleitkommarechnung

Auf Computern kann naturgemäß, aufgrund der stets beschränkten Ressourcen, nur mit einer endlichen Menge von Zahlen gerechnet werden. Die Menge der reellen Zahlen wird deshalb für Berechnungen auf der Maschine auf eine (endliche) Menge von Gleitkommazahlen abgebildet. Diese Gleitkommazahlen (im Folgenden teilweise auch als Maschi-

nenzahlen bezeichnet) haben allgemein die Form

$$x = \pm 0.x_1 x_2 \dots x_t \cdot B^e,$$

wobei $x_i \in \{0, 1, \dots, B-1\}$, $i = 1, \dots, t$, als Mantisse, $B \in \mathbb{N}, B \geq 2$ als Basis und e mit $e_{min} \leq e \leq e_{max}$ und $e_{min}, e, e_{max} \in \mathbb{Z}$ als Exponent bezeichnet werden, mit dem minimalen Exponenten e_{min} und dem maximalen Exponenten e_{max}. Die Menge $\mathbb{R}$ der reellen Zahlen wird also auf der Maschine in ein Raster $R = R(B, t, e_{min}, e_{max})$ von Gleitkommazahlen überführt. Die Menge der Maschinenzahlen wird mit $\mathbb{F}$ bezeichnet.

Diese Darstellung einer Gleitkommazahl ist nicht eindeutig, z.B. kann in einem System $R(10, 3, -3, 3)$ die Zahl 12 sowohl als $0.12 \cdot 10^2$ als auch als $0.012 \cdot 10^3$ dargestellt werden. Da eine eindeutige Darstellung im Allgemeinen von Vorteil ist, wird die Darstellung normalisiert, indem man $x_1 \neq 0$ fordert. Die Darstellung wird dadurch eindeutig. Als Spezialfall wird festgelegt, dass

$$0 = 0.000 \dots 000 \cdot B^0$$

gilt.

Um eine reelle Zahl sinnvoll in eine Gleitkommazahl überführen zu können, ist eine entsprechende Rundungsfunktion nötig.

Definition 2.1. *Eine Abbildung $\bigcirc : \mathbb{R} \to \mathbb{F}$ wird als Rundung bezeichnet, falls*

$$\begin{array}{rcll} \bigcirc x & = & x & \forall x \in \mathbb{F} \quad \textit{und} \\ x \leq y & \Rightarrow & \bigcirc x \leq \bigcirc y & \forall x, y \in \mathbb{R} \end{array}$$

gelten.

Insbesondere werden die folgenden Rundungen unterschieden:

$\square$: Rundung zur nächstgelegenen Gleitkommazahl
$\bigtriangledown$: Rundung in Richtung $-\infty$
$\bigtriangleup$: Rundung in Richtung $+\infty$

Definition 2.2. *Für jede Grundoperation $\circ \in \{+, -, *, /\}$ und Operanden $a, b \in \mathbb{F}$ existieren entsprechend gerundete Operationen*

$$a \circledcirc b = \bigcirc(a \circ b).$$

Insbesondere gibt es zu jeder Grundoperation also jeweils eine Variante mit Rundung zur nächsten Gleitkommazahl, Richtung ∞ und Richtung $-\infty$.

Alle Gleitkommaoperationen sind gerundete Operationen und werden entsprechend Definition 2.2 notiert. Alle Operationen, die in dieser Arbeit in Algorithmen benutzt werden, sind grundsätzlich als Gleitkommaoperationen zu verstehen, sofern nichts anderes angegeben wird. Gelegentlich wird auch die Schreibweise $\mathrm{fl}(\cdot)$ verwendet, um deutlich zu machen, dass alle Operationen in der Klammer gerundete Gleitkommaoperationen darstellen. Grundsätzlich wird dabei davon ausgegangen, dass die gerundete Operation

mit Rundung zur nächstgelegenen Gleitkommazahl verwendet wird, es sei denn, es wird explizit ein anderer Rundungsmodus angegeben.

Auf nahezu allen Rechnern wird heute der IEEE 754 Standard für Gleitkommazahlen [14] verwendet. In diesem werden ein binärer Datentyp in einfacher Genauigkeit (*single precision*) mit $B = 2$, $t = 24$, $e_{min} = -126$ und $e_{max} = 127$ sowie ein binärer Datentyp in doppelter Genauigkeit (*double precision*) mit $B = 2$, $t = 53$, $e_{min} = -1023$ und $e_{max} = 1024$ definiert. Da mit normalisierten Zahlen gearbeitet wird, gilt stets $x_0 = 1$. Dieses Bit wird daher nicht explizit gespeichert (*hidden bit*). Bei der Speicherung des Exponenten wird ein sogenannter Bias von 126 bzw. 1023 verwendet, d.h. zum eigentlichen Exponenten wird e_{min} hinzu addiert, so dass der gespeicherte Exponent stets größer oder gleich Null ist. Zusätzlich ist jeweils noch ein Vorzeichenbit nötig. Eine Gleitkommazahl in einfacher Genauigkeit hat somit eine Wortbreite von 32 Bit, während eine Gleitkommazahl in doppelter Genauigkeit eine Wortbreite von 64 Bit aufweist. Im neuesten IEEE 754 Standard wird für (binäre) Gleitkommazahlen in einfacher Genauigkeit die Bezeichnung *binary32* und für Gleitkommazahlen in doppelter Genauigkeit der Begriff *binary64* verwendet. In dieser Arbeit werden durchgehend die klassischen Bezeichnungen *single* und *double* benutzt.

Der Standard-Rundungsmodus im IEEE 754 Format ist die Rundung zur nächstgelegenen Gleitkommazahl. Der Rundungsmodus kann aber auf Rundung in Richtung $-\infty$, in Richtung ∞ oder zur 0 umgestellt werden. Bei handelsüblichen CPUs ist der Rundungsmodus nicht mit der einzelnen Operation verknüpft, d.h. es gibt keine expliziten Befehle für Operationen, die nicht den Standard-Rundungsmodus verwenden. Stattdessen kann er global, also für alle folgenden Operationen, umgestellt werden.

Zahlen, deren Betrag größer als die größte darstellbare Gleitkommazahl ist (für *double* ist dies z.B. ca. `1.798e+308`) liegen im Überlaufbereich, Zahlen, deren Betrag kleiner als die kleinste positive normalisierte Gleitkommazahl ist (für *double* ist dies z.B. ca. `2.225e-308`) liegen im Unterlaufbereich. Im IEEE 754 Standard wird vorgegeben, dass der Unterlaufbereich mit den sogenannten denormalisierten Zahlen aufgefüllt wird, bei welchen $e = e_{min}$ und $x_1 = 0$ gilt. Je näher man der 0 kommt, desto geringer wird also die Anzahl der darstellbaren Stellen, aber der direkte Sprung von der kleinsten positiven normalisierten Zahl zur 0 wird vermieden. Man spricht auch von einem schrittweisen Unterlauf (*gradual underflow*).

Bei der Verwendung von Gleitkommazahlen kommt es unweigerlich zu Rundungsfehlern, da das korrekte Ergebnis einer Gleitkommaoperation $a \circ b$, $a, b \in \mathbb{F}$ in der Regel keine Gleitkommazahl ist und somit eine Rundung vorgenommen werden muss. Der dabei maximal mögliche Fehler wird durch die relative Maschinengenauigkeit begrenzt. Diese ist gegeben durch

$$eps := \frac{1}{2} B^{1-t},$$

für *double* gilt also $eps = 2^{-53}$. Somit gilt für alle Operationen $\circ \in \{+, -, *, /\}$ und $a, b \in \mathbb{F}$ [58]:

$$fl(a \circ b) = (a \circ b)(1 + \delta), \quad |\delta| \leq eps.$$

Dabei wird allerdings davon ausgegangen, dass weder Unter- noch Überlauf auftreten.

Liegt das Ergebnis einer Gleitkomma-Addition oder -Subtraktion im Unterlaufbereich, so ist es - sofern *gradual underflow* verwendet wird - exakt [56]. Für $\circ \in \{*, /\}$ hingegen muss obiges Modell folgendermaßen abgeändert werden [58]:

$$fl(a \circ b) = (a \circ b)(1 + \delta) + \nu, \quad |\delta| \leq eps \quad |\nu| \leq eta.$$

Hierbei ist

$$eta := \frac{1}{2} B^{e_{min}-t}$$

der halbe Abstand zwischen zwei denormalisierten Zahlen im Unterlaufbereich (für *double* gilt $eta = 2^{-1076}$) und es gilt $\delta\nu = 0$ (tritt kein Unterlauf auf, so ist $\nu = 0$, ansonsten ist $\delta = 0$).

Hilfreich ist weiterhin die Einführung der Größe γ_n nach Higham [58]:

$$\gamma_n := \frac{n \cdot eps}{1 - n \cdot eps}, \quad n \in \mathbb{N}.$$

Dabei gilt, falls $n \cdot eps < 1$ (diese Bedingung ist bei Benutzung des IEEE *single*- oder *double*-Formats nahezu immer erfüllt, so dass im Folgenden bei Verwendung von γ_n stets $n \cdot eps < 1$ vorausgesetzt wird), für das Skalarprodukt zweier Vektoren $x, y \in \mathbb{R}^n$:

$$|x^T y - fl(x^T y)| \leq \gamma_n \sum_{i=1}^{n} |x_i y_i| = \gamma_n |x^T||y|.$$

Auf einigen Systemen (z.B. auf Basis von Intels IA-64 Architektur oder IBMs PowerPC Architektur) ist eine sogenannte *Fused Multiply and Add* (FMA) Operation vorhanden, welche eine Addition und Multiplikation in einem Arbeitsschritt mit nur einer Rundung ausführt. Bei Existenz einer solchen Anweisung gilt mit $a, b, c \in \mathbb{F}$

$$fl(a \pm b \cdot c) = (a \pm b \cdot c)(1 + \delta) + \nu, \quad |\delta| \leq eps, \quad |\nu| \leq eta,$$

wiederum mit $\delta\nu = 0$.

Zum Abschluss dieses Abschnitts soll noch die Einheit *ulp* in folgender Definition eingeführt werden.

Definition 2.3. *Die Einheit* ulp *(Unit in the Last Place) bezeichnet die Wertigkeit der letzten signifikanten Stelle einer Gleitkommazahl. Für die Gleitkommazahl*

$$a = M2^e$$

mit der Mantisse M der Länge t gilt also

$$1ulp = 2^{e-t}.$$

Somit kann also das Ergebnis einer Gleitkommarechnung, welches um nicht mehr als ein *ulp* vom exakten Resultat abweicht, als maximal genau angesehen werden.

Für eine sehr detaillierte Behandlung der Fehleranalyse bei der Benutzung von Gleitkommarechnung sei auf Higham [58] und das Standardwerk von Wilkinson [143] verwiesen. Eine ausführliche allgemeine Abhandlung über Gleitkommarechnung und verschiedene Gleitkommasysteme, insbesondere auch deren Implementierung in Hard- oder Software, findet sich bei Muller et al. [100].

2.1.2. Intervallarithmetik

Das wichtigste Werkzeug für die Verifikationsnumerik ist die Intervallarithmetik. In diesem Abschnitt sollen die wichtigsten Grundlagen zu Intervallen und die entsprechende Notation eingeführt werden. Dabei wird unter anderem auf reelle und komplexe Intervalle, verschiedene Darstellungsformen von Intervallen (Infimum-Supremum-Darstellung, Mittelpunkt-Radius-Darstellung und die Kreisscheiben-Arithmetik im Komplexen), Intervallvektoren und -matrizen sowie die Umsetzung von Intervallen auf dem Rechner eingegangen.

Definition 2.4. *Ein reelles Intervall ist definiert als die Menge*

$$\boldsymbol{x} := [\underline{x}, \overline{x}] := \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}.$$

Hierbei wird $\underline{x}$ als Infimum (auch Untergrenze oder Unterschranke) und $\overline{x}$ als Supremum (auch Obergrenze oder Oberschranke) bezeichnet.

Intervallgrößen werden als fett gedruckte Buchstaben notiert, also z.B. $\boldsymbol{x}$. Die Menge aller reellen, beschränkten und abgeschlossenen Intervalle wird mit $\mathbb{IR}$ bezeichnet. In dieser Arbeit wird stets, sofern nicht ausdrücklich etwas anderes angegeben wird, von beschränkten und abgeschlossenen Intervallen ausgegangen. Wird im Folgenden die einfache Bezeichnung Intervall verwendet, so ist in der Regel ein reelles Intervall gemeint. Ein Intervall $\boldsymbol{x}$, für welches $\underline{x} = \overline{x}$ gilt, wird als Punktintervall (oder auch dünnes Intervall) bezeichnet. Probleme, welche sich nur mit einfachen reellen oder komplexen Zahlen oder entsprechenden Punktintervallen beschäftigen, werden als Punktprobleme bezeichnet.

Da es sich bei Intervallen um Mengen handelt, werden die Mengenoperationen $=$, $\subset$, $\subseteq$, $\supset$, $\supseteq$, $\cup$ und $\cap$ für Intervalle im üblichen mengentheoretischen Sinne verwendet. Weiterhin werden folgende Operationen für Intervalle definiert.

Definition 2.5. *Sei $S \subset \mathbb{R}$ eine nicht leere beschränkte Menge und $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}$. Die Intervallhülle wird dann definiert als*

$$\square(S) := [\inf(S), \sup(S)].$$

Insbesondere gilt für die Intervallhülle von $\boldsymbol{x}$ und $\boldsymbol{y}$:

$$\square(\boldsymbol{x}, \boldsymbol{y}) = [\inf(\underline{x}, \underline{y}), \sup(\overline{x}, \overline{y})].$$

Das Innere der Menge S wird mit $\mathrm{int}(S)$ *bezeichnet. Liegt $\boldsymbol{x}$ im Inneren von $\boldsymbol{y}$, so gilt also*

$$\boldsymbol{x} \subset \mathrm{int}(\boldsymbol{y}) :\equiv \boldsymbol{x} \subset \boldsymbol{y} \wedge \underline{x} > \underline{y} \wedge \overline{x} < \overline{y}.$$

Die Einführung einer Intervallhülle ist insbesondere nötig, da die übliche mengenmäßige Vereinigung zweier Intervalle nicht immer auf ein Intervall gemäß Definition 2.4 führt.

Als nächstes werden die Grundoperationen $+,-,\cdot$ und $/$ für Intervalle eingeführt.

Definition 2.6. *Seien* $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$. *Für die Operationen* $\circ \in \{+, -, \cdot, /\}$ *gilt dann*

$$\boldsymbol{a} \circ \boldsymbol{b} := \{a \circ b \mid a \in \boldsymbol{a} \text{ und } b \in \boldsymbol{b}\}.$$

Für die Division muss dabei $0 \notin \boldsymbol{b}$ *gelten.*

Aus Definition 2.6 lassen sich die folgenden Formeln für die vier Grundoperationen ableiten:

$$\begin{array}{rcl}
\boldsymbol{a} + \boldsymbol{b} & = & [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \\
\boldsymbol{a} - \boldsymbol{b} & = & [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \\
\boldsymbol{a} \cdot \boldsymbol{b} & = & [\min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), \max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})] \\
\boldsymbol{a} / \boldsymbol{b} & = & [\min(\underline{a}/\underline{b}, \underline{a}/\overline{b}, \overline{a}/\underline{b}, \overline{a}/\overline{b}), \max(\underline{a}/\underline{b}, \underline{a}/\overline{b}, \overline{a}/\underline{b}, \overline{a}/\overline{b})], \ 0 \notin \boldsymbol{b}
\end{array}$$

Die einzelnen Fälle für die Multiplikation werden in Tabelle 2.1 aufgeführt, die für die Division in Tabelle 2.2. Bei einer konkreten Implementierung werden diese Tabellen mittels entsprechender Fallunterscheidungen umgesetzt. Es ist wichtig festzuhalten, dass die einzelnen Grundoperationen exakt sind.

$\boldsymbol{a} \cdot \boldsymbol{b}$	$\boldsymbol{b} \leq 0$	$0 \in \boldsymbol{b}$	$0 \geq \boldsymbol{b}$
$0 \geq \boldsymbol{a}$	$[\overline{a} \cdot \overline{b}, \underline{a} \cdot \underline{b}]$	$[\underline{a} \cdot \overline{b}, \underline{a} \cdot \underline{b}]$	$[\underline{a} \cdot \overline{b}, \overline{a} \cdot \underline{b}]$
$0 \in \boldsymbol{a}$	$[\overline{a} \cdot \underline{b}, \underline{a} \cdot \underline{b}]$	$[\min\{\underline{a} \cdot \overline{b}, \overline{a} \cdot \underline{b}\}, \max\{[\underline{a} \cdot \underline{b}, \overline{a} \cdot \overline{b}\}]$	$[\underline{a} \cdot \overline{b}, \overline{a} \cdot \overline{b}]$
$0 \leq \boldsymbol{a}$	$[\overline{a} \cdot \underline{b}, \underline{a} \cdot \overline{b}]$	$[\overline{a} \cdot \underline{b}, \overline{a} \cdot \overline{b}]$	$[\underline{a} \cdot \underline{b}, \overline{a} \cdot \overline{b}]$

Tabelle 2.1.: Intervallmultiplikation

Einige wichtige Eigenschaften der Grundoperationen werden im folgenden Satz zusammengefasst:

Satz 2.1. *Seien* $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{IR}$ *und* $\circ \in \{+, \cdot\}$. *Dann gelten:*

1. $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{b} \circ \boldsymbol{a}$ *(Kommutativität)*
2. $(\boldsymbol{a} \circ \boldsymbol{b}) \circ \boldsymbol{c} = \boldsymbol{a} \circ (\boldsymbol{b} \circ \boldsymbol{c})$ *(Assoziativität)*
3. $\boldsymbol{a} \cdot (\boldsymbol{b} + \boldsymbol{c}) \subseteq \boldsymbol{a} \cdot \boldsymbol{b} + \boldsymbol{a} \cdot \boldsymbol{c}$ *(Subdistributivität)*

$\boldsymbol{a}/\boldsymbol{b}$	$\boldsymbol{b} < 0$	$\boldsymbol{b} > 0$
$0 \geq \boldsymbol{a}$	$[\overline{a}/\underline{b}, \underline{a}/\overline{b}]$	$[\underline{a}/\underline{b}, \overline{a}/\overline{b}]$
$0 \in \boldsymbol{a}$	$[\overline{a}/\overline{b}, \underline{a}/\overline{b}]$	$[\underline{a}/\underline{b}, \overline{a}/\underline{b}]$
$0 \leq \boldsymbol{a}$	$[\overline{a}/\overline{b}, \underline{a}/\underline{b}]$	$[\underline{a}/\overline{b}, \overline{a}/\underline{b}]$

Tabelle 2.2.: Intervalldivision

Ein Beweis wird z.B. bei Alefeld/Herzberger [9] angegeben. Voraussetzungen, die bei Punkt 3 Gleichheit (also Distributivität) bewirken, werden bei Ratschek [117] und Spaniol [137] diskutiert.

Neben den Grundoperationen werden einstellige Operationen, zu denen insbesondere die mathematischen Standardfunktionen zählen (siehe z.B. Braune/Krämer [27]), wie folgt definiert:

Definition 2.7. *Sei* $\Phi : \mathbb{R} \mapsto \mathbb{R}$, $\boldsymbol{x} \in \mathbb{IR}$ *und* Φ *für jedes* $x \in \boldsymbol{x}$ *definiert und stetig. Die entsprechende Operation* $\boldsymbol{\Phi} : \mathbb{IR} \mapsto \mathbb{IR}$ *für Intervalle wird dann definiert als*

$$\boldsymbol{\Phi}(\boldsymbol{x}) := [\min_{x \in \boldsymbol{x}}(\Phi(x)), \max_{x \in \boldsymbol{x}}(\Phi(x))].$$

Man erhält die *Intervallerweiterung* $\boldsymbol{f} : \mathbb{IR} \mapsto \mathbb{IR}$ einer Funktion $\mathrm{f} : \mathbb{R} \mapsto \mathbb{R}$, die sich aus den oben definierten Operationen zusammensetzt, durch Ersetzen dieser Operationen durch die entsprechenden Intervalloperationen (sofern diese für die betrachteten Intervalle wohldefiniert sind). Eine solche Intervallerweiterung $\boldsymbol{f}(\boldsymbol{x})$ schließt den tatsächlichen Wertebereich $\{f(x) \mid x \in \boldsymbol{x}\}$ der Funktion über dem betrachteten Intervall verlässlich ein.

Satz 2.2. *(Inklusionsmonotonie / Teilmengeneigenschaft) Seien* $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d} \in \mathbb{IR}$, $\circ \in \{+, -, \cdot, /\}$, $\boldsymbol{f} : \mathbb{IR} \mapsto \mathbb{IR}$. *Dann gelten:*

1. $\boldsymbol{a} \subseteq \boldsymbol{b}, \boldsymbol{c} \subseteq \boldsymbol{d} \Rightarrow \boldsymbol{a} \circ \boldsymbol{c} \subseteq \boldsymbol{b} \circ \boldsymbol{d}$,
2. $\boldsymbol{a} \subseteq \boldsymbol{b} \Rightarrow \boldsymbol{f}(\boldsymbol{a}) \subseteq \boldsymbol{f}(\boldsymbol{b})$.

Insbesondere gelten also $a \in \boldsymbol{a}, b \in \boldsymbol{b} \Rightarrow a \circ b \in \boldsymbol{a} \circ \boldsymbol{b}$ und $a \in \boldsymbol{a} \Rightarrow \mathrm{f}(a) \in \boldsymbol{f}(\boldsymbol{a})$ (hier sei $\boldsymbol{f}$ die Intervallerweiterung der Funktion $\mathrm{f} : \mathbb{R} \mapsto \mathbb{R}$). Dieser Satz folgt direkt aus Definition 2.6 und Definition 2.7. Im Allgemeinen wird eine Intervallerweiterung $\boldsymbol{f}$ einer Abbildung $\mathrm{f} : \mathbb{R} \mapsto \mathbb{R}$ zu (oftmals beträchtlichen) Überschätzungen des tatsächlichen

Wertebereichs $\{f(x) \mid x \in \boldsymbol{x}\}$ führen. Daher sind beim Entwurf von Algorithmen für die Verifikationsnumerik oft andere Ansätze nötig (siehe Abschnitt 2.1.4).

Für den Umgang mit Intervallen sind einige Größen von Vorteil, die im Folgenden definiert werden:

Definition 2.8. *Seien $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$. Dann lassen sich folgende Größen definieren:*

1. $\operatorname{mid}(\boldsymbol{a}) := \frac{\underline{a}+\overline{a}}{2}$ *(Mittelpunkt)*
2. $\operatorname{rad}(\boldsymbol{a}) := \frac{\overline{a}-\underline{a}}{2}$ *(Radius)*
3. $\operatorname{diam}(\boldsymbol{a}) := \overline{a} - \underline{a}$ *(Durchmesser)*
4. $\operatorname{abs}(\boldsymbol{a}) = |\boldsymbol{a}| := \max\{|a| \mid a \in \boldsymbol{a}\} = \max\{|\underline{a}|, |\overline{a}|\}$ *(Betrag)*
5. $< \boldsymbol{a} >:= \begin{cases} 0 & \textit{falls } 0 \in \boldsymbol{a} \\ \min\{|\underline{a}|, |\overline{a}|\} & \textit{sonst} \end{cases}$ *(Betragsminimum)*
6. $\operatorname{diam}_{\mathrm{rel}}(\boldsymbol{a}) := \begin{cases} \frac{\operatorname{diam}(\boldsymbol{a})}{<\boldsymbol{a}>} & \textit{falls } 0 \notin \boldsymbol{a} \\ \operatorname{diam}(\boldsymbol{a}) & \textit{sonst} \end{cases}$ *(relativer Durchmesser)*
7. $\operatorname{dist}(\boldsymbol{a}, \boldsymbol{b}) := \max\{|\underline{a} - \underline{b}|, |\overline{a} - \overline{b}|\}$ *(Abstand)*
8. $\operatorname{relerr}(\boldsymbol{a}) := \begin{cases} \frac{\operatorname{rad}(\boldsymbol{a})}{\operatorname{mid}(\boldsymbol{a})} & \textit{falls } \operatorname{mid}(\boldsymbol{a}) \neq 0 \\ \operatorname{rad}(\boldsymbol{a}) & \textit{sonst} \end{cases}$ *(relativer Fehler)*

Eine Alternative zur bisher diskutierten Infimum-Supremum-Darstellung von Intervallen ist die sogenannte Mittelpunkt-Radius-Darstellung:

Definition 2.9. *Seien $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IR}$, $m_a, r_a, m_b, r_b \in \mathbb{R}$, $r_a, r_b \geq 0$. Die (zur Infimum-Supremum-Darstellung gleichwertige) Mittelpunkt-Radius-Darstellung von $\boldsymbol{a}$ und $\boldsymbol{b}$ ist*

$$\begin{array}{rcccl} \boldsymbol{a} & = & < m_a, r_a > & = & \{x \in \mathbb{R} \mid |x - m_a| \leq r_a\} \\ \boldsymbol{b} & = & < m_b, r_b > & = & \{x \in \mathbb{R} \mid |x - m_b| \leq r_b\}. \end{array}$$

Die vier Grundoperationen sowie die Inversion sind für Intervalle in Mittelpunkt-Radius-Darstellung wie folgt definiert:

$$\begin{array}{rcll} \boldsymbol{a} & + & \boldsymbol{b} & := < m_a + m_b, r_a + r_b > \\ \boldsymbol{a} & - & \boldsymbol{b} & := < m_a - m_b, r_a + r_b > \\ \boldsymbol{a} & \cdot & \boldsymbol{b} & := < m_a \cdot m_b, |m_a| r_b + r_a |m_b| + r_a r_b > \\ 1 & / & \boldsymbol{b} & := < m_b/\alpha, r_b/\alpha > \ \textit{mit } \alpha := m_b^2 - r_b^2 \textit{ und } 0 \notin \boldsymbol{b} \\ \boldsymbol{a} & / & \boldsymbol{b} & := \boldsymbol{a} \cdot (1/\boldsymbol{b}) \end{array}$$

Ein Nachteil der Mittelpunkt-Radius-Darstellung ist, dass bei der Multiplikation und der Division $\boldsymbol{a}/\boldsymbol{b}$ eine Überschätzung des Ergebnisses der entsprechenden Potenzmengenoperation mit einem Faktor von maximal 1.5 auftreten kann (siehe Rump [128], Theorem

2.4). Die Mittelpunkt-Radius-Darstellung wird in einigen Intervall-Bibliotheken verwendet, u.a. bei Intlab [130], da sie deutliche Vorteile bei der effizienten Implementierung von entsprechenden Matrix- und Vektor-Berechnungen bietet. Nähere Details hierzu finden sich bei Rump [128, 130] sowie in der Beschreibung der C-XSC-BLAS-Routinen in Abschnitt 3.2. Soweit nichts anderes angegeben ist, wird in dieser Arbeit stets von der (auch in C-XSC üblichen, siehe Abschnitt 2.3) Infimum-Supremum-Darstellung ausgegangen.

Als nächstes werden komplexe Intervalle näher betrachtet. Allgemein unterscheidet man bei komplexen Intervallen zwischen komplexen Rechteckintervallen, welche eine Infimum-Supremum-Darstellung von Real- und Imaginärteil verwenden, und der sogenannten Kreisscheibenarithmetik, welche analog zur Mittelpunkt-Radius-Darstellung im Reellen einen komplexen Mittelpunkt und einen Radius zur Beschreibung des Intervalls verwendet. Im ersten Fall ergibt sich also ein achsenparalleles Rechteck in der komplexen Ebene, im zweiten Fall eine Kreisscheibe. Die Menge der komplexen Intervalle wird allgemein mit $\mathbb{IC}$ notiert, möchte man zwischen Rechteck- und Kreisscheibendarstellung unterscheiden, so steht $\mathbb{IC}_\square$ für die Menge der komplexen Rechteckintervalle und $\mathbb{IC}_\circ$ für die Menge der komplexen Kreisscheibenintervalle. Im Folgenden werden die verschiedenen Darstellungen komplexer Intervalle und die entsprechenden Grundoperationen eingeführt.

Definition 2.10. *Seien $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{u}, \boldsymbol{v} \in \mathbb{IR}$. Ein komplexes Rechteckintervall $\boldsymbol{z} \in \mathbb{IR}_\square$ ist definiert als*

$$\boldsymbol{z} := \boldsymbol{x} + i\boldsymbol{y} = \{(x, y) \in \mathbb{C} \mid \underline{x} \leq x \leq \overline{x}, \underline{y} \leq y \leq \overline{y}\}.$$

Für komplexe Rechteckintervalle $\boldsymbol{z} = \boldsymbol{x} + i\boldsymbol{y}$ und $\boldsymbol{w} = \boldsymbol{u} + i\boldsymbol{v}$ lassen sich die Grundoperationen folgendermaßen definieren:

$$\begin{array}{lcl} \boldsymbol{z} \ +_\square \ \boldsymbol{w} & := & \boldsymbol{x} + \boldsymbol{u} + i(\boldsymbol{y} + \boldsymbol{v}) \\ \boldsymbol{z} \ -_\square \ \boldsymbol{w} & := & \boldsymbol{x} - \boldsymbol{u} + i(\boldsymbol{y} - \boldsymbol{v}) \\ \boldsymbol{z} \ \cdot_\square \ \boldsymbol{w} & := & \boldsymbol{xu} - \boldsymbol{yv} + i(\boldsymbol{xv} + \boldsymbol{yu}) \\ \boldsymbol{z} \ /_\square \ \boldsymbol{w} & := & \frac{\boldsymbol{xu}+\boldsymbol{yv}}{\boldsymbol{u}^2+\boldsymbol{v}^2} + i\frac{\boldsymbol{yu}-\boldsymbol{xv}}{\boldsymbol{u}^2+\boldsymbol{v}^2} \end{array}$$

Dabei ist $\circ_\square$ mit $\circ = \{+, -, \cdot, /\}$ als Rechteckintervalloperation und die Operationen auf der rechten Seite sind als reelle Intervalloperationen gemäß Definition 2.6 zu verstehen.

Die Grundoperationen aus Definition 2.10 liefern zwar in jedem Fall verlässliche Einschließungen, diese sind aber nicht zwangsläufig optimal:

Satz 2.3. *Seien $\boldsymbol{z}, \boldsymbol{w} \in \mathbb{IC}_\square$. Dann gilt:*

1. $\{z \pm w \mid z \in \boldsymbol{z}, w \in \boldsymbol{w}\} = \boldsymbol{z} \pm_\square \boldsymbol{w}$
2. $\{z \cdot w \mid z \in \boldsymbol{z}, w \in \boldsymbol{w}\} \subseteq \boldsymbol{z} \cdot_\square \boldsymbol{w}$ *(Im Allgemeinen gilt hier keine Gleichheit, allerdings ist $\square(\boldsymbol{z} \cdot \boldsymbol{w}) = \boldsymbol{z} \cdot_\square \boldsymbol{w}$)*
3. $\{z/w \mid z \in \boldsymbol{z}, w \in \boldsymbol{w}\} \subseteq \boldsymbol{z}/_\square\boldsymbol{w}$ *(auch hier gilt im Allgemeinen keine Gleichheit, die Einschließung ist im Allgemeinen* nicht *optimal)*

Für einen Beweis siehe z.B. Alefeld/Herzberger [9]. Die Ungleichheit bei Multiplikation und Division kommt dadurch zustande, dass das Ergebnis dieser Operationen, anders als bei den entsprechenden Operationen mit reellen Intervallen, im Allgemeinen kein komplexes Intervall ist, wie folgendes Beispiel zeigt.

Beispiel 2.1. *Gegeben seien* $\boldsymbol{a} \in \mathbb{IC}_{\square}$ *sowie* $b = \cos(\alpha) + i\sin(\alpha) \in \mathbb{C}$. *Durch die Multiplikation mit* b *wird also das Rechteck* $\boldsymbol{a}$ *in der komplexen Ebene auf ein entsprechendes Rechteck abgebildet, welches um den Winkel* α *gedreht ist. Folgende Abbildung zeigt* $\boldsymbol{a}$ *sowie die Menge* $M = \{a \cdot b \mid a \in \boldsymbol{a}\}$ *und das Ergebnis der Multiplikation* $\boldsymbol{a} \cdot_{\square} b$.

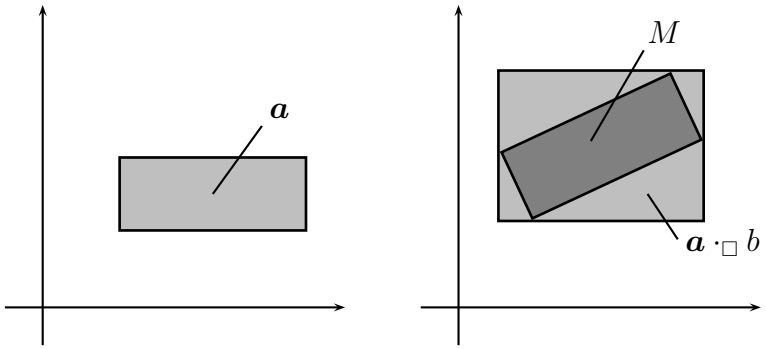


Abbildung 2.1.: Auswirkungen des Wrapping-Effektes bei der Multiplikation von komplexen Rechteckintervallen

Offensichtlich gilt $\boldsymbol{a} \cdot_{\square} b = \square(M)$, *allerdings muss die tatsächliche Ergebnismenge durch ein achsenparalleles Rechteck eingeschlossen werden, wodurch es zu einer Überschätzung kommt.*

In diesem Fall spricht man vom sogenannten *Wrapping-Effekt*, der erstmals von Moore [99] beschrieben wurde. Dass die Divisionsoperation von komplexen Rechteckintervallen gemäß Satz 2.3 im Allgemeinen keine optimale Einschließung berechnet, lässt sich bereits erkennen, wenn man den Imaginärteil gleich Null setzt, also die Operation auf eigentlich reelle Intervalle anwendet, wie folgendes Beispiel (aus [53]) zeigt.

Beispiel 2.2. *Es seien* $\boldsymbol{a} = [-1,1] + i[0,0]$ *und* $\boldsymbol{b} = [1,2] + i[0,0]$. *Dann ist*

$$\begin{aligned} \boldsymbol{a}/_{\square}\boldsymbol{b} &= \tfrac{[-1,1][1,2]+[0,0]}{[1,2]^2+[0,0]^2} + i\tfrac{[0,0]-[0,0]}{[1,2]^2+[0,0]^2} \\ &= \tfrac{[-2,2]}{[1,4]} + i[0,0] \\ &= [-2,2]. \end{aligned}$$

Eine reelle Auswertung entsprechend Definition 2.6 liefert aber als Intervallhülle

$$\begin{aligned} [-1,1]/[1,2] &= [\tfrac{-1}{1}, \tfrac{1}{1}] \\ &= [-1,1]. \end{aligned}$$

Die in Definition 2.8 eingeführten Größen werden für komplexe Rechteckintervalle jeweils getrennt für Real- und Imaginärteil angewendet. Ausgenommen sind Betrag und Betragsminimum, die für ein Rechteckintervall $\boldsymbol{a} = \boldsymbol{a}_1 + i \cdot \boldsymbol{a}_2$ als Ober- bzw. Unterschranke von $\{\sqrt{a_1^2 + a_2^2} \mid a_1 \in \boldsymbol{a}_1, a_2 \in \boldsymbol{a}_2\}$ definiert sind.

Die Darstellung komplexer Intervalle als Kreisscheiben in der komplexen Ebene und eine entsprechende Kreisscheibenarithmetik werden in folgender Definition eingeführt.

Definition 2.11. *Seien $m_a \in \mathbb{C}$, $r_a \in \mathbb{R}$, $r_a \geq 0$. Ein komplexes Kreisscheibenintervall $\boldsymbol{a} \in \mathbb{IC}_\circ$ ist dann definiert als*

$$\boldsymbol{a} =< m_a, r_a >= \{a \in \mathbb{C} \mid |a - m_a| \leq r_a\}.$$

Die Grundoperationen $\circ_\circ, \circ = \{+, -, \cdot, /\}$ für Kreisscheibenintervalle $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IC}_\circ$ werden folgendermaßen definiert:

$$\begin{array}{rcll} \boldsymbol{a} & +_\circ & \boldsymbol{b} & := & < m_a + m_b, r_a + r_b > \\ \boldsymbol{a} & -_\circ & \boldsymbol{b} & := & < m_a - m_b, r_a + r_b > \\ \boldsymbol{a} & \cdot_\circ & \boldsymbol{b} & := & < m_a \cdot m_b, |m_a| r_b + |m_b| r_a + r_a r_b > \\ 1 & /_\circ & \boldsymbol{b} & := & < \frac{\overline{m_b}}{m_b \overline{m_b} - r_b^2}, \frac{r_b}{m_b \overline{m_b} - r_b^2} >, 0 \notin \boldsymbol{b} \\ \boldsymbol{a} & /_\circ & \boldsymbol{b} & := & \boldsymbol{a} \cdot_\circ (1/ \circ \boldsymbol{b}), 0 \notin \boldsymbol{b} \end{array}$$

Die Notation $\overline{b}$ steht dabei für b konjugiert komplex.

Auch hier liefern die Grundoperationen verlässliche Einschließungen:

Satz 2.4. *Seien $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{IC}_\circ$. Dann gelten:*

1. $\boldsymbol{a} \pm_\circ \boldsymbol{b} = \{a \pm b \mid a \in \boldsymbol{a}, b \in \boldsymbol{b}\}$
2. $\boldsymbol{a} \cdot_\circ \boldsymbol{b} \supseteq \{a \cdot b \mid a \in \boldsymbol{a}, b \in \boldsymbol{b}\}$
3. $1/_\circ \boldsymbol{b} = \{1/b \mid b \in \boldsymbol{b}\}$
4. $\boldsymbol{a}/_\circ \boldsymbol{b} \supseteq \{a/b \mid a \in \boldsymbol{a}, b \in \boldsymbol{b}\}$

Bei 2. und 4. gilt im Allgemeinen Ungleichheit.

Ein entsprechender Beweis findet sich bei Rump [128]. Wie bei der Mittelpunkt-Radius-Darstellung reeller Intervalle gilt auch für die komplexe Multiplikation und Division, dass die ideale Einschließung um einen Faktor von maximal 1.5 überschätzt wird (siehe [128]). In dieser Arbeit wird im Allgemeinen, sofern nichts anderes angegeben wird, im Komplexen mit den auch in C-XSC üblichen (siehe Abschnitt 2.3) Rechteckintervallen gearbeitet.

Intervallvektoren und -matrizen sind Vektoren und Matrizen, deren Elemente Intervalle sind. Die Menge aller Intervallvektoren der Dimension n wird mit $\mathbb{IR}^n$ (im Reellen) bzw. $\mathbb{IC}^n$ (im Komplexen) bezeichnet, die Menge aller $m \times n$ Intervallmatrizen mit $\mathbb{IR}^{m \times n}$ (im Reellen) bzw. $\mathbb{IC}^{m \times n}$ (im Komplexen). Intervallvektoren werden mit fett gedruckten Kleinbuchstaben, z.B. $\boldsymbol{x}$, notiert, Intervallmatrizen mit fett gedruckten Großbuchstaben, z.B. $\boldsymbol{A}$.

Definition 2.12. *Intervallvektoren werden definiert als*

$$\boldsymbol{x} := (\boldsymbol{x}_i)_{i=1,\ldots,n} := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)^T \text{ mit } \boldsymbol{x} \in \mathbb{IR}^n \text{ bzw. } \boldsymbol{x} \in \mathbb{IC}^n.$$

Dabei wird das i-te Element eines Vektors $\boldsymbol{x}$ mit $\boldsymbol{x}_i$ bezeichnet. Intervallmatrizen werden definiert als

$$\boldsymbol{A} := (\boldsymbol{a}_{ij})_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} := \begin{pmatrix} \boldsymbol{a}_{11} & \ldots & \boldsymbol{a}_{1n} \\ \vdots & & \vdots \\ \boldsymbol{a}_{m1} & \ldots & \boldsymbol{a}_{mn} \end{pmatrix} \text{ mit } \boldsymbol{A} \in \mathbb{IR}^{m \times n} \text{ bzw. } \boldsymbol{A} \in \mathbb{IC}^{m \times n}.$$

Das Element an Position (i, j) einer Matrix $\boldsymbol{A}$ wird dabei mit $\boldsymbol{a}_{ij}$ bezeichnet.

Alle Matrix- und Vektoroperationen werden auf die übliche Art auf die bisher in diesem Abschnitt eingeführten Intervalloperationen zurückgeführt.

Die in den Definitionen 2.5 und 2.8 eingeführten Operatoren und Funktionen sind ebenso wie die Relationen $=, \subset, \subseteq, \supset, \supseteq$ für Intervallvektoren und -matrizen elementweise zu verstehen. Einige wichtige Normen lassen sich für $\boldsymbol{x} \in \mathbb{IK}^n, \boldsymbol{A} \in \mathbb{IK}^{n \times n}, \mathbb{IK} \in \{\mathbb{IR}, \mathbb{IC}\}$ über

$$\begin{aligned} ||\boldsymbol{x}||_\infty &:= \max_{1 \le i \le n} |\boldsymbol{x}_i| \\ ||\boldsymbol{A}||_\infty &:= \max_{1 \le i \le n} \sum_{j=1}^{m} |\boldsymbol{a}_{ij}| \\ ||\boldsymbol{x}||_1 &:= \sum_{i=1}^{n} |\boldsymbol{x}_i| \\ ||\boldsymbol{A}||_1 &:= \max_{1 \le j \le n} \sum_{i=1}^{m} |\boldsymbol{a}_{ij}| \\ ||\boldsymbol{x}||_2 &:= \sqrt{\sum_{i=1}^{n} |\boldsymbol{x}_i|^2} \\ ||\boldsymbol{A}||_2 &:= \max_{A \in \boldsymbol{A}} \sigma_{max}(A) \end{aligned}$$

auf Intervallvektoren und -matrizen erweitern ($\sigma_{max}(A)$ bezeichnet den größten Singulärwert einer Matrix A). Recht häufig benötigt wird die Ostrowskische Vergleichsmatrix.

Definition 2.13. *Es sei $\boldsymbol{A} \in \mathbb{IC}^{n \times n}$. Dann heißt die Matrix $\langle \boldsymbol{A} \rangle := (o_{ij})$, $\langle \boldsymbol{A} \rangle \in \mathbb{R}^{n \times n}$ mit*

$$o_{ij} := \begin{cases} \langle \boldsymbol{a}_{ij} \rangle & \text{für } i = j \\ -|\boldsymbol{a}_{ij}| & \text{für } i \neq j \end{cases}$$

Ostrowskische Vergleichsmatrix.

Beim Umgang mit Intervallen stößt man häufig auf das sogenannte *Abhängigkeitsproblem*, welches dann zum Tragen kommt, wenn das gleiche Intervall in einem Ausdruck mehrfach auftaucht. Dazu wird folgendes Beispiel betrachtet:

Beispiel 2.3. *Gegeben seien die Funktionen*

$$f_1(x) := \tfrac{1}{1+x} + \tfrac{1}{1-x}, \quad x \in \mathbb{R} \backslash \{-1, 1\},$$

und

$$f_2(x) := \tfrac{2}{1-x^2}, \quad x \in \mathbb{R}\backslash\{-1, 1\}.$$

Offensichtlich stimmt f_1 mit f_2 überein. Eine intervallmäßige Auswertung von f_1 für das Intervall $\boldsymbol{x} = [2, 3]$ liefert

$$\begin{aligned} f_1(\boldsymbol{x}) &= \tfrac{1}{1+[2,3]} + \tfrac{1}{1-[2,3]} \\ &= \tfrac{1}{[3,4]} + \tfrac{1}{[-2,-1]} \\ &= [\tfrac{1}{4}, \tfrac{1}{3}] + [-1, -\tfrac{1}{2}] \\ &= [-\tfrac{3}{4}, -\tfrac{1}{6}]. \end{aligned}$$

Die entsprechende intervallmäßige Auswertung von f_2 hingegen ergibt

$$\begin{aligned} f_2(\boldsymbol{x}) &= \tfrac{2}{1-([2,3])^2} \\ &= \tfrac{2}{1-[4,9])} \\ &= \tfrac{2}{[-8,-3])} \\ &= [-\tfrac{2}{3}, -\tfrac{1}{4}]. \end{aligned}$$

Es ist also $\operatorname{diam}(f_1(\boldsymbol{x})) = \frac{7}{12}$ *und* $\operatorname{diam}(f_2(\boldsymbol{x})) = \frac{5}{12}$.

Die Ursache für die Überschätzung ist, dass bei mehrfachem Auftreten einer Intervallvariablen in einem Ausdruck die intervallmäßige Auswertung nicht berücksichtigt, dass es sich immer um die gleiche Variable handelt. Das Ergebnis wird also so berechnet, als wären die einzelnen Vorkommen der Intervallvariablen voneinander unabhängig. Eine detailliertere Beschreibung des Problems findet sich bei Moore [99].

Eng verwandt mit dem Abhängigkeitsproblem ist der schon erwähnte Wrapping-Effekt, welcher allgemein bei Intervallvektor-Iterationen (die komplexe Multiplikation und Division sind im Prinzip ebenfalls solche Iterationen) auftreten kann. Intervallvektoren beschreiben immer einen achsenparallelen Quader (im Englischen spricht man daher bei einem Intervallvektor oft von einer *Box*). Wird dieser dann in einem Iterationsschritt auf eine Menge abgebildet, welche nicht einem achsenparallelen Quader entspricht, so muss diese durch den Ergebnisvektor, also wieder einen achsenparallelen Quader, eingeschlossen werden, wodurch es zu einer Überschätzung kommt.

Auf der Maschine werden Intervalle naturgemäß mit Hilfe von Gleitkommazahlen dargestellt:

Definition 2.14. *Ein Intervall $\boldsymbol{a} = [\underline{a}, \overline{a}]$ mit $\underline{a}, \overline{a} \in \mathbb{F}$ heißt* Maschinenintervall. *Die Menge aller Maschinenintervalle wird mit $\mathbb{IF}$ bezeichnet.*

Analog können Maschinenintervalle im Komplexen definiert werden. Bei der Implementierung auf dem Rechner muss darauf geachtet werden, dass trotz der auftretenden Rundungsfehler korrekte Einschließungen berechnet werden. In der Regel geschieht dies durch ein Umschalten des Rundungsmodus des Prozessors. Wie in Abschnitt 2.5.3 zu sehen sein wird, sind dabei einige wichtige Punkte zu beachten, um die Korrektheit und auch die Laufzeiteffizienz einer entsprechenden Implementierung zu gewährleisten.

Die Eingangsdaten für numerische Software liegen oftmals im Dezimalformat vor. Beim Einlesen dieser Daten und Umwandeln in eine binäre Gleitkommazahl kommt es dabei im Allgemeinen zu einem Konversionsfehler, da die entsprechende Dezimalzahl auf der Maschine nicht darstellbar ist.

Beispiel 2.4. *Die Zahl* 0.1 *soll in einem Gleitkommasystem mit Basis* $B = 2$ *dargestellt werden. Dies ist für kein entsprechendes Gleitkommasystem möglich, denn es gilt*

$$0.1 = \sum_{k=1}^{\infty} (2^{-4k} + 2^{-(4k+1)}).$$

Die Dezimalzahl 0.1 *hat also im Binärformat unendlich viele Nachkommastellen. Es ist daher unmöglich, sie in einem Gleitkommasystem mit Basis* $B = 2$ *mit endlicher Mantissenlänge exakt darzustellen.*

Beim Einlesen von Dezimalzahlen sollten diese daher zunächst als String behandelt und schließlich entsprechend gerundet in ein Maschinenintervall übernommen werden. Eine solche Funktionalität wird in der Regel von jeder Intervallbibliothek bereitgestellt (siehe z.B. Abschnitt 2.3 für die C-XSC Implementierung).

Die in diesem Abschnitt aufgeführten Grundlagen stellen eine Zusammenfassung der wichtigsten Punkte dar. Die Ausführungen basieren im Wesentlichen auf den klassischen Einführungswerken von Alefeld und Herzberger [9] und Moore [97]. Weitere Quellen waren Hammer et al. [53] sowie Krämer [81, 82]. Die Notation in diesem Abschnitt (und in der gesamten Arbeit) basiert auf einer Veröffentlichung von Kearfott et al. [71], welche versucht, eine einheitliche Notation für wissenschaftliche Arbeiten aus dem Gebiet der Verifikationsnumerik einzuführen. Zur Zeit wird an einem IEEE Standard für Intervallarithmetik auf der Maschine gearbeitet [4], welcher aber noch recht weit von der Fertigstellung entfernt ist und daher weder in dieser Arbeit noch in der C-XSC Bibliothek explizit Berücksichtigung findet.

2.1.3. Lineare Intervall-Gleichungssysteme

Ein lineares Intervallgleichungssystem hat allgemein die Form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \quad \boldsymbol{A} \in \mathbb{IK}^{m \times n}, \boldsymbol{b} \in \mathbb{IK}^n, \mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}.$$

(Anmerkung: In diesem Abschnitt werden nur reelle Gleichungssysteme betrachtet, im Komplexen gelten im Wesentlichen analoge Aussagen). Ein lineares Intervallgleichungssystem zu lösen bedeutet, eine Einschließung der Intervallhülle $\boldsymbol{x} \in \mathbb{IK}^m$ der im Folgenden definierten Lösungsmenge zu bestimmen.

Definition 2.15. *Es sei ein lineares Intervallgleichungssystem* $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ *vorgegeben. Die Lösungsmenge* Σ *dieses Systems ist definiert als*

$$\Sigma = \Sigma(\boldsymbol{A}, \boldsymbol{b}) := \{x \in \mathbb{R}^n \mid \exists A \in \boldsymbol{A}, \exists b \in \boldsymbol{b} \text{ mit } Ax = b\}.$$

Weiterhin lässt sich auch für Intervallmatrizen ein Regularitätsbegriff einführen [102].

Definition 2.16. *Eine Intervallmatrix $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ heißt regulär, falls jede Matrix $A \in \boldsymbol{A}$ regulär ist. Sie heißt stark regulär, falls* $\operatorname{mid}(\boldsymbol{A})^{-1}\boldsymbol{A}$ *regulär ist.*

Auf die starke Regularität wird im Verlauf dieser Arbeit noch näher eingegangen.

Ist die Systemmatrix $\boldsymbol{A}$ eines linearen Intervallgleichungssystems regulär, so lässt sich die Lösungsmenge offensichtlich auch über

$$\Sigma(\boldsymbol{A}, \boldsymbol{b}) = \{A^{-1}b \mid A \in \boldsymbol{A}, b \in \boldsymbol{b}\}$$

beschreiben. Im Folgenden soll $\Sigma(\boldsymbol{A}, \boldsymbol{b})$ zunächst näher charakterisiert werden.

Es beschreibe $\sigma_k, k = 1, \ldots, 2^n$ den k-ten Orthanten des $\mathbb{R}^n$. Dann lässt sich für jeden Orthanten eine Signaturmatrix $S^{(k)} \in \mathbb{R}^{n\times n}$ angeben, welche eine Diagonalmatrix mit

$$|s_{ii}^{(k)}| = 1, 1 \leq i \leq n$$

ist. Jede Matrix $S^{(k)}$ unterscheidet sich durch die Vorzeichenbelegung der Diagonalelemente und beschreibt über

$$\sigma_k := \{x \in \mathbb{R}^n \mid S^{(k)}x \geq 0\}$$

den k-ten Orthanten im $\mathbb{R}^n$. Mit Hilfe von $S^{(k)}$ lassen sich nun für jeden Orthanten Matrizen $C^{(k)} \in \mathbb{R}^{n\times n}$ und $D^{(k)} \in \mathbb{R}^{n\times n}$ festlegen als

$$c_{ij}^{(k)} := \begin{cases} \underline{a}_{ij} & \text{falls } s_j^{(k)} = 1 \\ \overline{a}_{ij} & \text{falls } s_j^{(k)} = -1 \end{cases} \qquad d_{ij}^{(k)} := \begin{cases} \overline{a}_{ij} & \text{falls } s_j^{(k)} = 1 \\ \underline{a}_{ij} & \text{falls } s_j^{(k)} = -1 \end{cases} .$$

Für jeden Orthanten k werden dann jeweils n Halbräume $\overline{H}^{(k)}$ und $\underline{H}^{(k)}$ definiert als

$$\begin{aligned} \underline{H}_i^{(k)} &:= \left\{ x \in \mathbb{R}^n \mid \sum_{j=1}^{n} c_{ij}^{(k)} x_j \leq \overline{b}_i \right\} \\ \overline{H}_i^{(k)} &:= \left\{ x \in \mathbb{R}^n \mid \sum_{j=1}^{n} d_{ij}^{(k)} x_j \geq \underline{b}_i \right\}. \end{aligned}$$

Mit Hilfe von $\overline{H}^{(k)}$ und $\underline{H}^{(k)}$ lässt sich nun die exakte Lösungsmenge Σ eines linearen Intervallgleichungssystems folgendermaßen charakterisieren [16].

Satz 2.5. *Gegeben sei ein lineares Intervallgleichungssystem $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ mit einer regulären Systemmatrix $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ und $\boldsymbol{b} \in \mathbb{IR}^n$. Dann gilt für den Schnitt der exakten Lösungsmenge Σ mit dem k-ten Orthanten σ_k:*

$$\Sigma \cap \sigma_k = \bigcap_{i=1}^{n} \left(\underline{H}_i^{(k)} \cap \overline{H}_i^{(k)} \right) \cap \sigma_k.$$

Ist die Menge $\Sigma \cap \sigma_k$ nicht leer, so ist sie konvex, kompakt, zusammenhängend und ein Polytop. Weiterhin gilt für die Lösungsmenge:

$$\Sigma = \bigcup_{k=1}^{2^n} \left(\bigcap_{i=1}^{n} (\underline{H}_i^{(k)} \cap \overline{H}_i^{(k)}) \cap \sigma_k \right).$$

Die Lösungsmenge Σ ist kompakt und zusammenhängend, aber im Allgemeinen nicht konvex.

Ein Beweis dieses Satzes findet sich bei Alefeld/Mayer [10]. Zur Verdeutlichung folgt ein klassisches Beispiel von Barth und Nuding [15].

Beispiel 2.5. *Gegeben ist das lineare Intervallgleichungssystem $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ mit der regulären Systemmatrix*

$$\boldsymbol{A} := \begin{pmatrix} [2,4] & [-2,0] \\ [-1,0] & [2,4] \end{pmatrix}$$

sowie der rechten Seite

$$\boldsymbol{b} := \begin{pmatrix} [1,2] \\ [-2,2] \end{pmatrix}.$$

Die exakte Lösungsmenge dieses Systems wird in Abbildung 2.2 dargestellt.

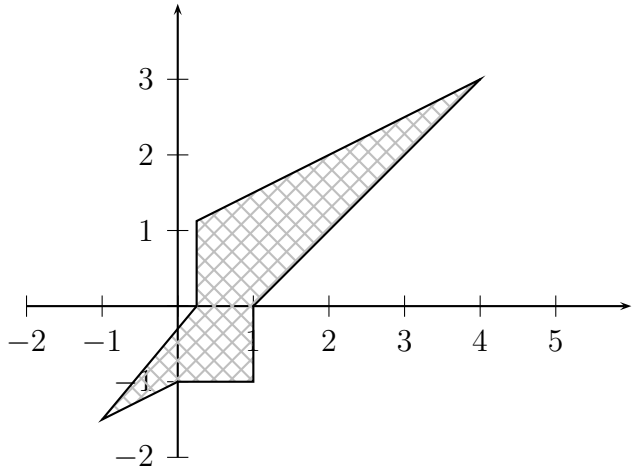


Abbildung 2.2.: Exakte Lösungsmenge des Beispielsystems

Die Lösung eines linearen Intervallgleichungssystems zu finden bedeutet, eine (möglichst enge) Intervalleinschließung der Lösungsmenge $\Sigma(\boldsymbol{A}, \boldsymbol{b})$ zu bestimmen. Die optimale Lösung eines linearen Intervallgleichungssystems ist demnach gerade die Intervallhülle $\square\Sigma(\boldsymbol{A}, \boldsymbol{b})$ der Lösungsmenge. In verschiedenen Arbeiten, insbesondere von Rohn und Kreinovich, wurde gezeigt, dass das Finden einer solchen optimalen Lösung ein NP-schwieriges Problem ist [57, 86, 119, 120]. Entsprechende Algorithmen zur Bestimmung einer optimalen Lösung haben daher im Allgemeinen exponentielle Laufzeit, wie z.B. der Sign-Accord-Algorithmus von Rohn [121,122]. Jansson beschreibt einen Algorithmus, der

unter bestimmten Umständen nur (hohen) polynomialen Aufwand, im worst-case aber weiterhin exponentielle Laufzeit benötigt [67].

Zum verifizierten Lösen linearer Intervallgleichungssysteme sind also spezielle Algorithmen von Bedeutung, welche eine möglichst enge Einschließung der Intervallhülle der exakten Lösungsmenge in polynomialer Laufzeit berechnen können. Entsprechende Algorithmen werden in den Kapiteln 4 und 5 besprochen. Dass hierbei das naive Übernehmen von bewährten Algorithmen für Punktsysteme (durch ersetzen der Operationen durch entsprechende Intervalloperationen) zumeist keine praktikable Lösung ist (diese Aussage gilt allgemein für die allermeisten numerischen Algorithmen), zeigt das Beispiel des Intervall-Gauß-Algorithmus (IGA). Dieser entspricht dem bekannten Gaußalgorithmus, in dem alle Operationen durch die entsprechenden Intervalloperationen ersetzt wurden.

Der IGA ist durchführbar, sofern keine Division durch ein Intervall welches Null enthält auftritt. In der Praxis liefert der IGA in der Regel, außer für sehr kleine Dimensionen, entweder Ergebnisse mit sehr starker Überschätzung, welche daher nahezu wertlos sind, oder - und das ist abgesehen von sehr kleinen Dimensionen der Normalfall - er bricht vorzeitig aufgrund einer Division durch Null ab. Rump [132] demonstriert, dass mit Zufallsmatrizen ab einer Dimension von etwa $n = 70$ der IGA grundsätzlich versagt. Es kann allerdings gezeigt werden, dass der IGA für bestimmte Matrixtypen (insbesondere M-Matrizen und streng diagonal dominante Matrizen) immer erfolgreich ist. Unter bestimmten Voraussetzungen liefert er dann sogar die exakte Intervallhülle. Für weitere Details und eine Erläuterung, warum der IGA meist nicht zum Erfolg führt, siehe Rump [132].

2.1.4. Grundprinzipien der Verifikationsnumerik

In der Verifikationsnumerik ist es das Ziel, eine verlässliche und verifizierte Lösung eines Problems zu finden. Es soll also, mit anderen Worten, mit Hilfe des Computers (fehlerfrei arbeitende Hardware vorausgesetzt) bewiesen werden, dass die Lösung des betrachteten Problems ungeachtet der möglichen auftretenden Fehler (insbesondere Rundungsfehler) in jedem Fall durch die berechnete Lösungsmenge bzw. die Intervallhülle der Lösungsmenge umfasst wird.

Dabei ist die Intervallarithmetik ein wichtiges Werkzeug. Allerdings reicht es, wie im letzten Abschnitt am Beispiel des Intervall-Gauß-Verfahrens gesehen, in aller Regel nicht aus, in einem bekannten Algorithmus einfach alle Operationen durch entsprechende Intervalloperationen zu ersetzen.

Eine weitere wichtige Grundlage für die Verifikationsnumerik sind speziell auf das jeweilige Problem angepasste verifizierende Algorithmen. Viele verifizierende Algorithmen berechnen zunächst (zumeist mit den jeweils üblichen numerischen Methoden) eine Näherungslösung. Das Ziel ist letztlich aber, mathematische Sätze herzuleiten, welche (oftmals auf Grundlage einer Näherungslösung) eine Aussage über eine verifizierte Einschließung der tatsächlichen Lösung treffen, und deren Voraussetzungen auf dem Rechner geprüft werden können. Rump nennt dies das *Design Principle of Verification Methods* [132].

Zu beachten ist, dass ein verifizierender Algorithmus verlässlich eine Einschließung der Lösung eines Problems berechnet, es also bei einem erfolgreichen Ablauf des Algo-

rithmus mathematisch bewiesen ist, dass die berechnete Einschließung die tatsächliche Lösung umfasst, dass aber umgekehrt bei einem nicht erfolgreichen Ablauf des Algorithmus in der Regel keine Aussage über die Lösbarkeit des Problems getroffen wird (also kein Beweis erbracht ist, dass das jeweilige Problem nicht lösbar ist). Ein Versagen des in Abschnitt 4.1.1 besprochene Algorithmus zur verifizierten Lösung dicht besetzter Gleichungssysteme bedeutet also z.B. nicht zwingend, dass die Systemmatrix des betrachteten Gleichungssystems singulär ist (sie kann es aber sein).

Viele verifizierende Algorithmen benutzen Fixpunkt-Argumentationen zur Ergebnisverifikation. Von besonderem Interesse, gerade für die Thematik dieser Arbeit mit endlichdimensionalen Fragestellungen, ist dabei der Brouwersche Fixpunktsatz:

Satz 2.6. *Sei $M \in \mathbb{R}^n$ eine kompakte, konvexe und nicht leere Menge. Weiterhin sei $f : M \to \mathbb{R}^n$ eine auf M stetige Funktion. Gilt dann $f(M) \subseteq M$, so besitzt f wenigstens einen Fixpunkt $x^* \in M$ (es gilt also $f(x^*) = x^*$).*

Für einen Beweis siehe z.B. Neumaier [102]. Ein Intervallvektor $\boldsymbol{x} \in \mathbb{IR}^n$ erfüllt offensichtlich die Voraussetzungen von Satz 2.6. Mit einer stetigen Funktion $\mathrm{f} : \mathbb{IR}^n \mapsto \mathbb{IR}^n$ gilt also, falls ein Intervallvektor $\boldsymbol{x}$ gefunden werden kann, für den $\mathrm{f}(\boldsymbol{x}) \subseteq \boldsymbol{x}$, dass $\boldsymbol{x}$ wenigstens einen Fixpunkt von f einschließt.

Mit Hilfe dieses Werkzeugs kann also z.B. ein verifizierender Algorithmus für ein bestimmtes Problem entworfen werden, indem das ursprüngliche Problem in Fixpunktform $x = \mathrm{f}(x)$ überführt wird. Gilt dann z.B. für eine Näherungslösung $\boldsymbol{x}^{(0)}$ und die Iterationsvorschrift

$$\boldsymbol{x}^{(k+1)} = \mathrm{f}(\boldsymbol{x}^{(k)}), k = 0, 1, 2, \ldots$$

dass $\boldsymbol{x}^{(k+1)} \subseteq \boldsymbol{x}^{(k)}$ für ein beliebiges $k \in \{0, 1, 2, \ldots\}$, so ist mathematisch bewiesen, dass $\boldsymbol{x}$ wenigstens eine Lösung des ursprünglichen Problems einschließt. Bei manchen Problemen gibt es auch die Möglichkeit nachzuweisen, dass $\boldsymbol{x}$ sogar eine (eventuell lokal) eindeutige Lösung einschließt.

Für einen allgemeinen Überblick über das Gebiet der Verifikationsnumerik siehe insbesondere Rump [132].

2.2. Parallele Programmierung

Die Parallelisierung von Programmen hat in der heutigen Zeit eine enorme Bedeutung erlangt. Waren noch vor zehn Jahren Systeme mit mehreren Prozessoren oder Prozessorkernen allenfalls im Serverbereich und bei professionellen Anwendern anzutreffen, so ist es heute nahezu unmöglich, einen Computer, egal welchen Preissegments, zu kaufen, der nicht zumindest mit einem Zweikern-Prozessor ausgestattet ist. Selbst in mobilen Geräten wie Smartphones werden nun vermehrt Mehrkernprozessoren verbaut. Weiterhin wird das enorme Potential moderner programmierbarer Grafikchips, sogenannter GPUs (*Graphics Processing Unit*), welche eine immense Anzahl parallel arbeitender (aber im Vergleich zu einer CPU sehr einfacher) Rechenkerne verwenden, durch entsprechende Programmier-Schnittstellen wie CUDA [133] und OpenCL [73] auch für allgemeine Aufgaben, gerade auch numerischer Natur, nutzbar gemacht.

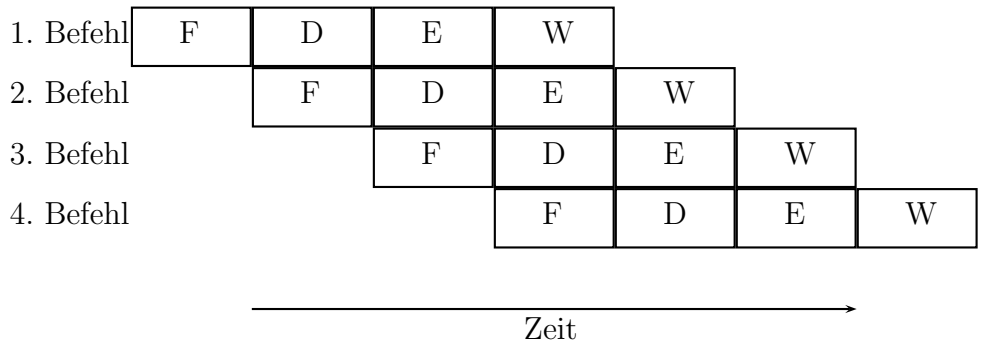


Abbildung 2.3.: Schematische Darstellung des Pipelinings

In diesem Abschnitt sollen zunächst einige grundlegende Begriffe aus der parallelen Programmierung eingeführt werden. Danach folgen genauere Erläuterungen insbesondere zu den in dieser Arbeit genutzten Techniken. Dabei wird zwischen Shared Memory Parallelisierung (Abschnitt 2.2.1) und Distributed Memory Parallelisierung (Abschnitt 2.2.2) unterschieden.

Zunächst sollen aber einige wichtige Begriffe eingeführt werden. In dieser Arbeit wird unter Parallelisierung im Allgemeinen tatsächliche Nebenläufigkeit verstanden, also der parallele Ablauf verschiedener Programmteile mit Hilfe von mehreren Threads oder Prozessen. In modernen Prozessoren findet man an verschiedenen Stellen auch andere parallele Abläufe. Zum einen ist hier insbesondere das sogenannte Pipelining, bei dem die einzelnen Teil der verschiedenen Maschinenbefehle für mehrere aufeinander folgende Befehle parallel ausgeführt werden, zu nennen. Abbildung 2.3 zeigt eine schematische Darstellung.

In der Abbildung wird von einer vierstufigen Pipeline ausgegangen, wie sie in einer klassischen RISC-Architektur (*Reduced Instruction Set Computer*) verwendet wird. Jeder Befehl durchläuft dabei folgende Stufen, welche in der Abbildung durch die entsprechenden Buchstaben repräsentiert werden:

- Fetch (**F**): Einlesen des nächsten Befehls.
- Decode (**D**): Dekodieren des eingelesenen Befehls.
- Execute (**E**): Ausführung des Befehls.
- Write (**W**): Schreiben des Ergebnisses in das entsprechende Register.

Werden mehrere Befehle hintereinander ausgeführt, ohne dass die Pipeline geleert werden muss (ein sogenannter Flush, z.B. aufgrund einer Verzweigung im Programm), so können diese Befehle also quasi gleichzeitig abgearbeitet werden, da jeweils für jeden Befehl ein anderer Verarbeitungsschritt im gleichen Takt durchgeführt wird.

Ein weiterer paralleler Ablauf in der CPU wird durch die sogenannte Vektorisierung erreicht. Auf allen modernen x86-Prozessoren stehen z.B. die sogenannten SSE-Befehlseinheiten (*Streaming SIMD Extensions*) zur Verfügung, welche den gleichen Be-

fehl in einem Prozessortakt auf mehrere Daten anwenden können, die in speziellen Registern gespeichert werden (die SSE-Register sind im Allgemeinen 128 Bit breit und können somit gleichzeitig zwei *double*-Werte aufnehmen). So können z.B. gleichzeitig zwei Additionen von zwei verschiedenen *double*-Wertepaaren durchgeführt werden. Moderne Compiler stellen als Optimierungsfeature die sogenannte Autovektorisierung zur Verfügung, welche automatisch versucht, Schleifen mit Hilfe dieser Befehle zu vektorisieren und so die Abarbeitung zu beschleunigen. Weiterhin stellen die meisten Compiler auch intrinsische Funktionen zum direkten Aufruf der entsprechenden Befehle zur Verfügung. Außerdem können sie natürlich auch direkt über Assembler-Anweisungen ausgeführt werden.

Die angesprochenen Techniken sind in dieser Arbeit zwar auch von Bedeutung (das Pipelining spielt z.B. eine wichtige Rolle beim Umschalten des Rundungsmodus, siehe Abschnitt 2.5.3, während die SSE-Befehle bei der Implementierung der fehlerfreien Transformationen in Abschnitt 3.1.3 genutzt werden), werden aber hier nicht dem Begriff der Parallelisierung zugeordnet.

Parallele Computerarchitekturen werden nach Flynn [43] in folgende Kategorien eingeordnet:

- **SISD**: Single Instruction, Single Data. Es wird immer nur ein Befehl mit einem Datensatz ausgeführt. Es handelt sich hier also um klassische, seriell arbeitende Computer.
- **SIMD**: Single Instruction, Multiple Data. Ein Befehl wird gleichzeitig auf mehrere Datensätze angewendet. Diesem Modell entspricht z.B. die oben angesprochene Vektorisierung ebenso wie GPUs.
- **MISD**: Multiple Instruction, Single Data. Mehrere Befehle werden gleichzeitig auf einen Datensatz angewendet. Dieses Modell wird sehr selten verwendet.
- **MIMD**: Multiple Instruction, Multiple Data. Mehrere unabhängige Befehle werden auf unabhängige Datensätze angewendet. Dies ist das am weitesten verbreitete Modell.

In einer Aktualisierung von Flynns Einteilung wird die MIMD-Kategorie u.a. nach Darema [32] heute zumeist noch weiter unterteilt:

- **SPMD**: Single Program, Multiple Data. Das gleiche Programm wird von verschiedenen unabhängigen Prozessoren bzw. Prozessorkernen mit verschiedenen Datensätzen durchgeführt. Dies entspricht den meisten heutigen Parallelrechnern.
- **MPMD**: Multiple Program, Multiple Data. Verschiedene Programme werden von unabhängigen Prozessoren bzw. Prozessorkernen mit unterschiedlichen Datensätzen ausgeführt. Diesem Modell entspricht z.B. die Cell-Architektur.

In diesem Abschnitt werden die Parallelisierungsmethoden nach der Art des Speicherzugriffes unterteilt (geteilter Speicher (shared memory) oder verteilter Speicher (distributed memory)). Eine Einteilung in die obigen Kategorien wird in den Erläuterungen

der folgenden Abschnitte jeweils mit angegeben oder ergibt sich aus den jeweiligen Beschreibungen.

Bei der parallelen Programmierung sind einige Grundbegriffe sowie Messgrößen zur Beurteilung einer durchgeführten Parallelisierung von Bedeutung:

Definition 2.17. *Es bezeichne* $\mathrm{t}(p)$ *die benötigte Zeit zur Ausführung eines Programms mit* p *Prozessoren oder Threads. Dann bezeichnet*

$$\mathrm{S}(p) := \frac{\mathrm{t}(1)}{\mathrm{t}(p)}$$

den Speed-Up *des Programms. Üblicherweise verwendet man für die serielle Ausführung eine serielle Version des Programms (also nicht die parallelisierte Version mit nur einem Prozessor bzw. Thread). Man spricht dann auch vom* eigentlichen Speed-Up*. Weiterhin bezeichne*

$$\mathrm{E}(p) := \frac{\mathrm{S}(p)}{p}$$

die Effizienz *der Parallelisierung, welche im Idealfall nahe bei* 1 *liegen sollte (unter bestimmten Umständen kann die Effizienz in der Realität, z.B. durch Caching-Effekte, auch größer als* 1 *sein, man spricht dann von* superlinearem Speed-Up*). Wird die Effizienz mit dem eigentlichen Speed-Up berechnet, so spricht man entsprechend von der* eigentlichen Effizienz*. Möchte man deutlich machen, dass der Speed-Up auch von der Problemgröße* n *abhängt, so schreibt man* $\mathrm{S}(p, n)$*. Damit lässt sich der* Scaleup *einführen als*

$$\mathrm{Sc}(p) := \sup_{n \in \mathbb{N}} \mathrm{S}(p, n).$$

Man sagt, dass ein paralleles Programm skaliert*, falls*

$$\lim_{p \to \infty} \frac{\mathrm{Sc}(p)}{p} > 0$$

gilt.

Von Interesse ist bei der Parallelisierung auch, bis zu welchem Grad sich ein Programm überhaupt parallelisieren lässt. Eine entsprechende Aussage lässt sich mittels Amdahls Gesetz [11] treffen.

Satz 2.7. *Es sei* $P \in \mathbb{R}$ *mit* $0 \leq P \leq 1$ *der Anteil eines Programms, welcher sich parallelisieren lässt und entsprechend* $(1 - P)$ *der rein serielle Anteil. Dann gilt für alle* $p > 1$*:*

$$\mathrm{S}(p) \leq \frac{1}{(1 - P)}.$$

Der Beweis ist einfach zu führen: Egal wieviele Prozessoren genutzt werden, es muss immer gelten, dass $\mathrm{t}(p) \geq (1 - P)\mathrm{t}(1)$ und somit gilt $\frac{1}{(1-P)} \geq \frac{\mathrm{t}(1)}{\mathrm{t}(p)} = \mathrm{S}(p)$. □

Amdahls Gesetz gilt im Allgemeinen als pessimistisch, da Effekte wie z.B. der größere zur Verfügung stehende Cache-Speicher bei Mehrprozessor- und Mehrkernsystemen, welcher durch seine Geschwindigkeit je nach Problemgröße zu superlinearem Speed-Up führen kann, nicht berücksichtigt werden.

2.2.1. Shared Memory Parallelisierung

Bei der Shared Memory Parallelisierung werden Systeme betrachtet, in denen sich alle Prozessoren bzw. Kerne (wird im Folgenden von Prozessoren gesprochen, so sind damit der Einfachheit halber auch einzelne Prozessorkerne gemeint, sofern nichts anderes angegeben wird) einen gemeinsamen Speicher teilen. Dies betrifft insbesondere moderne Mehrkernprozessoren, aber auch die meisten Mehrprozessorsysteme und GPUs.

Der Vorteil bei dieser Art von parallelem System ist, dass die einzelnen Prozessoren direkten Zugriff auf alle notwendigen Daten haben und somit keine aufwändige Kommunikation zwischen den einzelnen Prozessoren wie bei Systemen mit verteiltem Speicher (siehe Abschnitt 2.2.2) nötig ist. Auf der anderen Seite ergibt sich dadurch der Nachteil, dass Zugriffe auf die gleichen Daten im Speicher synchronisiert werden müssen, um inkonsistente Daten zu vermeiden (z.B. falls ein Prozessor gerade die Einträge einer Matrix im Speicher überschreibt, während ein anderer diese ausliest).

Bei Mehrkern- bzw. Mehrprozessorsystemen kann eine solche Parallelisierung durch die Verwendung von mehreren Threads (*Multithreading*) oder mehreren Prozessen realisiert werden, wobei die Verwendung von Multithreading am Gebräuchlichsten ist. Threads stellen dabei nebenläufige Programmteile dar, die vom Betriebssystem auf die zur Verfügung stehenden Rechenkerne verteilt werden (*Scheduling*). Diese Verteilung hängt insbesondere von der aktuellen Systemlast ab. Der Ablauf nebenläufiger Threads ist daher nicht deterministisch, d.h. die Reihenfolge, in welcher die parallelen Programmteile tatsächlich abgearbeitet werden, ändert sich in der Regel bei jeder Ausführung. Dies muss in einem Programm, das Multithreading verwendet, entsprechend berücksichtigt werden.

Zur Implementierung von Multithreading-Programmen gibt es mehrere Möglichkeiten. Insbesondere stehen viele Bibliotheken (z.B. Posix-Threads, Qt) zur Verfügung, die das explizite Verwalten von Threads ermöglichen. Meistens wird dabei eine Funktion als Startpunkt an einen solchen Thread übergeben. Diese Programmierung „von Hand“ und die notwendige Verwaltung der Threads ist allerdings vergleichsweise aufwändig. Eine heute vielfach verwendete Alternative ist das auch in dieser Arbeit genutzte OpenMP (*Open Multi-Processing*) [109].

Bei OpenMP wird der Quellcode um Compiler-Pragmas (Steueranweisungen an den Compiler) ergänzt, welche einem OpenMP-fähigen Compiler anzeigen, wie der folgende Codeblock behandelt werden soll. Dabei werden vom Programmierer keine Threads explizit erzeugt, sondern abstraktere Anweisungen, z.B. zur Parallelisierung einer for-Schleife, angegeben. Dadurch ist es oftmals möglich, bestehenden seriellen Code einfach um entsprechende OpenMP-Anweisungen zu erweitern, ohne grundsätzliche Änderungen am Programmaufbau vornehmen zu müssen. In solchen Fällen kann das Programm meist auch mit einem nicht OpenMP-fähigen Compiler übersetzt werden, der Code wird dann seriell (aber weiterhin korrekt) ausgeführt.

OpenMP beinhaltet sogenannte Konstrukte, welche den folgenden Quellcode-Block charakterisieren, z.B. als parallele Region (`parallel`-Konstrukt) oder als Bereich, der nur von einem Thread abgearbeitet werden soll (`single`-Konstrukt). Diese Konstrukte können über sogenannte Direktiven genauer konfiguriert werden. Über die Direktiven wird insbesondere gesteuert, welche Variablen innerhalb des Konstrukts von allen

Threads geteilt werden (`shared`-Direktive) bzw. bei welchen Variablen jeder Thread mit einer lokalen Kopie arbeiten soll (`private`-Direktive). Der Quelltextauszug in Listing 2.1 zeigt, wie mittels des `parallel for` Konstruktes eine for-Schleife auf einfache Art parallelisiert werden kann und wie dabei der Zugriff auf Variablen über Direktiven gesteuert werden kann.

```
1  int n = 1000;
2  rvector x(n),y(n),z(n);
3
4  //...Vektoren fuellen....
5
6  #pragma omp parallel for private(i) shared(x,y,z)
7  for(int i=1 ; i<=n ; i++) {
8    z[i] = x[i] + y[i];
9  }
```

Listing 2.1: Beispiel für die Parallelisierung einer for-Schleife mittels OpenMP

In diesem Beispiel werden zwei Vektoren elementweise addiert und in einem dritten Vektor gespeichert. Durch das Hinzufügen einer einzigen Pragma-Anweisung parallelisiert der Compiler diese for-Schleife, indem er das Laufintervall der Laufvariablen `i` auf die zur Verfügung stehenden Threads aufteilt. Die `private`-Direktive gibt im Beispiellisting an, dass für die Laufvariable `i` von jedem Thread eine private Kopie benutzt werden soll (dies geschieht bei der Parallelisierung von for-Schleifen automatisch, die Direktive wird hier nur zur Erläuterung explizit aufgeführt) und dass die drei Vektorvariablen `x`, `y` und `z` jeweils von allen Threads geteilt werden (d.h. bei Benutzung von diesen drei Variablen wird von jedem Thread auf den selben Speicherbereich zugegriffen). Auch diese Direktive könnte in obigem Beispiel weggelassen werden, da `shared` die Voreinstellung für alle Variablen ist. Zu Bemerken ist noch, dass obiges Code-Fragment auch von Compilern ohne OpenMP Unterstützung kompilierbar ist. Das Compiler-Pragma würde dann ignoriert und einfacher serieller Quellcode erzeugt werden.

Weiterhin bietet OpenMP einige Funktionen, welche über den Header `<omp.h>` zur Verfügung stehen. Die Namen aller eingebauten OpenMP-Funktionen beginnen stets mit `omp`. Ein Beispiel zu deren Benutzung und wird in Listing 2.2 angegeben.

```
1  #include <stdio.h>
2  #ifdef _OPENMP
3  #include <omp.h>
4  #endif
5
6  int main() {
7    omp_set_num_threads(4);
8    #pragma omp parallel
9    {
10     int p = omp_get_thread_num();
11     printf("Ich bin Thread %d von %d\n", p, omp_get_num_threads());
12   }
13   return 0;
```

```
14 }
```

Listing 2.2: Vollständiges OpenMP-Beispielprogramm mit Benutzung von eingebauten OpenMP-Funktionen

In diesem Programm wird zunächst über das `parallel`-Konstrukt eine parallele Region erzeugt. Parallele Regionen werden von allen zur Verfügung stehenden Threads durchlaufen. Deren Anzahl wird mittels `omp_set_num_threads` (einer eingebauten OpenMP-Funktion) auf 4 gesetzt. Innerhalb der parallelen Region fragt jeder Thread zunächst über `omp_get_thread_num` seine eigene ID (diese beginnen bei 0) ab und gibt diese anschließend zusammen mit der Gesamtzahl aller Threads aus.

Bei der Ausgabe innerhalb von parallelen Regionen sollten in der Regel nicht, wie sonst in C++ üblich, Ausgabeströme verwendet werden, da bei diesen durch die Nebenläufigkeit die Ausgaben mehrerer Threads vermischt werden können. Stattdessen wird in der Regel auf die C-Standardfunktionen zur Ein- und Ausgabe wie `printf` zurückgegriffen.

Interessant an obigem Beispiel ist noch, dass von einem OpenMP-fähigen Compiler automatisch die Präprozessorvariable `_OPENMP` gesetzt wird. Mit deren Hilfe lässt sich also während der Kompilierung abfragen, ob OpenMP unterstützt wird. So können gegebenenfalls Programmbereiche, bei denen der normale serielle Code nicht einfach mit OpenMP angepasst werden kann, eine spezialisierte parallele Version benutzen. Außerdem sollte der OpenMP-Header wie in obigem Beispiel stets nur dann eingebunden werden, wenn diese Präprozessorvariable gesetzt ist.

OpenMP wird mittlerweile von nahezu allen bedeutenden C++ Compilern (insbesondere auch dem GNU Compiler, dem Intel Compiler und Microsofts Visual Studio Compiler) unterstützt und ist ein offener Standard, der ständig weiterentwickelt wird. Die vollständigen aktuellen Spezifikationen finden sich in der offiziellen Schnittstellenbeschreibung [109]. Obige Ausführungen sind natürlich nur ein sehr oberflächlicher Einstieg in die Möglichkeiten von OpenMP. Für weitere Details und eine detailliertere Einführung empfiehlt sich das Buch von Chapman et al. [28].

Zur Kategorie der Shared Memory Parallelisierung zählen auch die bereits angesprochenen GPUs. Diese stellen eine weitaus größere Zahl von Rechenkernen als gebräuchliche Mehrkernprozessoren zur Verfügung, welche allerdings dafür auch (im Vergleich zu einer vollwertigen CPU) nicht sehr komplex und nur für einfache Aufgaben geeignet sind. GPUs entsprechen dem SIMD Modell, d.h. eine Anweisung (z.B. eine Addition) kann gleichzeitig für eine Vielzahl von Datensätzen ausgeführt werden. Eine GPU mit 1024 Rechenkernen könnte so z.B. gleichzeitig 1024 Paare von Gleitkommazahlen addieren, was bei entsprechend geeigneten, von Natur aus gut parallelisierbaren Problemstellungen natürlich enorme Geschwindigkeitsgewinne verspricht.

Zur Nutzung dieser Fähigkeiten stehen insbesondere das proprietäre CUDA (*Compute Unified Device Architecture*) von Nvidia, welches auch nur mit GPUs dieser Firma genutzt werden kann, sowie OpenCL (*Open Computing Language*), ein offener Standard, welcher unter Zusammenarbeit mehrerer Firmen weiterentwickelt wird und mit GPUs aller Hersteller kompatibel ist. Beiden gemein ist, dass bei der Programmierung sogenannte Kernel benutzt werden. Dies sind kleine Programmfragmente, welche die parallel auszuführenden Anweisungen enthalten. Die Kernels werden jeweils in einer eigenen Ab-

wandlung der C-Sprache implementiert (CUDA C bzw. OpenCL C).

Ein großer Unterschied zwischen CUDA und OpenCL ist, dass OpenCL keine direkte Unterscheidung zwischen den Rechenkernen einer CPU und denen der GPU macht. Die einzelnen Kernels werden erst zur Laufzeit kompiliert und auf die auf dem jeweiligen System zur Verfügung stehende Hardware verteilt. OpenCL hat dadurch gegenüber CUDA also eine höhere Flexibilität, büßt allerdings durch die Kompilierung zur Laufzeit etwas an Geschwindigkeit ein. Ein Performance-Vergleich der beiden Ansätze findet sich z.B. bei Karimi et al. [68].

Der Einsatz von GPUs im Bereich der Verifikationsnumerik war lange Zeit problematisch. Die meisten GPUs unterstützten nur Berechnungen im `single`-Format oder erlitten enorme Geschwindigkeitseinbußen bei Verwendung des `double`-Formats. Auch wurde oftmals die für den Einsatz von Intervallen so wichtige Umschaltung des Rundungsmodus nicht unterstützt. Diese Probleme wurden bei neueren Grafikchips weitgehend behoben. Die GPUs, welche die Umschaltung des Rundungsmodus unterstützen, bieten dabei den Vorteil, dass der Rundungsmodus fest der jeweiligen Operation zugeordnet wird, also nicht global umgeschaltet werden muss (näheres zu den dadurch entstehenden Problemen in Abschnitt 2.5.3).

Im Rahmen dieser Arbeit werden die Möglichkeiten durch den Einsatz von GPUs in der Verifikationsnumerik noch nicht näher betrachtet. Für die Zukunft ist der Einsatz von GPUs auch in der Verifikationsnumerik und speziell in C-XSC aber sicherlich von großem Interesse.

2.2.2. Distributed Memory Parallelisierung

Die Distributed Memory Parallelisierung beschäftigt sich mit Systemen mit verteiltem Speicher. In der Regel handelt es sich dabei um in einem Netzwerk zusammengeschlossene Computer, in diesem Zusammenhang auch Knoten genannt. Jeder dieser Knoten ist ein autark arbeitendes System. Um also zusammen ein Problem bearbeiten zu können, ist es nötig, Nachrichten untereinander auszutauschen. Man spricht bei solchen Systemen auch von Clustern. Auch der Begriff Supercomputer bezieht sich üblicherweise auf solche Systeme. Die aktuell schnellsten dieser Supercomputer werden in der regelmäßig aktualisierten Top 500 Liste [8] aufgeführt.

Der größte Nachteil solcher Systeme ist, dass das Austauschen von Nachrichten in der Regel sehr zeitaufwändig ist. Zwar werden bei entsprechenden Systemen Hochleistungsnetzwerke, die speziell für diesen Zweck optimiert sind, verwendet, dennoch ist der Aufwand für das Verschicken einer einfachen Nachricht bereits um ein Vielfaches höher als eine einfache arithmetische Operation, also z.B. eine Addition. Allgemein wird dies grob über die Formel

$$t_c = (\alpha + \beta n) t_a$$

ausgedrückt, wobei t_c den Zeitaufwand für die betreffende Kommunikation, t_a den Zeitaufwand für eine einzelne arithmetische Operation, n die Anzahl der zu übertragenden Daten (also z.B. n Gleitkommazahlen) und α und β systemabhängige Konstanten bezeichnen. Dabei entspricht α dem Zeitfaktor gegenüber t_a für das Anstoßen einer

Kommunikationsoperation und β dem Zeitfaktor gegenüber t_a für das Verschicken eines Datums (also z.B. einer Gleitkommazahl). Typische Werte sind dabei $\alpha = 1000$ und $\beta = 10$. Daraus ergeben sich einige Kernpunkte, die bei der Erstellung von parallelen Programmen für Systeme mit verteiltem Speicher zu beachten sind:

- Unnötige Kommunikation ist zu vermeiden.
- Wenn Kommunikation erforderlich ist, sollten möglichst viele Daten in einem Kommunikationsvorgang verschickt werden, statt wenige in vielen Kommunikationsvorgängen.
- Andererseits darf die Kommunikation auch nicht soweit eingeschränkt werden, dass viele Knoten sich im Leerlauf befinden, da ihnen notwendige Daten für ihre Berechnungen fehlen. Hierbei spricht man auch von Lastverteilung (*load balancing*).

Der große Vorteil solcher Systeme ist, dass sie weitaus größer sein können als übliche Shared Memory Systeme. Diese sind nach aktuellem Stand der Technik auf relativ wenige Rechenkerne beschränkt, auch der Hauptspeicher kann nur bis zu einer gewissen Größe ausgebaut werden. Bei Systemen mit verteiltem Speicher hingegen können enorm viele Knoten in einem System zusammengeschlossen werden, wodurch auch der insgesamt zur Verfügung stehende Speicher enorme Kapazitäten erreichen kann. Weiterhin ist es möglich (und heutzutage auch üblich), für die Knoten dieser Distributed Memory Systeme wiederum Shared Memory Systeme zu verwenden. Jeder Knoten im Netz nutzt also einen oder mehrere Prozessoren mit mehreren Kernen. So können die einzelnen Knoten ihre Berechnungen wiederum mittels OpenMP oder ähnlicher Methoden lokal parallelisieren, während das Gesamtproblem global vom ganzen Cluster bearbeitet wird. Auch GPUs werden mittlerweile oft eingesetzt, um die Rechenleistung der einzelnen Knoten noch weiter zu steigern.

Auf Distributed Memory Systemen wird ein Programm gleichzeitig von allen beteiligten Prozessen ausgeführt (normalerweise wird pro Knoten ein Prozess gestartet), wobei innerhalb des Programms Verzweigungen anhand der eindeutigen ID des jeweiligen Prozesses vorgenommen werden. Dies entspricht also dem SPMD-Modell aus Abschnitt 2.2. Für die Implementierung der notwendigen Kommunikationsvorgänge wird in den meisten Fällen MPI (*Message Passing Interface*) verwendet, ein offener Standard der Universität von Tennesse [94,95]. Entsprechende Implementierungen existieren von verschiedenen Quellen, z.B. MPICH und MPICH2, OpenMPI und IntelMPI.

MPI stellt insbesondere diverse Kommunikationsfunktionen zur Verfügung, die entweder für die direkte Kommunikation zweier Knoten (*point-to-point communication*) oder für die gemeinsame Kommunikation in einer Gruppe von Knoten (*collective communication*) zuständig sind. Zur ersten Kategorie zählen insbesondere die Funktionen

- `MPI_Send` zum Senden von Daten und
- `MPI_Recv` zum Empfangen von Daten.

Beide Funktionen sind normalerweise blockierend, d.h. das Programm läuft erst weiter, wenn der Sende- bzw. Empfangsvorgang abgeschlossen ist. Es gibt jeweils allerdings

mehrere verschiedene Varianten, zu denen u.a. auch nicht blockierende Versionen dieser Funktionen zählen. Für einen kompletten Überblick sei hier auf den Standard [95] verwiesen.

Zu den Funktionen für gemeinsame Kommunikation zählt insbesondere die Funktion `MPI_Bcast`, welche Daten von einem Knoten an alle anderen Knoten in derselben Gruppe sendet (*Broadcast*). Eine solche MPI-Gruppe kann alle zur Verfügung stehenden Knoten umfassen (hierzu steht als Vorgabe die Gruppe `MPI_COMM_WORLD` zur Verfügung), der Programmierer kann aber auch eigene Gruppen definieren. Dies ist z.B. dann sinnvoll, wenn die Knoten für eine einfachere Implementierung eines Algorithmus logisch in einer bestimmten Form, z.B. in einem zweidimensionalen Gitter, angeordnet werden sollen oder wenn man eine spezielle physikalische Netzwerktopologie (wie Ring, Torus oder Gitter) abbilden will, um die Kommunikationseffizienz zu steigern.

Weiterhin stellt MPI diverse weitere Funktionen zur Verfügung, welche u.a. zu Verwaltungszwecken (z.B. Abfragen der eigenen Prozess-ID), zum Definieren von zu kommunizierenden Daten (z.B. können beliebige Daten in einem Puffer mittels `MPI_Pack` zusammengefasst werden) und zum Erstellen eigener Gruppen (s.o.) dienen. Für einen kompletten Überblick sei hier auf den MPI-Standard verwiesen [95]. Gute Einführungen in die Programmierung mit MPI finden sich bei Pacheco [111] und Karniadakis et al. [69].

Zum Abschluss dieses Abschnitts soll noch ein kurzes, einfaches Beispielprogramm zur Verwendung von MPI betrachtet werden. Dieses wird in Listing 2.3 angegeben.

```
1  #include <iostream>
2  #include <mpi.h>
3
4  using namespace std;
5
6  int main(int argc, char *argv[]) {
7
8    int mypid;  // Eigene Prozess ID
9    MPI_Status status; //Status Flag
10
11   // MPI Initialisierung
12   MPI_Init(&argc, &argv);
13
14   //Eigene ID bestimmen
15   MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
16
17   float pi;
18
19   if (mypid==0) {
20
21      pi = 3.14159;
22
23      cout << "Wert von Pi in Prozess 0:" << endl;
24      cout << pi << endl;
25
26      // Variable pi von Prozess 0 an Prozess 1 senden
27      MPI_Send(pi, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
```

```
28
29    } else if(mypid==1) {
30
31        //Prozess 1 empfaengt Pi von Prozess 0
32        MPI_Recv(pi, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
33
34        cout << "Von Prozess 1 empfangen:" << endl;
35        cout << pi << endl;
36    }
37
38    MPI_Finalize();
39
40    return 0;
41 }
```

Listing 2.3: Einfaches Beispielprogramm zur Verwendung von MPI

In diesem Programm wird eine Variable `pi` von dem Prozess mit der ID 0 mit dem Wert 3.14159 belegt und anschließend mittels `MPI_Send` an Prozess 1 gesendet. Beide Prozesse geben den gespeicherten Wert von `pi` aus, der bei erfolgreicher Kommunikation gleich sein sollte. Weiterhin wird die Initialisierung mittels `MPI_Init` und das Beenden von MPI mittels `MPI_Finalize` demonstriert. Die eigene Prozess ID lässt sich zur Laufzeit mittels `MPI_Comm_rank` abfragen.

Im Verlauf dieser Arbeit werden bei Bedarf benötigte MPI-Funktionalitäten noch genauer erklärt, z.B. zur Erzeugung und Verwendung von MPI-Gruppen in Abschnitt 4.1.3 und zur Kommunikation von eigenen Datentypen in Abschnitt 3.4. Die Ausführungen in diesem Abschnitt basieren neben den bisher schon angegebenen Literaturhinweisen vor allem auf Frommer [44].

2.3. Die C-XSC Bibliothek

C-XSC (*eXtended Scientific Computing*) ist eine C++ Klassenbibliothek für das wissenschaftliche Rechnen und stellt insbesondere Datentypen und Werkzeuge zur Arbeit mit Intervallen zur Verfügung. Die Entwicklung begann Anfang der 90er Jahre an der Universität Karlsruhe, aufbauend auf früheren Bibliotheken wie ACRITH-XSC [62] und PASCAL-XSC [74, 105]. Bei letzterem wurden Programme in Pascal geschrieben, wobei etliche Erweiterungen wie z.B. die einfache Notation und hochgenaue Berechnung von Skalarproduktausdrücken zur Verfügung standen. PASCAL-XSC Programme wurden zunächst in Standard C-Quellcode übersetzt, wonach sie dann mit einem üblichen C-Compiler für die jeweilige Maschine kompiliert werden konnten.

Einige Teile des Laufzeitsystems von PASCAL-XSC lassen sich heute noch in C-XSC finden, insbesondere die Implementierung des langen Akkumulators (siehe Bohlender [22] und Abschnitt 3.1.1). Des Weiteren baut die Funktionalität von C-XSC stark auf PASCAL-XSC auf. Die einzige wesentliche Ausnahme bilden die angesprochenen Skalarproduktausdrücke, die so nicht in C-XSC zur Verfügung stehen, deren Funktionalität sich aber (wenn auch nicht in so kurzer und eleganter Notation) auf andere Weise

in C-XSC abbilden lässt. Allerdings wurde eine allgemeine Vorgehensweise zur Verarbeitung solcher Skalarproduktausdrücke von Hammer [52] erarbeitet und ein erster Ansatz für eine entsprechende Umsetzung für C-XSC in Form eines Präcompilers wurde vom Autor im Zuge seiner Bachelor Thesis erstellt [145].

Im Folgenden soll die C-XSC Bibliothek näher vorgestellt werden. Zunächst folgt ein Abschnitt über die von C-XSC benutzten Datentypen und deren Operatoren. Danach wird genauer auf die sogenannte Toolbox eingegangen, welche Implementierungen einer Reihe von Algorithmen aus der Verifikationsnumerik zur Verfügung stellt. Schließlich folgt noch ein kurzer Überblick über einige Zusatzpakete, die zusätzliche Funktionalitäten bereitstellen. Ein allgemeiner und detaillierterer Überblick über C-XSC findet sich insbesondere bei Krämer und Hofschuster [59, 75]. Die Toolboxalgorithmen und deren Implementierung werden bei Hammer et al. [53, 54] näher besprochen.

2.3.1. Datentypen und Operatoren

C-XSC verwendet vier Grunddatentypen, auf denen letztlich auch alle anderen Datentypen in C-XSC basieren. Diese sind:

- `real`: Der Datentyp `real` ist eine Wrapperklasse für den Datentyp `double`. Sie wird zum einen aus historischen Gründen verwendet (in Analogie zu PASCAL-XSC), zum anderen aber vor allem um bestimmte Operatoren und Standardfunktionen neu überladen zu können, wie z.B. Ein- und Ausgabeoperatoren welche die Ein- und Ausgabeflags von C-XSC unterstützen.

- `interval`: Repräsentiert ein reelles Maschinenintervall, dessen Infimum und Supremum jeweils vom Typ `real` sind.

- `complex`: Repräsentiert eine komplexe Zahl, deren Real- und Imaginärteil vom Typ `real` sind.

- `cinterval`: Repräsentiert ein komplexes Rechteckintervall, dessen Real- und Imaginärteil vom Typ `interval` sind. In C-XSC wird für komplexe Intervalle grundsätzlich die komplexe Rechteckarithmetik verwendet.

Für jeden dieser Datentypen sind die jeweils sinnvollen arithmetischen Operatoren entsprechend Tabelle 2.3 und relationale Operatoren entsprechend Tabelle 2.4 gemäß der in Abschnitt 2.1.2 definierten Intervallarithmetik ebenso wie die gebräuchlichsten Standardfunktionen mit einer Genauigkeit von 1 *ulp* definiert.

Weiterhin steht für jeden Grunddatentyp ein zugehöriger Datentyp, welcher einen langen Akkumulator repräsentiert (`dotprecision`, `idotprecision`, `cdotprecision`, `cidotprecision`), zur Verfügung. Mit diesen kann die Funktionalität der Skalarproduktausdrücke aus PASCAL-XSC in C-XSC umgesetzt werden. Die Implementierung basiert auf C-Code aus dem PASCAL-XSC Laufzeitsystem, welcher in vielen Programmen benutzt wurde und inzwischen sehr ausgereift ist. Eine Auflistung der für die `dotprecision`-Typen verfügbaren Operatoren findet sich ebenfalls in Tabelle 2.3. Der lange Akkumulator und seine Verwendung in C-XSC werden in Abschnitt 3.1.1 näher erläutert.

Im Rahmen dieser Arbeit wurden die `dotprecision`-Typen außerdem um die Fähigkeit erweitert, Skalarprodukte mit K-facher *double*-Präzision zu berechnen. Details dazu finden sich in Abschnitt 3.1.

linker Operand \ rechter Operand	real complex	interval cinterval	dotprecision cdotprecision	idotprecision cidotprecision
monadisch	–	–	–	–
real complex	+, −, ∗, /, |	+, −, ∗, /, |	+, −, |	+, −, |
interval cinterval	+, −, ∗, /, |	+, −, ∗, /, |, `&`	+, −, |	+, −, |, `&`
dotprecision cdotprecision	+, −, |	+, −, |	+, −, |	+, −, |
idotprecision cidotprecision	+, −, |	+, −, |, `&`	+, −, |	+, −, |, `&`

\| : Konvexe Hülle `&` : Schnitt

Tabelle 2.3.: Arithmetische Operatoren für `dotprecision`- und Grunddatentypen (aus [59])

Aufbauend auf den Grunddatentypen stehen auch passende Matrix- und Vektordatentypen zur Verfügung, mit Elementen vom jeweiligen Grunddatentyp. Diese sind:

- Vektoren: `rvector`, `ivector`, `cvector`, `civector`
- Matrizen: `rmatrix`, `imatrix`, `cmatrix`, `cimatrix`

Die für Vektoren und Matrizen zur Verfügung stehenden Operatoren werden in den Tabellen 2.5 und 2.4 angegeben. Für Skalarproduktberechnungen, z.B. bei Matrix-Matrix-Produkten, wird dabei ein passendes `dotprecision`-Objekt verwendet. Mit den in Abschnitt 3.1 beschriebenen Änderungen ist die verwendete Präzision auch hierbei wählbar. In Abschnitt 3.2 wird beschrieben, wie alternativ hochoptimierte BLAS-Routinen für diese Berechnungen verwendet werden können.

Neben den in den Tabellen aufgeführten Operatoren stehen für die Matrix- und Vektordatentypen auch die `[]`- und `()`-Operatoren zur Verfügung. Ersterer dient dem Zugriff auf einzelne Elemente einer Matrix bzw. eine Vektors. Der `()`-Operator kann benutzt werden, um Bereiche aus einer Matrix oder einem Vektor auszuschneiden. Auf diese Ausschnitte kann dann sowohl schreibend als auch lesend zugegriffen werden.

Eine weitere Besonderheit ist, dass der Indexbereich für alle Matrix- und Vektordatentypen frei festgelegt werden kann. Standardmäßig wird die mathematische Indizierung verwendet, die Zählung beginnt also bei 1, alternativ kann aber z.B. auch auf eine 0-basierte umgeschaltet oder eine komplett andere Indizierung (auch negative Indizes sind

erlaubt) verwendet werden. Diese Fähigkeit hilft oftmals bei der Implementierung komplexerer Algorithmen.

linker Operand \ rechter Operand	real	interval	complex	cinterval	rvector	ivector	cvector	civector	rmatrix	imatrix	cmatrix	cimatrix
monadisch	!	!	!	!	!	!	!	!	!	!	!	!
real	$\vee_{all}$	$\vee_{\subset}$	$\vee_{eq}$	$\vee_{\subset}$								
interval	$\vee_{\supset}$	$\vee^{1}_{all}$		$\vee^{1}_{all}$								
complex	$\vee_{eq}$		$\vee_{eq}$	$\vee_{\subset}$								
cinterval	$\vee_{\supset}$	$\vee^{1}_{all}$	$\vee_{\supset}$	$\vee^{1}_{all}$								
rvector					$\vee_{all}$	$\vee_{\subset}$	$\vee_{eq}$	$\vee_{\subset}$				
ivector					$\vee_{\supset}$	$\vee^{1}_{all}$		$\vee^{1}_{all}$				
cvector					$\vee_{eq}$		$\vee_{eq}$	$\vee_{\subset}$				
civector					$\vee_{\supset}$	$\vee^{1}_{all}$	$\vee_{\supset}$	$\vee^{1}_{all}$				
rmatrix									$\vee_{all}$	$\vee_{\subset}$	$\vee_{eq}$	$\vee_{\subset}$
imatrix									$\vee_{\supset}$	$\vee^{1}_{all}$		$\vee^{1}_{all}$
cmatrix									$\vee_{eq}$		$\vee_{eq}$	$\vee_{\subset}$
cimatrix									$\vee_{\supset}$	$\vee^{1}_{all}$	$\vee_{\supset}$	$\vee^{1}_{all}$

$\vee_{all} = \{==, !=, <=, <, >=, >\}$ $\quad$ $\vee_{eq} = \{==, !=\}$

$\vee_{\subset} = \{==, !=, <=, <\}$ $\quad$ $\vee_{\supset} = \{==, !=, >=, >\}$

[1] <=: "Teilmenge von", <: "echte Teilmenge von", >=: "Obermenge von", >: "echte Obermenge von"

Tabelle 2.4.: Relationale Operatoren (aus [59])

Weiterhin stehen von den Datentypen `real` und `interval` und den zugehörigen Matrix- und Vektorklassen entsprechende Staggered-Varianten zur Verfügung, welche mit wählbarer Präzision K rechnen können (alle Werte, also auch das Ergebnis, haben K-fache *double*-Länge). Die Genauigkeit lässt sich dabei über die globale Variable `stagprec` einstellen. Die Namen dieser Klassen entsprechen den Namen der normalen Datentypen mit einem vorangestellten `l_`, also z.B. `l_real` und `l_interval`. Alle arithmetischen und relationalen Operatoren der einfachen Datentypen stehen auch für diese Staggered-Typen zur Verfügung. Weitere Details finden sich in der offiziellen C-XSC Dokumentation [59, 75].

Neben den bisher angesprochenen Operatoren sind auch Ein- und Ausgabeoperatoren (`<<`, `>>`) zur Verwendung mit dem Streamkonzept von C++ definiert. Diese Operatoren sorgen u.a. bei der Ein- und Ausgabe von Intervallen mit Strings in Dezimaldarstellung automatisch für eine korrekte Rundung. Weiterhin lassen sie sich über eine Reihe von Flags, ähnlich den in der C++-Standardbibliothek verwendeten Flags, vielfältig konfigurieren. Zu den zur Verfügung stehenden Flags zählen unter anderem:

- `SetPrecision(int w, int d)`: Setzt die Feldbreite (diese hat je nach gewähltem Format eine andere Bedeutung) für die Ausgabe auf `w` und die Anzahl der Stellen auf `d`.

linker Operand \ rechter Operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
monadisch	$-$	$-$	$-$	$-$	$-$	$-$
integer real complex	$+, -, *, /,$ $\|$	$+, -, *, /,$ $\|$	$*$	$*$	$*$	$*$
interval cinterval	$+, -, *, /,$ $\|$	$+, -, *, /,$ $\|, \&$	$*$	$*$	$*$	$*$
rvector cvector	$*, /$	$*, /$	$+, -, *^1,$ $\|$	$+, -, *^1,$ $\|$		
ivector civector	$*, /$	$*, /$	$+, -, *^1,$ $\|$	$+, -, *^1,$ $\|, \&$		
rmatrix cmatrix	$*, /$	$*, /$	$*^1$	$*^1$	$+, -, *^1,$ $\|$	$+, -, *^1,$ $\|$
imatrix cimatrix	$*, /$	$*, /$	$*^1$	$*^1$	$+, -, *^1,$ $\|$	$+, -, *^1,$ $\|, \&$

|: Konvexe Hülle &: Schnitt [1]: Skalarprodukt mittels `dotprecision`

Tabelle 2.5.: Arithmetische Operatoren für Vektoren und Matrizen (aus [59])

- `RndDown`: Ein- und Ausgaben werden zur nächstkleineren Gleitkommazahl gerundet.
- `RndUp`: Ein- und Ausgaben werden zur nächstgrößeren Gleitkommazahl gerundet.
- `Hex`: Ein- und Ausgabe werden auf hexadezimales Format umgestellt.
- `StoreOpt`: Die aktuell gewählten Einstellungen werden zwischengespeichert.
- `RestoreOpt`: Vorher zwischengespeicherte Einstellungen werden wieder geladen.

Alle zur Verfügung stehenden Flags werden in [75] erläutert.

In Abschnitt 3.3 wird die Erweiterung der C-XSC Bibliothek um Datentypen für dünn besetzte Matrizen und Vektoren mit entsprechenden Operatoren besprochen. Im Zuge dieser Änderung wurden die Ein- und Ausgabeflags um einige Möglichkeiten erweitert, welche ebenfalls in Abschnitt 3.3 genauer beschrieben werden. In neueren C-XSC Versionen (ab Version 2.5.0) finden sich außerdem Klassen für eine Staggered Darstellung mit extrem weitem Exponentenbereich. Die Namen dieser Klassen beginnen jeweils mit `lx_`, also z.B. `lx_interval`. Sie werden bei Blomquist et al. [21] detailliert besprochen.

2.3.2. Die C-XSC Toolbox

Zusätzlich zur C-XSC Kernbibliothek steht eine Toolbox genannte Sammlung von Implementierungen einiger nützlicher verifizierender Algorithmen zu weit verbreiteten numerischen Problemstellungen zur Verfügung. Die Toolbox wurde für PASCAL-XSC entwickelt [53] und später nach C-XSC portiert [54]. Die entsprechenden Implementierungen sind weitestgehend äquivalent. Zur Benutzung müssen nur die jeweiligen Header der verwendeten Module inkludiert werden, die Routinen selbst werden bereits bei der Kompilierung der C-XSC Bibliothek mit in die Bibliotheksdatei gelinkt.

Eindimensionale Problemstellungen

- **Auswertung reeller Polynome**: Berechnung einer maximal genauen Intervalleinschließung für reelle Polynome mit einer Variablen.
- **Automatische Differentiation**: Ein Ausdruck in einer Variablen wird in eine Baumstruktur überführt. Die Werte der Ableitung an einer bestimmten Stelle lassen sich dann automatisch unter Anwendung der entsprechenden Ableitungsregeln für jeden Knoten berechnen.
- **Nichtlineare Gleichungen in einer Variablen**: Es werden Einschließungen für alle Nullstellen einer nichtlinearen, eindimensionalen Funktion über eine Erweiterung des Intervall-Newton-Verfahrens berechnet.
- **Globale Optimierung**: Es werden Einschließungen aller Kandidaten für einen globalen Minimierer einer eindimensionalen reellen Funktion in einem Interval $\boldsymbol{x}$ berechnet. Zum Ausschluss von Kandidaten werden verschiedene Verfahren (Test auf Monotonie, Test auf Konkativität, Intervall-Newton-Schritt) verwendet.
- **Auswertung arithmetischer Ausdrücke**: Hochgenaue Auswertung arithmetischer Ausdrücke in einer Variablen mittels iterativer Verfeinerung.
- **Nullstellen komplexer Polynome**: Berechnung einer verifizierten Einschließung für alle Nullstellen eines komplexen (Intervall-)Polynoms.

Mehrdimensionale Problemstellungen

- **Lineare Gleichungssysteme**: Berechnung der Einschließung der Lösung eines linearen reellen Gleichungssystems mittels des Krawczyk-Operators. Diese Modul basiert auf dem gleichen Algorithmus wie die in Abschnitt 4.1 beschriebenen Löser.
- **Lineare Optimierung**: Berechnet die Einschließung der Lösung eines linearen Optimierungsproblems auf Basis des Simplex-Algorithmus.
- **Automatische Differentiation für Gradienten, Jacobi- und Hessematrizen**: Eine Erweiterung des Moduls zur automatischen Differentiation im eindimensionalen Fall auf den mehrdimensionalen Fall für Gradienten, Jacobi- sowie Hessematrizen.

- **Nichtlineare Gleichungssysteme**: Generalisierung des Moduls zum Finden der Nullstellen komplexer Polynome in einer Variable auf den mehrdimensionalen Fall mittels der erweiterten Intervall-Newton Gauß-Seidel Methode.
- **Globale Optimierung**: Erweiterung des Moduls zur globalen Optimierung im eindimensionalen Fall auf mehrdimensionale nicht-lineare Funktionen, basierend auf ähnlichen Ideen und der mehrdimensionalen automatischen Differentiation.

Im Zuge dieser Arbeit wurde die Toolbox um den in Abschnitt 4.1 beschriebenen Löser für lineare Gleichungssysteme erweitert, welcher deutlich flexibler und schneller ist, ohne dabei wesentlich an Genauigkeit zu verlieren. Dabei wurde auch eine zweite Implementierung des Hilfsmoduls zur Berechnung einer Näherungsinversen hinzugefügt, welche die Näherungsinverse mit einfacher Gleitkommarechnung berechnet, wobei optional LAPACK verwendet werden kann. Dieses neue Hilfsmodul wird in Abschnitt 3.2 genauer beschrieben.

2.3.3. Zusätzliche Pakete

Neben den in der C-XSC Bibliothek enthaltenen Datentypen und Funktionen sowie den Toolbox-Modulen stehen noch diverse Zusatzpakete als eigenständiger Download auf der C-XSC Webseite [2] zur Verfügung. Zu diesen Paketen gehören u.a.:

- Ein- und mehrdimensionale Intervall-Taylor-Arithmetik.
- Ein- und mehrdimensionale (Intervall)-Steigungs-Arithmetik.
- Eine Bibliothek komplexer Intervall-Standardfunktionen (*CoStLy*).
- Ein verifizierender, paralleler Löser für Fredholm-Integral-Gleichungen (siehe auch [50]).
- Eine MPI-Erweiterung für C-XSC (diese wird in Abschnitt 3.4 zusammen mit einigen Erweiterungen näher beschrieben).
- Eine Langzahlarithmetik für reelle und komplexe Intervalle auf Basis von MPFR und MPFI (siehe auch [21]).

Für eine vollständige Liste und Links zu entsprechenden erläuternden Artikeln und sonstiger Dokumentation für jedes Paket sei hier auf die C-XSC Webseite [2] verwiesen.

2.4. Numerische Bibliotheken

Im Verlauf dieser Arbeit werden diverse numerische Softwarebibliotheken verwendet, die jeweils benötigte Routinen in (oftmals) hochoptimierter Form bereitstellen. Die Nutzung solcher Bibliotheken ist auch im Bereich der Verifikationsnumerik von großer Bedeutung, um den unumgänglichen Laufzeitunterschied zwischen einem verifizierenden Algorithmus

und einem entsprechenden approximativem Algorithmus möglichst gering zu halten. Im Verlauf dieses Abschnitts werden die später in dieser Arbeit verwendeten externen Bibliotheken kurz vorgestellt.

2.4.1. BLAS

Die BLAS-Bibliothek (*Basic Linear Algebra Subprograms*) ist eine Sammlung von grundlegenden Routinen aus der linearen Algebra. Die Schnittstelle und Funktionsweise der Routinen sind standardisiert [40], wodurch prinzipiell jeder eine eigene Implementierung der BLAS-Bibliothek erstellen kann. Insbesondere Prozessorhersteller erstellen speziell für ihre Prozessoren optimierte Implementierungen, welche gegenüber naivem, „von Hand“ geschriebenem Code, z.B. in C oder C++, oftmals deutliche Geschwindigkeitsvorteile bieten.

Erreicht wird dies in erster Linie durch eine optimierte Ausnutzung des CPU-Caches. Der Cache ist ein enorm schneller, aber vergleichsweise kleiner Zwischenspeicher in der CPU. Liegen zur Verarbeitung einer Anweisung benötigte Daten hier vor, so können sie deutlich schneller in die entsprechenden Register geladen werden, als wenn sie zunächst aus dem Hauptspeicher oder schlimmstenfalls sogar von der Festplatte gelesen werden müssten. Weiterhin nutzen optimierte BLAS-Bibliotheken Mehrkernprozessoren sehr gut aus.

Die BLAS-Routinen werden in drei unterschiedliche Kategorien, genannt BLAS Level eingeteilt:

1. `BLAS Level 1`: Routinen mit einem Aufwand von $\mathcal{O}(n)$, insbesondere Vektor-Vektor Routinen wie einfache Skalarprodukte.

2. `BLAS Level 2`: Routinen mit einem Aufwand von $\mathcal{O}(n^2)$, insbesondere Matrix-Vektor Routinen wie Matrix-Vektor-Produkte.

3. `BLAS Level 3`: Routinen mit einem Aufwand von $\mathcal{O}(n^3)$, insbesondere Matrix-Matrix Routinen wie Matrix-Matrix-Produkte.

Die Routinen aus den unterschiedlichen BLAS Leveln sind unterschiedlich gut optimierbar. Generell gibt das Verhältnis von Speicherzugriffen zu arithmetischen Operationen einen groben Gradmesser für die Optimierungsmöglichkeiten, vor allem im Hinblick auf die Cache-Ausnutzung, an. Hierbei sind besonders die Routinen aus BLAS Level 3 sehr gut optimierbar und erreichen oftmals weit über 90% der theoretisch möglichen Rechenleistung (handgeschriebener Code kommt oft nicht über 10% hinaus).

Die Namen der BLAS-Routinen folgen einem einheitlichen Schema. Der erste Buchstabe des Namens gibt den zugrunde liegenden Datentyp an:

- **F**: Gleitkommazahl einfacher Genauigkeit.
- **D**: Gleitkommazahl doppelter Genauigkeit.
- **C**: Komplexe Gleitkommazahl einfacher Genauigkeit (Real- und Imaginärteil sind Gleitkommazahlen einfacher Genauigkeit).

- **Z**: Komplexe Gleitkommazahl doppelter Genauigkeit (Real- und Imaginärteil sind Gleitkommazahlen doppelter Genauigkeit).

Bei den Level 2 und Level 3 Routinen geben die nächsten beiden Buchstaben den Typ der betrachteten Matrizen an:

- **GE** (GEneral): Allgemeine Matrizen.
- **SY** (SYmmetric): Symmetrische Matrizen.
- **HE** (HErmitian): Hermitesche Matrizen.
- **TR** (TRiangular): (Untere) Dreiecksmatrizen.

Weiterhin existieren noch Kürzel für entsprechende Bandmatrizen und gepackte Matrizen. Für Details siehe das BLAS Benutzerhandbuch [20].

Die danach folgenden Buchstaben bezeichnen die eigentliche Routine. Ein Beispiel ist die für diese Arbeit bedeutsame Routine `DGEMM` zur Matrixmultiplikation. Der erste Buchstabe zeigt an, dass mit doppelter Genauigkeit gerechnet wird, der zweite und dritte Buchstabe, dass mit allgemeinen Matrizen gerechnet wird, und die letzten beiden Buchstaben geben die Art der Routine (*Matrix Multiplication*) an.

Ein wichtiger zu beachtender Punkt ist, dass BLAS Bibliotheken in FORTRAN implementiert sind. Für die Benutzung in C- und C++-Programmen gibt es die Möglichkeit, die verwendeten Routinen als extern zu deklarieren und später einfach die BLAS-Bibliothek hinzu zu linken. Hierbei muss allerdings beachtet werden, dass die Routinen je nach verwendetem FORTRAN Compiler eine andere Bezeichnung verwenden können (z.B. mit einem oder zwei hinzugefügten Unterstrichen, also z.B. `dgemm__`). Alternativ besteht die Möglichkeit, eine ebenfalls standardisierte C-Schnittstelle namens CBLAS zu verwenden.

In jedem Fall ist zu beachten, dass in FORTRAN Matrizen spaltenweise im Speicher liegen, während sie in C und C++ üblicherweise zeilenweise gespeichert werden. Bei der Verwendung der FORTRAN Routinen muss daher evtl. mit transponierten Matrizen gerechnet werden. Die CBLAS-Routinen können jeweils mit einem Flag so umkonfiguriert werden, dass sie entweder zeilen- oder spaltenweise abgespeicherte Matrizen verwenden. In dieser Arbeit werden BLAS-Routinen benutzt, um einige der C-XSC Operatoren für Matrizen und Vektoren optional um die Möglichkeit zu erweitern, mit diesen hochoptimierten Routinen zu arbeiten (siehe Abschnitt 3.2). Hierbei wird auch die C-Schnittstelle CBLAS verwendet.

Nahezu alle bekannten Prozessorhersteller bieten eigene BLAS-Implementierungen an, insbesondere Intel im Rahmen der MKL (*Math Kernel Library*) [64]. Bedeutsame Implementierungen, welche nicht von Prozessorherstellern stammen, sind Goto-BLAS [47] und ATLAS-BLAS [141]. Die Besonderheit bei ATLAS ist, dass es sich bei der Kompilierung der Bibliothek selbstständig für das jeweilige System optimiert, indem verschiedene Berechnungsvarianten getestet und die schnellste schließlich übernommen wird. Mit ATLAS steht damit auf nahezu jedem System eine optimierte, freie BLAS-Bibliothek zur

Verfügung. Weiterhin ist die BLAS-Referenzimplementierung frei verfügbar [1], welche aber keine speziell auf die verwendete Hardware angepassten Optimierungen verwendet.

Detailliert beschrieben wird die gesamte BLAS-Bibliothek im Benutzerhandbuch [20] und im BLAS-Standard [40].

2.4.2. LAPACK

Die LAPACK-Bibliothek (*Linear Algebra Package*) stellt komplexere Algorithmen aus der linearen Algebra zur Verfügung. Sie baut dabei auf der BLAS-Bibliothek auf, d.h. sie verwendet entsprechende BLAS-Routinen für ihre Berechnungen. Zur Verwendung von LAPACK wird daher auch immer eine BLAS-Bibliothek benötigt.

Ähnlich wie bei der BLAS-Bibliothek sind Interface und Funktionsweise von LAPACK standardisiert, so dass jeder Interessierte und insbesondere Hardwarehersteller eine eigene, optimierte Version von LAPACK implementieren können. Auch bei LAPACK handelt es sich um eine FORTRAN-Bibliothek. Zur Verwendung in C oder C++ müssen die entsprechenden Routinen daher wiederum als extern deklariert werden und es muss zu einer entsprechenden LAPACK Bibliothek gelinkt werden (mit den in Abschnitt 2.4.1 erwähnten Fallstricken). Weiterhin ist auch bei der Verwendung von LAPACK zu beachten, dass Matrizen spalten- und nicht zeilenweise gespeichert werden.

LAPACK stellt Routinen u.a. für folgende Problemstellungen bereit:

- LU-Zerlegung
- Cholesky-Zerlegung
- QR-Zerlegung
- Lösung von linearen Gleichungssystemen mit Dreiecksmatrix
- Matrixinvertierung (auf Basis einer LU-Zerlegung)
- Bestimmung von Eigenwerten und -vektoren
- Singulärwertzerlegung

Weiterhin stehen sogenannte *Driver-Routinen* zur Verfügung, welche mehrere der obigen Routinen kombinieren, um ein größeres Problem zu lösen (z.B. steht eine Routine zur Lösung eines linearen Gleichungssystems zur Verfügung, welche zunächst eine LU-Zerlegung durchführt und anschließend die beiden entstehenden Dreieckssysteme löst). Die Namenskonventionen der LAPACK-Routinen gleichen denen der BLAS-Bibliothek. Eine detaillierte Aufzählung und genaue Erklärung aller zur Verfügung stehenden Funktionen findet sich im LAPACK Users' Guide [13].

Oftmals wird LAPACK zusammen mit einer BLAS-Bibliothek ausgeliefert. Dies ist z.B. bei ATLAS [141] und der Intel MKL [64] der Fall. Eine nicht optimierte Referenzimplementierung steht ebenfalls zur Verfügung [5]. In Kombination mit einer optimierten BLAS-Bibliothek kann diese aber schon sehr gute Geschwindigkeiten erzielen, da vor allem die Routine zur Berechnung eines Matrix-Matrix-Produktes für viele

LAPACK-Routinen von Bedeutung ist. Im Rahmen dieser Arbeit wird LAPACK für die Berechnung einer Näherungsinversen bei den verifizierenden Lösern für dicht besetzte Gleichungssysteme (siehe Kapitel 4) und für die QR-Zerlegung einer Matrix bei einer Variante der verifizierenden Löser für dünn besetzte Gleichungssysteme (siehe Kapitel 5) verwendet. Eine Schnittstelle zwischen C-XSC und den benötigten LAPACK-Funktionen wird in Abschnitt 3.2 genauer erläutert.

2.4.3. ScaLAPACK

Um LAPACK-Routinen auf Systemen mit verteiltem Speicher aus der SPMD Kategorie (siehe Abschnitt 2.2), also insbesondere auf großen Clustern, effektiv nutzen zu können, steht mit ScaLAPACK (*Scalable LAPACK*) eine auf MPI basierende, speziell auf solche Systeme angepasste Version von LAPACK zur Verfügung (es wurde dabei allerdings nur eine Untermenge der in LAPACK zur Verfügung stehenden Routinen angepasst). Die Routinen sind dabei so implementiert, dass

- benötige Daten gleichmäßig auf alle Knoten verteilt werden,
- der Kommunikationsaufwand möglichst gering gehalten und
- die Rechenauslastung aller Knoten trotzdem auf einem möglichst hohen Niveau gehalten wird.

ScaLAPACK baut auf einer speziellen parallelen Version der BLAS-Bibliothek namens PBLAS (*Parallel BLAS*) sowie zu Kommunikationszwecken auf der BLACS-Bibliothek (*Basic Linear Algebra Communication Subprograms*) auf. Alle Kommunikationsvorgänge werden durch diese beiden Hilfsbibliotheken gesteuert. Weiterhin werden die eigentlichen Berechnungen lokal auf jedem Knoten durch die normalen BLAS und LAPACK Bibliotheken vorgenommen. Bei Verwendung optimierter Versionen dieser Bibliotheken werden so auch evtl. vorhandene Mehrkernprozessoren in jedem Knoten direkt ausgenutzt.

ScaLAPACK verwendet für alle Routinen eine zweidimensionale, blockzyklische Verteilung der Matrizen (für folgende Ausführungen wird von einer $m \times n$ Matrix ausgegangen). Hierbei werden die verfügbaren Prozessoren in einem zweidimensionalen Gitter der Größe $nr \times nc$ logisch angeordnet. Abbildung 2.4 zeigt ein entsprechendes Beispiel für ein 4×3 Gitter.

P_0	P_1	P_2
P_3	P_4	P_5
P_6	P_7	P_8
P_9	P_{10}	P_{11}

Abbildung 2.4.: Prozessgitter für die zweidimensionale, blockzyklische Verteilung

Die Einträge der Matrix werden dann in Blöcke der Größe $nb \times nb$ aufgeteilt, wobei nb frei gewählt werden kann. Die optimale Größe hängt vom jeweils verwendeten System ab.

Jeder dieser Blöcke wird dem Prozess in Zeile i und Spalte j im Prozessgitter zugeordnet, wobei $i = \frac{m}{nb} \bmod nr$ und $j = \frac{n}{nb} \bmod nc$ gelten. Jeder Prozess speichert die ihm zugeordneten Blöcke zusammengefasst in einer Matrix.

Abbildung 2.5 zeigt ein entsprechendes Beispiel für $m = n = 10$, $nb = 3$ und $nr = nc = 2$. Gezeigt wird dabei, in welche Blöcke die ursprüngliche Matrix aufgeteilt wird. Jeder Block ist mit der Bezeichnung des Prozesses beschriftet, dem er zugeordnet wird. Die grau markierten Blöcke zeigen zur Verdeutlichung alle Teile der Matrix, die dem Prozess P_0 zugeordnet sind. Dieser speichert also diese vier Blöcke zusammengefasst als eine Matrix.

$$\left(\begin{array}{|c|c|c|c|} \hline P_0 & P_1 & P_0 & P_1 \\ \hline P_2 & P_3 & P_2 & P_3 \\ \hline P_0 & P_1 & P_0 & P_1 \\ \hline P_2 & P_3 & P_2 & P_3 \\ \hline \end{array}\right)$$

Abbildung 2.5.: Aufteilung einer Matrix auf die einzelnen Prozesse bei zweidimensionaler, blockzyklischer Verteilung

Eine solche Aufteilung der Daten ermöglicht eine effiziente und skalierende Implementierung der jeweiligen Algorithmen. Durch die gleichmäßige Verteilung der Daten wird der im Cluster verfügbare Speicher annähernd optimal ausgenutzt, wodurch sich die maximal bearbeitbare Problemgröße entsprechend vergrößert.

ScaLAPACK wird im Rahmen dieser Arbeit für den parallelen verifizierenden Löser für allgemeine dicht besetzte Gleichungssysteme eingesetzt, welcher in Kapitel 4 beschrieben wird. Für weitere Details zu ScaLAPACK und seinen Hilfsbibliotheken sei an dieser Stelle auf den ScaLAPACK User's Guide verwiesen [19]. Neben der Referenzimplementierung gibt es teilweise auch eigene Implementierungen durch die Anbieter angepasster BLAS- und LAPACK-Versionen, z.B. in der Intel MKL [64].

2.4.4. SuiteSparse (UMFPACK und CHOLMOD)

SuiteSparse ist eine Sammlung einer Vielzahl von Paketen zur Arbeit mit dünn besetzten Matrizen, die von (bzw. unter Leitung von) Timothy A. Davis an der Universität von Florida erstellt wurden. Um die potentiellen Vorteile bei Speicher- und Rechenzeitbedarf bei der Verwendung von dünn besetzten Matrizen ausnutzen zu können, sind spezielle, teilweise sehr komplexe Datenstrukturen und Algorithmen nötig. SuiteSparse enthält

einige der am weitesten verbreiteten und teilweise auch in professioneller Software (z.B. Matlab) eingesetzten Pakete, welche zu den schnellsten ihrer Art gehören.

Als Datenstruktur wird für nahezu alle Pakete die sogenannte *Compressed Column Storage* (CCS) verwendet. Diese wird in Abschnitt 3.3 genauer erläutert. Zu den zur Verfügung stehenden Paketen zählen insbesondere die Folgenden, welche auch im Verlauf dieser Arbeit verwendet werden:

- **AMD**: Berechnet eine näherungsweise *Minimum Degree* Anordnung für eine dünn besetzte Matrix. Diese wird z.B. vor einer Cholesky Zerlegung zur Permutation der Matrix verwendet, um den sogenannten *Fill-In* (hinzufügen neuer Nicht-Null-Elemente zu den Dreiecksmatrizen während der Zerlegung) und somit Rechenzeit und Speicherbedarf zu reduzieren. Der Minimum Degree Algorithmus basiert auf einer Idee von Markowitz [92]. Für Details siehe auch George und Liu [45].

- **COLAMD**: Abwandlung des Minimum Degree Algorithmus, mit welcher eine Spaltenpermutation einer Matrix A berechnet wird, für welche insbesondere die LU-Zerlegung in der Regel weniger Fill-In aufweist als für die ursprüngliche Matrix.

- **CHOLMOD**: Berechnet die Cholesky Zerlegung für eine symmetrische dünn besetzte Matrix. Dabei wird auf das AMD Paket zurückgegriffen.

- **UMFPACK**: Berechnet die LU-Zerlegung für eine dünn besetzte Matrix unter Benutzung des COLAMD- (und evtl. auch des AMD-) Paketes.

- **CSparse** und **CXSparse**: CSparse ist eine eigenständige umfassende Bibliothek mit Algorithmen für dünn besetzte Matrizen. Dazu zählen grundlegende Algorithmen wie die Multiplikation dünn besetzter Matrizen, aber auch kompliziertere Algorithmen wie LU- oder Cholesky-Zerlegung. Die Bibliothek wurde vor allem zu Lehrzwecken erstellt und ist Begleitsoftware zu einem Buch [35]. Daher sind gerade die Algorithmen zur Matrixzerlegung nicht so schnell wie die aus den CHOLMOD und UMFPACK Paketen (dafür aber auch deutlich einfacher). Grundlegende Operationen wie das Matrix-Matrix-Produkt sind mit dieser Bibliothek allerdings enorm schnell. CXSparse ist eine Erweiterung von CSparse, welche auch Funktionen für komplexe Matrizen enthält.

Die meisten dieser Pakete werden in Kapitel 5 für die verifizierenden Löser für dünn besetzte Gleichungssysteme verwendet. CSparse und CXSparse werden in Abschnitt 3.3 zum Vergleich mit den C-XSC Datentypen für dünn besetzte Vektoren und Matrizen herangezogen. Da es sich bei SuiteSparse um einzelne Pakete handelt, existiert auch keine gemeinsame Dokumentation. Stattdessen gibt es eigene Artikel zu den einzelnen Paketen AMD [12], COLAMD [37], CHOLMOD [29] und UMFPACK [36] (ebenso auch zu den anderen, hier nicht aufgeführten Paketen). CSparse wird in einem eigenen Buch von Davis behandelt [35]. Eine Übersicht über alle Pakete und entsprechende Referenzen findet sich auf der SuiteSparse-Webseite [7].

2.5. Effizienz, Compiler und Compileroptionen

Der richtige Umgang mit dem verwendeten Compiler ist von enormer Bedeutung für das Laufzeitverhalten und gerade im Bereich der Verifikationsnumerik auch für die Korrektheit eines Programms. Um ein laufzeiteffizientes Programm zu erhalten, ist die Wahl sinnvoller Compileroptionen und eine für die durch den Compiler vorgenommenen Optimierungen vorteilhafte Organisation des Quelltextes von großer Wichtigkeit. Weiterhin ist bei der Implementierung einer Intervallarithmetik auf dem Rechner die Manipulation des Rundungsmodus für Gleitkommarechnungen essentiell, welche bei ungeschickter Programmierung und falsch gewählten Compileroptionen sowohl starke Auswirkungen auf die Geschwindigkeit als auch auf die Korrektheit des fertigen Programms haben kann.

In diesem Abschnitt sollen einige Grundlagen zur Effizienz selbst implementierter Programme und zum Umgang mit dem Compiler und der Wahl der richtigen Compileroptionen dargelegt werden. Insbesondere wird in Abschnitt 2.5.3 auf mögliche Probleme beim Umschalten des Rundungsmodus eingegangen. Bei den Beschreibungen wird dabei vornehmlich auf den GNU Compiler und den Intel Compiler eingegangen. Andere Compiler bieten in der Regel äquivalente Optionen an. Eine genaue Beschreibung der einzelnen Optionen der betrachteten und aller weiteren Möglichkeiten findet sich in den jeweiligen Handbüchern [63, 138].

2.5.1. Einige Grundlagen für effiziente Implementierungen

Eine grundlegende Regel bei der Optimierung von Programmen ist die sogenannte 80-20 Regel. Diese besagt, dass im Allgemeinen 20% des Codes 80% der Laufzeit ausmachen und man sich daher insbesondere auf diese Programmteile konzentrieren sollte. Um die Stellen im Quellcode zu finden, welche den meisten Aufwand verursachen, können sogenannte Profiler, wie z.B. Valgrind [140], eingesetzt werden. Für diese Arbeit sind aber folgende Punkte von allgemeiner Bedeutung:

- **Inlining**: Jeder Funktionsaufruf in einem Programm ist mit Verwaltungsaufwand verbunden, welcher etwas Rechenzeit benötigt. Weiterhin verhindert er eventuell einige Optimierungen durch den Compiler, sogenannte interprozedurale Optimierungen. Wird z.B. eine Funktion, an deren Anfang bestimmte Initialisierungen vorgenommen werden, innerhalb einer Schleife aufgerufen, ist es dem Compiler eventuell möglich (falls diese Initialisierungen vom aktuellen Schleifendurchlauf unabhängig sind), den entsprechenden Code vor die Schleife zu verschieben. Weiterhin bietet es auch Vorteile bei der Cache-Ausnutzung (die voraussichtlich nächsten Anweisungen werden von der CPU automatisch im sogenannten Instruction Cache hinterlegt), wenn der gesamte Code lokal vorliegt, während ein Funktionsaufruf in diesem Zusammenhang einer Verzweigung entspricht.

  Um diese negativen Auswirkungen, welche vor allem beim Aufrufen kleiner Funktionen entstehen, zu umgehen, kann das sogenannte Inlining eingesetzt werden.

Hierbei wird der Code der Funktion vom Compiler direkt an die Stelle des Aufrufs kopiert, wodurch der Aufwand für den Funktionsaufruf umgangen und dem Compiler weitergehende Optimierungen ermöglicht werden. Die Verwendung von Inlining ist gerade bei extensiver Verwendung von Operatorüberladung, wie sie in der C-XSC-Bibliothek stattfindet, von großem Vorteil, da jede Benutzung eines überladenen Operators einem Funktionsaufruf entspricht.

Inlining kann vom Compiler automatisch vorgenommen werden (je nach Optimierungseinstellungen, siehe Abschnitt 2.5.2). Der Benutzer kann dem Compiler durch voranstellen des Schlüsselwortes `inline` vor die Funktionsdeklaration aber auch anzeigen, dass diese Funktion möglichst inline ausgeführt werden sollte (dies ist allerdings nur eine Empfehlung an den Compiler). Die Entscheidung, ob eine Funktion inline ausgeführt wird, trifft der Compiler anhand bestimmter Grenzwerte, welche in erster Linie die Größe der Funktion betreffen, da jede Funktion, welche als Inline kompiliert wird, die Codegröße und damit die Größe des späteren Binärcodes des Programms anwachsen lässt. Die entsprechenden Grenzwerte können in der Regel ebenfalls über Compileroptionen beeinflusst werden.

Um dem Compiler Inlining überhaupt zu ermöglichen, muss die Definition der aufgerufenen Funktion in der gleichen Übersetzungseinheit vorhanden sein wie der Aufruf. Aus diesem Grund werden in C-XSC die meisten Definitionen in den Header-Dateien vorgenommen (bzw. in `*.inl`-Dateien, welche von den Headern inkludiert werden). Dies verlängert zwar die Kompilierungszeit von C-XSC-Programmen (da der entsprechende Teil der Bibliothek mit kompiliert werden muss), der Gewinn durch das Inlining ist durch die starke Verwendung von Operatorüberladung aber so groß, das dieser Nachteil (gerade auf modernen Maschinen) in Kauf genommen werden kann.

- **Cache-Ausnutzung**: Heutige CPUs werden in ihrer Leistungsfähigkeit vor allem dadurch beschränkt, dass die Daten, welche für eine Berechnung nötig sind, in der Regel nicht so schnell in die entsprechenden Register geladen werden können, dass die CPU ständig voll ausgelastet ist. Moderne CPUs sind also so schnell, dass sie im Wesentlichen durch die Datentransferrate des Hauptspeichers beschränkt werden. Um diesen Effekt zu mildern, wird ein CPU-naher, sehr schneller Zwischenspeicher, der Cache, benutzt, welcher von der CPU automatisch verwaltet wird. Der Programmierer kann allerdings für eine gute Ausnutzung des Caches sorgen, indem er versucht, möglichst oft während einer Berechnung auf die gleichen Daten zuzugreifen.

  Als Beispiel sei eine einfache Implementierung eines Matrix-Matrix-Produktes über drei for-Schleifen gegeben, wie in Listing 2.4 zu sehen.

```
1 const int n = 1000;
2 double A[1000][1000],B[1000][1000],C[1000][1000];
3
4 //...A und B mit Zufallswerten fuellen, C=0 setzen...
5
6 for(int i=0 ; i<n ; i++)
```

```
7     for(int j=0 ; j<n ; j++)
8       for(int k=0 ; k<n ; k++)
9         C[i][j] += A[i][k] * B[k][j];
```

Listing 2.4: Einfaches Matrix-Matrix-Produkt zur Veranschaulichung der Cache-Ausnutzung

Die Reihenfolge der for-Schleifen ist hierbei offensichtlich für das Endergebnis ohne Belang, auch bei Berücksichtigung von Rundungsfehlern: Bei Betrachtung des kompletten Produktes ändert sich zwar die Reihenfolge der Operationen, für jedes einzelne Element der Ergebnismatrix bleibt die Reihenfolge der Operationen aber gleich. Je nach Anordnung der for-Schleifen ändert sich aber die Abfolge der Speicherzugriffe und damit die Möglichkeit für die CPU, Daten im Cache zwischenzuspeichern. Misst man die Zeit für die verschiedenen Varianten, so erhält man typischerweise die Verhältnisse aus Tabelle 2.6 (Anmerkung: Manche Compiler können, je nach Optimierungsoptionen, diesen Code von sich aus optimieren, so dass dann möglicherweise kein Unterschied mehr festzustellen ist).

Variante	**Faktor**
ijk	1.0
ikj	0.18
jik	0.99
jki	2.02
kij	0.18
kji	2.02

Tabelle 2.6.: Geschwindigkeitsvergleich der verschiedenen Varianten des einfachen Matrix-Matrix-Produktes

In der Tabelle wird dabei der „Standardfall“ ijk mit 1 gewichtet. Wie zu sehen ist, sind die beiden Varianten, bei denen beide Matrizen zeilenweise durchlaufen werden (also so wie sie tatsächlich im Speicher vorliegen) deutlich am schnellsten, während die beiden Varianten, welche die Matrizen spaltenweise durchlaufen, sehr viel langsamer sind. Die anderen beiden Varianten durchlaufen jeweils eine Matrix zeilen- und eine spaltenweise, die Laufzeiten dieser Anordnungen liegt daher zwischen denen der anderen beiden Varianten.

Dieses Beispiel verdeutlicht das enorme Optimierungspotential durch eine gute Cache-Ausnutzung. In den meisten Fällen ist eine entsprechende Optimierung aber bei weitem nicht so einfach, wie in diesem Beispiel. Aus diesem Grund ist es vor allem von Vorteil, falls möglich entsprechend optimierte Bibliotheken wie BLAS und LAPACK zu verwenden. Eine sehr detaillierte Abhandlung zum Thema Cache-Ausnutzung findet sich bei Drepper [41].

- **Parallelisierung**: Hiermit ist in erster Linie das Ausnutzen von Prozessoren mit mehreren Rechenkernen, also eine Optimierung für Shared Memory Systeme z.B.

mittels OpenMP, gemeint. Da moderne Prozessoren nicht mehr, wie lange Zeit üblich und nun bedingt durch physikalische Grenzen nicht mehr ohne weiteres möglich, im Wesentlichen durch eine Steigerung der Taktfrequenz im Verlauf der Zeit an Geschwindigkeit gewinnen, sondern durch das Hinzufügen immer weiterer Rechenkerne, kann ihre volle Leistungsfähigkeit auch nur über eine entsprechende Anpassung von Programmen, also eine Parallelisierung, erreicht werden. Auch hier ist es von Vorteil, wenn verwendete externe Bibliotheken eine Parallelisierung bieten, so wie dies bei vielen BLAS- und LAPACK-Bibliotheken der Fall ist.

Ein weiterer wichtiger Punkt ist der richtige Umgang mit der Umschaltung des Rundungsmodus, welcher in Abschnitt 2.5.3 genauer erläutert wird. Allgemeine Vorgehensweisen, um effektiven C++ Code zu schreiben, finden sich z.B. bei Meyers [96].

2.5.2. Optimierungsoptionen

Compiler bieten eine Vielzahl von Optionen zur automatischen Optimierung, d.h. ohne dass Änderungen am Quellcode nötig wären. Die gebräuchlichsten Optimierungsoptionen werden dabei üblicherweise in drei Leveln gebündelt, welche bei den meisten Compilern über die Option `-Ox` aktiviert werden, wobei für `x` der entsprechende Level (1, 2, 3 oder 0 für keine Optimierungen) eingesetzt wird. Je höher der Optimierungslevel, desto mehr Optimierungsoptionen werden eingeschaltet. Der dritte Level bündelt vor allem solche Optionen, welche die Codegröße des kompilierten Programms anwachsen lassen können.

Wie in Abschnitt 2.5.1 erläutert, ist gerade die Aktivierung von Inlining wichtig. Dieses wird in der Regel bei Aktivierung der dritten Optimierungsstufe eingeschaltet. Insbesondere beim GNU-Compiler können die Limits, ab welchen eine Funktion vom Compiler inline kompiliert wird, sehr genau eingestellt werden (siehe [138], Abschnitt 3.10).

Bei Verwendung von Gleitkommaberechnungen gibt es diverse Fallstricke zu beachten. Viele Prozessoren (darunter die meisten x86-Prozessoren) verwenden Register mit einer Länge von 80 Bit für Gleitkommazahlen doppelter Länge. Diese sogenannte *Excess Precision* sorgt dafür, dass andere Ergebnisse als bei Verwendung des Standard-IEEE-*double*-Formates berechnet werden. Oftmals ist die höhere Genauigkeit von Vorteil, in manchen Fällen ist aber eine genaue Einhaltung des IEEE-Standards notwendig (z.B. bei den fehlerfreien Transformationen aus Abschnitt 3.1.3). Weiterhin kann durch einige Optimierungen die Reihenfolge von Gleitkommaoperationen verändert werden, was zu anderen Ergebnissen und evtl. unerwünschten Nebenerscheinungen führen kann. Im Folgenden werden jeweils einige in diesen Fällen nützliche Optionen des GNU und des Intel Compilers aufgeführt.

GNU Compiler

- `-mfpmath=sse`: Mit dieser Option werden für alle Gleitkommaoperationen die SSE-Register verwendet, welche keine Excess Precision aufweisen. Die berechneten Ergebnisse entsprechen somit dem IEEE-Standard. Mit dieser Option sind keine Laufzeiteinbußen verbunden (in der Regel wird der Code mit Verwendung der SSE-Register sogar schneller).

- `-ffloat-store`: Speichert alle Zwischenergebnisse im Hauptspeicher und rundet diese dabei in das *double* Format. Auf diese Art wird ebenfalls keine Excess Precision verwendet, allerdings kommt es zu erheblichen Geschwindigkeitseinbußen.

Intel Compiler

- `-fpmodel=strict|source|...` Mit dieser Option lässt sich das verwendete Gleitkommamodell wählen. Dies beeinflusst zum einen den Grad der Optimierung von Gleitkommaoperationen, zum anderen aber auch, ob die verwendete Genauigkeit der im Quelltext angegebenen Genauigkeit entspricht. Bei Verwendung von `strict` oder `source` wird keine Excess Precision verwendet, allerdings treten Laufzeiteinbußen auf. Alle verfügbaren Optionen werden in [63] aufgelistet.

Der Intel Compiler verwendet auf 64 Bit x86 Systemen grundsätzlich die SSE-Register für Gleitkommaberechnungen, wodurch die `-fpmodel` Option bei Problemen mit Excess Precision nicht verwendet werden muss. Für eine genaue Übersicht über alle Compileroptionen siehe [138] bzw [63]. Andere Compiler weisen üblicherweise ähnliche Optionen auf.

2.5.3. Nutzung des Rundungsmodus bei Gleitkommaberechnungen

Normalerweise wird bei Gleitkommaberechnungen jeweils zur nächstgelegenen Gleitkommazahl gerundet. Der IEEE-Standard sieht aber die Möglichkeit vor, den Rundungsmodus global einstellen zu können, insbesondere auf „Rundung zur nächstgrößeren Gleitkommazahl“ und „Rundung zur nächstkleineren Gleitkommazahl“. Je nach gewähltem Rundungsmodus wird ein entsprechendes Flag in einem speziellen Register, mit welchem die Gleitkommaeinheit des Prozessors konfiguriert wird bzw. in welchem auch entsprechende Exception Flags gesetzt werden (z.B. bei Auftreten von Unter- oder Überlauf), gesetzt.

Um den Rundungsmodus plattformunabhängig wechseln zu können, kann die im C99-Standard [65] definierte Funktion `fesetround` aus dem Header `fenv.h` verwendet werden. Die jeweilige Implementierung dieser Standardfunktion muss dafür sorgen, dass das entsprechende Flag dann korrekt gesetzt wird. Alternativ kann der Programmierer es auch per Assemblerbefehl setzen (diese Methode ist dann natürlich plattformabhängig). In jedem Fall sind Spezialfälle wie z.B. bei x86-Prozessoren zu beachten, wo die Konfiguration des mathematischen Co-Prozessors und der SSE-Einheit über zwei verschiedene Register erfolgt.

Allgemein ist beim Umschalten des Rundungsmodus zu beachten, dass jede Änderung dafür sorgt, dass die Pipeline des Prozessors geleert wird. Wird der Rundungsmodus also sehr häufig umgeschaltet, kann die Pipeline nicht effektiv genutzt werden, was sich sehr negativ auf die Geschwindigkeit eines Programms auswirkt. Bei der Programmierung ist also darauf zu achten, den Rundungsmodus möglichst selten umzuschalten.

Ein weiteres Problem ist, dass der Compiler beim Optimieren des Quellcodes eventuelle Änderungen des Rundungsmodus ignoriert, d.h. es wird davon ausgegangen, dass

der Rundungsmodus im Verlauf eines Programmes konstant bleibt. Dies kann zu fehlerhaften Programmen führen, wie das Programm in Listing 2.5 zeigt, für welches vom GNU Compiler 4.2 mit Optimierungsstufe `-O3` der in Listing 2.6 auszugsweise gezeigte Assemblercode erzeugt wird.

```
1  #include <fenv.h>
2  #include <iostream>
3  #include <iomanip>
4
5  using namespace std;
6
7  int main() {
8    double x, y, z1, z2;
9    x = 1.0;
10   y = 1.0e-20;
11
12   cout << setprecision(20) << scientific;
13
14   //Aufwaertsrundung
15   fesetround(FE_UPWARD);
16   z1 = x+y;
17
18   //Abwaertsrundung
19   fesetround(FE_DOWNWARD);
20   z2 = x+y;
21
22   cout << "z1=" << z1 << ", z2=" << z2 << endl;
23
24   return 0;
25 }
```

Listing 2.5: Einfaches Beispiel zur Rundungsmodusumschaltung

```
1          movq    _ZSt4cout(%rip), %rdx
2          movq    -24(%rdx), %rax
3          movq    $20, _ZSt4cout+8(%rax)
4          addq    -24(%rdx), %rdi
5          call    _ZSt10scientificRSt8ios_base
6          movl    $2048, %edi
7          call    fesetround
8          movl    $1024, %edi
9          call    fesetround
```

Listing 2.6: Auszug aus zu Listing 2.5 erzeugtem Assemblercode

Im Beispielprogramm wird zweimal die gleiche Addition vorgenommen, allerdings mit unterschiedlichen Rundungsmodi. Da die Rundungsumschaltung vom Compiler bei der Optimierung des Codes ignoriert wird, geht er davon aus, dass die beiden Additionen das gleiche Ergebnis liefern. Da somit eine der beiden Additionen überflüssig erscheint, wird im Zuge der Optimierung einfach direkt das Ergebnis der ersten Addition als Ergebnis der zweiten Addition verwendet. Weiterhin wird die Reihenfolge der Anweisungen verändert. Im Assemblercode ist zu sehen, dass die Addition (der `addq` Befehl) vor den

beiden Aufrufen von `fesetround` erfolgt. Die Addition wird hier also nur einmal mit dem Standardrundungsmodus vorgenommen und man erhält die falsche Ausgabe

```
z1=1.00000000000000000000e+00, z2=1.00000000000000000000e+00,
```

während eine Kompilierung ohne Optimierungen Reihenfolge und Berechnungen des Programms nicht beeinflusst und die richtige Ausgabe

```
z1=1.00000000000000022204e+00, z2=1.00000000000000000000e+00
```

erzeugt.

Im C99-Standard [65] ist ein Compiler-Pragma vorgesehen, welches solche Fälle verhindern soll. Durch Hinzufügen der Zeile `#pragma STDC FENV_ACCESS ON` soll bei den Optimierungen aller im jeweiligen Quellcodeblock noch folgenden Operationen beachtet werden, dass der Rundungsmodus gewechselt werden kann. Dieses Pragma ist im Standard allerdings nicht als verpflichtend gekennzeichnet und wird nur selten unterstützt. Es kann daher im Allgemeinen nicht verwendet werden.

Eine weitere Möglichkeit ist, eine entsprechende Compileroption zu wählen, welche diesen Effekt vermeidet. Der GNU Compiler stell hierfür die Option `-frounding-math` zur Verfügung. Diese ist aber auch in der bei Erstellung dieser Arbeit aktuellen Version 4.7 noch als experimentell gekennzeichnet und scheint in vielen Fällen (auch in obigem Beispiel) keine Auswirkung zu haben. In zukünftigen Versionen des GCC könnte sie aber das Problem beheben. Beim Intel Compiler kann die schon in Abschnitt 2.5.2 angesprochene Option `-fpmath=strict` verwendet werden, durch welche entsprechende Probleme mit dem Rundungsmodus komplett vermieden werden. Allerdings werden durch diese Option eine ganze Reihe von Optimierungen deaktiviert, wodurch der erzeugte Code deutlich langsamer wird. C-XSC umgeht dieses Problem zumindest auf x86-Systemen durch die Verwendung von Assemblercode für die entsprechenden Rundungsoperationen.

Insgesamt gilt es also bei der Verwendung des Rundungsmodus für Gleitkommaoperationen, welcher für die Implementierung einer Intervallarithmetik essentiell ist, große Sorgfalt sowohl in Bezug auf die Korrektheit als auch das Laufzeitverhalten der implementierten Programme walten zu lassen. In jedem Fall sollte die Dokumentation des verwendeten Compilers genau studiert werden, um die relevanten Optionen des Compilers zu kennen.

3. Erweiterungen der C-XSC Bibliothek

Die C-XSC Bibliothek bietet bereits sehr viele Datentypen und Funktionalitäten, welche bei der Erstellung eigener Programme im Bereich der Verifikationsnumerik von großem Nutzen sein können. Der Schwerpunkt bei der Entwicklung von C-XSC lag allerdings für lange Zeit auf Funktionalität und Verlässlichkeit, weniger Geschwindigkeit und Flexibilität. Weiterhin war die Bibliothek auf eine zukünftige Hardwareunterstützung für das extrem schnelle exakte Skalarprodukt aus Basis des im Verlauf dieses Abschnitts erläuterten langen Akkumulators ausgelegt, welche aber bis heute leider ausgeblieben ist und durch eine Softwareemulation ersetzt werden muss.

Für eine Anwendung im Bereich des High Performance Computing sind daher einige Anpassungen und Erweiterungen nötig. Prinzipiell würden sich viele dieser neuen Funktionalitäten auch als Zusatzpakete realisieren lassen, also außerhalb der eigentlichen Bibliothek. Eine Umsetzung in Form von Zusatzpaketen hat allerdings den Nachteil, dass der Endbenutzer für die Verwendung von Programmen wie den in dieser Arbeit beschriebenen Lösern diese zusätzlichen Pakete extra herunterladen und installieren müsste. Außerdem könnten dann die mit der Bibliothek mitgelieferten Programme aus der C-XSC-Toolbox nicht auf diese neuen Funktionalitäten zurückgreifen.

Aus diesen Gründen sind fast alle der in diesem Kapitel beschriebenen Erweiterungen bis C-XSC Version 2.5.4 sukzessive zur Kernbibliothek hinzugefügt worden. Zu den wesentlichen Änderungen zählen Skalarprodukte mit wählbarer Präzision, welche eine flexible Wahl zwischen Geschwindigkeit und Rechengenauigkeit je nach Anwendung ermöglichen, die Nutzung von hochoptimierten BLAS und LAPACK Routinen, welche zu einer drastischen Beschleunigung der betroffenen Berechnungen führen können, sowie eigene Datentypen für dünn besetzte Vektoren und Matrizen, welche eine Grundvoraussetzung für die später beschriebenen Gleichungssystemlöser mit entsprechenden Systemmatrizen sind. All diese (und weitere kleinere) Erweiterungen werden im Folgenden detailliert erläutert.

3.1. Skalarprodukte in K-facher Arbeitsgenauigkeit

Die Berechnung von Skalarprodukten ist eine der grundlegendsten Aufgaben der Numerik. Auf der einen Seite besteht daher im Allgemeinen der Wunsch, diese möglichst schnell zu berechnen. Andererseits ist aber, wie in Abschnitt 2.1 gesehen, das Skalarprodukt bei Berechnung im Gleitkommaformat auch anfällig für Rundungsfehler, die zu völlig falschen Ergebnissen führen können. Daher ist, gerade im Bereich der Veri-

fikationsnumerik, eine genaue oder sogar exakte Berechnung von Skalarprodukten mit verlässlichen Fehlerschranken von großer Bedeutung.

In vielen Algorithmen der Verifikationsnumerik wird zunächst eine Näherungslösung berechnet, aus der dann später in einem Verifikationsschritt, oft unter Verwendung eines Fixpunktsatzes, eine Einschließung des Fehlers der Näherungslösung bestimmt wird. Im ersten Schritt ist eine möglichst hohe Genauigkeit der berechneten Näherungslösung wünschenswert, im zweiten Schritt ist es erforderlich, verlässliche Intervalleinschließungen berechnen zu können.

Für eine Bibliothek aus dem Bereich des wissenschaftlichen Rechnens wie C-XSC ist es also wünschenswert, bei der Berechnung von Skalarprodukten möglichst flexibel zwischen hoher oder sogar maximaler Genauigkeit auf der einen und Geschwindigkeit auf der anderen Seite wählen zu können. Eine solche Möglichkeit wurde in der C-XSC Version 2.3.0 eingeführt und wird im Folgenden genauer beschrieben. Eine ähnliche Umsetzung als Zusatzpaket in Form von ergänzenden Klassen zur Skalarproduktberechnung, aus der auch die im Folgenden beschriebene Implementierung hervorging, wird in [146] beschrieben.

3.1.1. Bisheriges Vorgehen in C-XSC

Eine grundlegende Prämisse bei der ursprünglichen Entwicklung von C-XSC (bzw. auch schon beim Vorgänger PASCAL-XSC) war es, dass alle Operatoren ein Ergebnis zurückliefern, dass maximal eine Rundung im Gleitkommaraster vom tatsächlichen, exakten Ergebnis entfernt ist. Dazu ist es erforderlich, dass alle Skalarprodukte, seien es explizite oder implizite, z.B. in Matrix-Matrix-Produkten, mit maximaler Genauigkeit berechnet werden können.

Zu diesem Zweck wird in C-XSC traditionell der sogenannte lange Akkumulator verwendet. Der Grundgedanke dabei ist, dass in einem Fixpunkt-Akkumulator ausreichender Länge das Ergebnis wiederholter Additionen bzw. Produkte von Gleitkommazahlen, wie sie bei der Berechnung von Skalarprodukten auftreten, stets maximal genau berechnet und gespeichert werden kann. Eine schematische Darstellung ist in Abbildung 3.1 zu sehen.

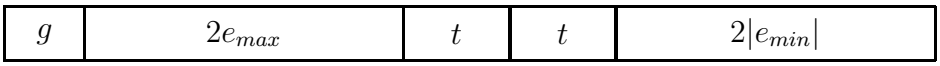


Abbildung 3.1.: Langer Akkumulator

Die Länge des Akkumulators ergibt sich aus folgender Überlegung: Bei der Berechnung eines Skalarproduktes $\sum a_i \cdot b_i,\ a_i, b_i \in \mathbb{F}$ liegen die einzelnen Summanden, also die Produkte $a_i \cdot b_i$, in einem Raster $R = (B, 2t, 2e_{min}, 2e_{max})$. Die Mantisse jedes Summanden mit Exponent Null lässt sich in $2t$ Bits exakt speichern. Ist der Exponent e des Summanden ungleich Null, so kann er durch verschieben der Mantisse um $|e|$ Ziffern nach links oder rechts (je nach Vorzeichen von e) in $2t + |e|$ Bits exakt gespeichert werden. Da für den Exponent e jedes Summanden $2e_{min} < e < 2e_{max}$ gilt, kann das Ergebnis also in einem Fixpunktregister der Länge $2t + 2|e_{min}| + 2e_{max}$ exakt gespeichert werden.

Um einen Überlauf zu verhindern, werden zusätzlich noch g Bits für einige Schutzziffern reserviert. Dabei wird g so gewählt, dass auch bei konstanter Summierung der größten darstellbaren Maschinenzahl während der Lebenszeit eines Rechners kein Überlauf auftreten kann.

Der lange Akkumulator und die entsprechenden Algorithmen, um Skalarprodukte bzw. allgemein Summen von Gleitkommazahlen mit seiner Hilfe maximal genau (bzw. vor Rundung in die Arbeitsgenauigkeit sogar exakt) zu berechnen, wird in vielen Artikeln und Büchern detailliert erklärt, insbesondere bei Kulisch [89] und Bohlender [22]. Forschungen an der Universität Karlsruhe [88] haben gezeigt, dass eine Hardwareimplementierung des langen Akummulators mit vergleichsweise geringem Aufwand möglich und dabei sogar schneller wäre, als normale Gleitkommarechnung, obwohl ein Ergebnis in maximaler Genauigkeit geliefert wird. Leider wird der Akkumulator aber bis heute nicht von allgemein verfügbarer Hardware unterstützt. Durch seine Aufnahme in den bevorstehenden IEEE-Standard 1788 [4] zur Intervallrechnung besteht mittlerweile zwar eine größere Wahrscheinlichkeit, dass sich dies in absehbarer Zukunft ändert. Bis dahin kann allerdings nur eine reine Softwareimplementierung, welche in der Regel auf Integer-Arithmetik basiert, genutzt werden, welche natürlich deutlich langsamer als eine Hardwareimplementierung und damit auch als normale Gleitkommarechnung ist. Eine solche Softwareimplementierung wird auch in C-XSC verwendet.

Der Akkumulator wird in C-XSC bei der Benutzung von Operatoren, welche direkt oder indirekt ein oder mehrere Skalarprodukte berechnen, implizit verwendet. Weiterhin steht er aber auch zur expliziten Verwendung als eigener Datentyp über die `dotprecision`-Klassen zur Verfügung. Mit Hilfe dieser Klassen ist es möglich, ganze Ausdrücke eines bestimmten Formats, sogenannte Skalarproduktausdrücke, exakt zu berechnen und ein Gleitkommaergebnis zu bestimmen, welches maximal eine Rundung im Gleitkommaraster vom exakten Ergebnis entfernt liegt.

Unter einem Skalarproduktausdruck wird eine Summe verstanden, deren einzelne Summanden Skalare, Vektoren oder Matrizen bzw. entsprechende einfache Produkte dieser sind. In PASCAL-XSC können solche Ausdrücke in einer speziellen Schreibweise, sogenannten #-Ausdrücken, welche stark an die übliche mathematische Notation angelehnt sind, angegeben werden. Diese werden dann intern mit Hilfe des langen Akkumulators berechnet. In C-XSC ist diese Notation aktuell nicht möglich. Stattdessen muss explizit mit den `dotprecision`-Klassen gearbeitet werden.

Als Beispiel für die Berechnung eines Skalarproduktes wird das Residuum $r = b - A\tilde{x}$ einer Näherungslösung $\tilde{x}$ eines linearen Gleichungssystems $Ax = b$, $A \in \mathbb{F}^{n\times n}$, $b, x \in \mathbb{F}^n$ betrachtet. Es soll eine engstmögliche Intervalleinschließung berechnet werden. Bei Verwendung von #-Ausdrücken könnte man diese Berechnung in der Form

```
r = ##(b - A*xa)
```

schreiben (`xa` ist die Näherungslösung). Das erste `#` leitet den Ausdruck ein, das zweite `#` gibt an, dass eine Intervalleinschließung berechnet werden soll. Für eine genaue Beschreibung des Aufbaus von #-Ausdrücken siehe Hammer [52] sowie die PASCAL-XSC Dokumentation [74, 105]. In C-XSC wird ein solcher Ausdruck, wie in Listing 3.1 zu sehen, mit Hilfe der `dotprecision`-Datentypen berechnet.

```
1  //Akkumulatorvariable
2  dotprecision dot;
3
4  for(int i=1 ; i<=n ; i++) {
5    dot = b[i];
6    //Skalarprodukt der i-ten Zeile von A mit xa berechnen
7    accumulate(dot, -A[i], xa);
8    //Aussenrundung des exakten Ergebnisses berechnen und zuweisen
9    rnd(dot, r[i]);
10 }
```

Listing 3.1: Berechnung des Skalarproduktausdrucks `r = ##(b - A*xa)` in C-XSC

Auf Basis von theoretischen Überlegungen von Hammer [52] wurde allerdings bereits eine erste Version eines Präcompilers für C-XSC implementiert [145], welcher die Benutzung von #-Ausdrücken prinzipiell ermöglicht. Dieser Präcompiler müsste allerdings für den praktischen Einsatz noch deutlich erweitert werden. Weiterhin müssten #-Ausdrücke und Präcompiler um Möglichkeiten zur Nutzung der neuen C-XSC Erweiterung zur freien Wahl der Skalarprodukt-Genauigkeit, welche in den folgenden Abschnitten näher erläutert wird, erweitert werden. Bis dahin muss auf die Nutzung von `dotprecision`-Variablen bei der Berechnung von Skalarproduktausdrücken zurückgegriffen werden.

3.1.2. Der DotK-Algorithmus

Als Alternative zur Benutzung des (langsamen aber hochgenauen) langen Akkumulators wird nun der von Oishi et al. eingeführte DotK-Algorithmus [106] betrachtet, der sich als gut geeignet für eine Implementierung innerhalb der C-XSC `dotprecision`-Klassen erwiesen hat (siehe dazu Abschnitt 3.1.3). Der DotK-Algorithmus ermöglicht die Berechnung von Skalarprodukten in ungefähr K-facher Arbeitsgenauigkeit (in dieser Arbeit wird dabei als Arbeitsgenauigkeit grundsätzlich vom IEEE-*double*-Format ausgegangen). Dazu werden ausschließlich normale Gleitkommaoperationen verwendet und es können auf einfache Art verlässliche Fehlerschranken berechnet werden, was eine sinnvolle Verwendung bei der Arbeit mit Intervallen ermöglicht. Die in diesem Abschnitt besprochenen Algorithmen werden wie im einführenden Artikel [106] in einer Matlab-ähnlichen Notation angegeben, da dies für die hier betrachteten Problemstellungen eine elegante und effiziente Formulierung ermöglicht. Die Qualität der mit dem DotK-Algorithmus berechneten Ergebnisse entspricht nicht exakt K-facher Genauigkeit, in der Folge wird der Einfachheit halber aber trotzdem von K-facher Genauigkeit gesprochen. Gemeint ist dabei stets die vom DotK-Algorithmus erreichte Genauigkeit für das jeweils gewählte K.

Im Folgenden wird davon ausgegangen, dass alle Berechnungen im IEEE-754 *double*- bzw. *binary64*-Format mit Rundung zur nächsten Gleitkommazahl vorgenommen werden. Außerdem wird das Vorhandensein von *gradual underflow* vorausgesetzt. Die in den Algorithmen angegebenen arithmetischen Operationen sind stets als Gleitkommaoperationen zu verstehen.

Der DotK-Algorithmus basiert auf der Benutzung der sogenannten fehlerfreien Transformationen (*error-free transformations*). Dieser Begriff wurde erst kürzlich geprägt

[106], das entsprechende Prinzip und die zugehörigen Algorithmen sind aber schon seit den 60er und 70er Jahren bekannt [23, 38, 76]:

Satz 3.1. *Seien $a, b, x \in \mathbb{F}$. Dann gibt es für alle Operationen $\circ \in \{+, -, \cdot\}$, sofern weder Über- noch Unterlauf auftreten, ein $y \in \mathbb{F}$, so dass gilt:*

$$a \circ b = x + y$$

mit $x = fl(a \circ b)$.

Dieser Satz besagt, dass der bei einer der angegebenen Gleitkommaoperationen auftretende Rundungsfehler selbst wiederum eine Gleitkommazahl ist und dass sich alle Informationen zum exakten Ergebnis der Operation mit Hilfe zweier Gleitkommazahlen (dem Ergebnis und dem Rundungsfehler) speichern lassen, deren exakte Summe das korrekte Ergebnis ist. Der Rundungsfehler y aus Satz 3.1 wird im Folgenden auch als Korrekturwert bezeichnet. Eine ähnliche Aussage gilt auch für die Division und die Quadratwurzel [23]. Diese Operationen werden aber hier nicht benötigt und daher auch in dieser Arbeit nicht weiter betrachtet.

Die Aussage dieses Satzes ist recht leicht einzusehen: Sowohl das Ergebnis der Addition als auch der Multiplikation zweier Gleitkommazahlen lässt sich jeweils mit maximal $2t$ Ziffern darstellen und kann somit auf zwei Gleitkommazahlen aufgeteilt werden. Bei normaler Gleitkommarechnung erhält man als Ergebnis die ersten t Ziffern.

Der Korrekturwert ist während einer normalen Gleitkommaberechnung im Prozessor sogar als Zwischenergebnis explizit vorhanden. Leider gibt es keine Möglichkeit für den Benutzer, diesen Wert direkt abzufragen, obwohl entsprechende Vorschläge schon vor langer Zeit veröffentlicht wurden [23] und der notwendige Hardwareaufwand sehr gering wäre.

Aus diesem Grund müssen die fehlerfreien Transformationen auf andere Weise bestimmt werden. Glücklicherweise lassen sich sowohl für die Addition als auch für die Multiplikation Algorithmen angeben, welche den gesuchten Korrekturwert nur mit Hilfe einfacher Gleitkommaoperationen bestimmen. Diese Algorithmen werden im Folgenden vorgestellt und genauer besprochen. Die Algorithmen und die entsprechenden Abschätzungen zur Genauigkeit stammen in der angegebenen Form aus Oishi et al. [106], sie sind allerdings schon länger bekannt. In den folgenden Betrachtungen wird grundsätzlich davon ausgegangen, dass kein Überlauf auftritt (näheres zur Sicherheit der Ergebnisse bei Auftreten von Überlauf in Abschnitt 3.1.3), Unterlauf wird aber explizit erlaubt.

Zunächst wird der Algorithmus *TwoSum* für die Berechnung der fehlerfreien Transformation der Gleitkomma-Addition nach Knuth [76, Abschnitt 4.2.2] betrachtet (Algorithmus 3.1). Dieser Algorithmus lässt sich auch direkt auf die Subtraktion anwenden, indem b auf $-b$ gesetzt wird.

Ein Beweis für die Korrektheit dieses Algorithmus findet sich bei Knuth [76]. Der Beweis bleibt auch bei Auftreten von Unterlauf korrekt, da Addition und Subtraktion dann exakt sind [56]. Nach Ablauf des Algorithmus erhält man also das normale Gleitkommaergebnis $x = fl(a + b)$ sowie den Korrekturwert y mit $|y| \leq eps|x|$ und $|y| \leq eps|a + b|$ [106].

Eingabe : $a, b \in \mathbb{F}$
Ausgabe : $x, y \in \mathbb{F}$, mit $x = \mathrm{fl}(a + b)$, für die bei exakter Rechnung $a + b = x + y$ gilt
x = a + b
z = x − a
y = (a − (x − z)) + (b − z)

Algorithmus 3.1: TwoSum

Als nächstes wird ein entsprechender Algorithmus zur fehlerfreien Transformation einer Gleitkomma-Multiplikation betrachtet. Hierzu wird zunächst ein anderer Algorithmus benötigt, welcher eine Gleitkommazahl a in zwei nicht überlappende Gleitkommazahlen x und y mit $|y| < |x|$ aufteilt, s.d. $a = x + y$. Hierzu wird Algorithmus 3.2 (*Split*) nach Dekker [38] verwendet.

Eingabe : $a \in \mathbb{F}$
Ausgabe : $x, y \in \mathbb{F}$, x und y nicht überlappend, für die bei exakter Rechnung $a = x + y$ gilt
c = factor · a
x = c − (c − a)
y = a − x

Algorithmus 3.2: Split

Die Konstante $factor$ berechnet sich hierbei aus $factor = 2^s + 1$ und $s = \lceil t/2 \rceil$, wobei t gegeben ist durch $eps = 2^{-t}$. Ein entsprechender Beweis findet sich ebenfalls bei Dekker [38]. Auch hier gilt wiederum, dass der Beweis auch im Unterlaufbereich seine Gültigkeit behält, da Addition und Subtraktion dann exakt sind.

Mit Hilfe von Algorithmus 3.2 kann nun der Algorithmus 3.3 (*TwoProduct*) nach Veltkamp [38] für die fehlerfreie Transformation des Produkts zweier Gleitkommazahlen angegeben werden.

Eingabe : $a, b \in \mathbb{F}$
Ausgabe : $x, y \in \mathbb{F}$, mit $x = \mathrm{fl}(a \cdot b)$, für die bei exakter Rechnung $a \cdot b = x + y$ gilt
x = a · b
$[a_1, a_2]$ = Split(a)
$[b_1, b_2]$ = Split(b)
$y = a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2)$

Algorithmus 3.3: TwoProduct

Ein Beweis für die Korrektheit von Algorithmus 3.3 findet sich wiederum bei Dekker [38]. Sofern kein Unterlauf auftritt, erhält man somit das Gleitkommaresultat $x = fl(a \cdot b)$ sowie den Korrekturwert y mit $|y| \leq eps|x|$ und $|y| \leq eps|a \cdot b|$. Sollte bei den fünf Multiplikationen in Algorithmus 3.3 Unterlauf auftreten, so erhält man durch eine grobe

Abschätzung [106] des dadurch entstehenden Fehlers stattdessen die Transformation $a \cdot b = x + y + 5\eta$ mit $|y| \leq eps|x| + 5eta$, $|y| \leq eps|a \cdot b| + 5eta$ und $|\eta| < eta$.

Bei allen angegebenen Algorithmen zur fehlerfreien Transformation ist zu beachten, dass sie nur einfache Gleitkommaoperationen verwenden und keinerlei Verzweigungen (if-Abfragen) enthalten, welche zu deutlichen Geschwindigkeitseinbußen durch das eventuell erforderliche Leeren der Pipeline führen können (siehe Abschnitt 2.5.1). Der Aufwand für den Algorithmus *TwoSum* beträgt 6 Gleitkommaoperationen, der Aufwand für den Algorithmus *TwoProduct* 17 Gleitkommaoperationen (inklusive je vier Gleitkommaoperationen für die beiden Aufrufe von *Split*). Einige Plattformen (z.B. Power PC und Intel Itanium) bieten eine Fused-Multiply-and-Add (*fma*) Operation in Hardware an. Bei dieser wird für drei Gleitkommazahlen $a, b, c \in \mathbb{F}$ die Berechnung $a \cdot b + c$ mit nur einer Operation und einer abschließenden Rundung berechnet. Steht diese Operation zur Verfügung, so lässt sich der Algorithmus *TwoProduct* einfacher realisieren (Algorithmus 3.4)

Eingabe : $a, b \in \mathbb{F}$
Ausgabe : $x, y \in \mathbb{F}$, mit $x = \mathrm{fl}(a \cdot b)$, für die bei exakter Rechnung $a \cdot b = x + y$ gilt
x = a + b
y = fma(a,b,-x)

Algorithmus 3.4: TwoProduct mit *fma*-Ausnutzung

Die Funktion `fma(a,b,c)` steht dabei für die Berechnung von $a \cdot b + c$ in Hardware mit nur einer Rundung. Steht diese Operation zur Verfügung, so reduziert sich also der Aufwand für *TwoProduct* auf nur noch 2 Gleitkommaoperationen.

Mit den Algorithmen für die fehlerfreien Transformationen können nun weiterführende Algorithmen formuliert werden. Zunächst wird dabei die Berechnung von Summen in doppelter Arbeitsgenauigkeit betrachtet. Hierzu dient Algorithmus 3.5 (*Sum2*).

Eingabe : $p \in \mathbb{F}^n$
Ausgabe : $fl(\sum p_i)$, in doppelter Arbeitsgenauigkeit
for $i = 2 : n$ **do**
  $[p_i, p_{i-1}] = \mathrm{TwoSum}(p_i, p_{i-1})$
$\mathrm{res} = \mathrm{fl}(\sum_{i=1}^{n-1} p_i + p_n)$

Algorithmus 3.5: Sum2

In diesem Algorithmus werden die Elemente eines Vektors $p \in \mathbb{F}^n$ aufsummiert. Zunächst wird p mittels kaskadierender Aufrufe von *TwoSum* transformiert. Dabei wird in jedem Durchlauf der for-Schleife das i-te Summenelement zum aktuellen Zwischenergebnis der normalen Gleitkommarechnung, welches jeweils an Position $i-1$ gespeichert wird, mittels *TwoSum* hinzu addiert. Das neue Gleitkomma-Zwischenergebnis wird an Position i gespeichert, der Korrekturwert für diese Addition an Position $i-1$. Nach Durchlauf der for-Schleife ist also p_n das Ergebnis der einfachen Gleitkommasummation und die

Elemente p_1 bis p_{n-1} sind die Korrekturwerte der $n-1$ Gleitkomma-Additionen. Da bisher nur mit fehlerfreien Transformationen gearbeitet wurde, gehen durch diese Umwandlung von p keinerlei Informationen verloren, die exakte Summe der Elemente von p wird also nicht verändert.

Um nun ein Ergebnis in doppelter Genauigkeit zu erhalten, werden die Elemente von p in normaler Gleitkommarechnung aufsummiert. Man kann zeigen [106], dass für das so berechnete Resultat res, auch bei Auftreten von Unterlauf, die Abschätzung

$$|res - \sum p_i| \leq eps|\sum p_i| + \gamma_{n-1}^2 \sum |p_i|$$

gilt, falls $n \cdot eps < 1$. Der Term $eps|\sum p_i|$ ist dabei die durch Rundung zur nächstgelegenen Gleitkommazahl unvermeidbare Fehlerschranke, da die exakte Summe in der Regel keine Gleitkommazahl sein wird. Der zweite Term $\gamma_{n-1}^2 \sum |p_i|$ wiederum zeigt, dass das Ergebnis die gleiche Qualität aufweist, als wäre es in doppelter Arbeitsgenauigkeit berechnet und anschließend gerundet worden. Bei der Fehleranalyse, die zu obiger Abschätzung führt, macht man sich zunutze, dass bei der Transformation in Algorithmus 3.5 keinerlei Informationen verloren gehen und man daher bei exakter Summation der Elemente des transformierten Vektors p auch die exakte Summe erhalten würde.

Eine Analyse der Transformation des Vektors p in Algorithmus 3.5 zeigt [106], dass die exakte Summe der Elemente von p zwar nicht verändert wird, die Konditionszahl der Summe sich aber um einen Faktor eps verbessert. Aus dieser Erkenntnis folgt direkt eine Methode für die Summation in K-facher Arbeitsgenauigkeit, welche in Algorithmus 3.6 (*SumK*) genutzt wird.

Eingabe : $p \in \mathbb{F}^n$, gewünschte Genauigkeit K
Ausgabe : $fl(\sum p_i)$, in K-facher Arbeitsgenauigkeit
for *k = 1 : K-1* **do**
  for *i = 2 : n* **do**
    $[p_i, p_{i-1}]$ = TwoSum(p_i,p_{i-1})
res = $\mathrm{fl}(\sum_{i=1}^{n-1} p_i + p_n)$

Algorithmus 3.6: SumK

Die Transformation des Eingabevektors wird nun also $K-1$ mal durchgeführt, wobei jede dieser Transformationen die Kondition um einen Faktor eps verbessert (für $K = 2$ ist der Algorithmus *SumK* offensichtlich identisch zum Algorithmus *Sum2*). Eine Analyse von Algorithmus 3.6 [106] ergibt, dass für das berechnete Ergebnis res die Abschätzung

$$|res - \sum p_i| \leq (eps + 3\gamma_{n-1}^2)|\sum p_i| + \gamma_{2n-2}^K \sum |p_i|$$

gilt, falls $4n \cdot eps < 1$. Der erste Term entspricht hierbei wieder der Rundung des Endergebnisses in Arbeitsgenauigkeit (der Term $3\gamma_{n-1}^2$ ist gegenüber eps vernachlässigbar), während der zweite Term aussagt, dass res die Qualität eines in K-facher Arbeitsgenauigkeit berechneten Resultats besitzt.

Als nächstes werden nun Algorithmen für die Berechnung von Skalarprodukten betrachtet. Wie bisher gesehen, kann mit dem Algorithmus *TwoProduct* das Produkt zweier Gleitkommazahlen in eine äquivalente Summe zweier Gleitkommazahlen umgewandelt werden, während der Algorithmus *SumK* die Möglichkeit bietet, Summen in K-facher Genauigkeit zu berechnen. Es liegt daher Nahe, beide Algorithmen für die Berechnung von Skalarprodukten in K-facher Genauigkeit zu kombinieren, wie in Algorithmus 3.7 (*SDotK*) zu sehen.

Eingabe : $x, y \in \mathbb{F}^n$, gewünschte Genauigkeit K
Ausgabe : Das Skalarprodukt $fl(x \cdot y)$, in K-facher Arbeitsgenauigkeit
for *i = 1 : n* **do**
  [r_i, r_{n+i}] = TwoProduct(x_i,y_i)
res = SumK(r,K)

Algorithmus 3.7: SDotK

Das Skalarprodukt der Länge n wird also zunächst mit Hilfe von *TwoProduct* in eine Summe der Länge $2n$ umgewandelt. Tritt kein Unterlauf auf, so gehen bei dieser Umwandlung keine Informationen über das exakte Ergebnis verloren (Unterlauf wird bei den folgenden Betrachtungen vorerst noch außer Acht gelassen). Diese Summe (und somit das Skalarprodukt) kann dann mit Hilfe des Algorithmus *SumK* in der gewünschten Genauigkeit K berechnet werden.

Allerdings hat Algorithmus 3.7 einen Aufwand von $6K(2n-1)+7n+5$ Gleitkommaoperationen, d.h. für den Fall $K = 2$ beträgt der Aufwand $31n - 7$ Operationen. Dieser Aufwand kann noch weiter reduziert werden, indem die von *TwoProduct* berechneten Korrekturwerte $r_{n+1}, \ldots, r_{2n}$ mit einfacher Gleitkommarechnung statt mit Hilfe von *TwoSum* summiert werden. Dadurch verringert sich der Aufwand für $K = 2$ auf $25n - 7$ Operationen. Da die Korrekturwerte $r_{n+1}, \ldots, r_{2n}$ in der Regel (zumindest bei den für eine Genauigkeit $K = 2$ wesentlichen Fällen) klein sind verglichen mit den Ergebnissen $r_1, \ldots, r_n$ der Gleitkommaprodukte, wird die Genauigkeit des Endergebnisses dadurch nicht entscheidend beeinflusst. Für $K = 2$ führt diese Änderung zu Algorithmus 3.8 (*Dot2*).

Eingabe : $x, y \in \mathbb{F}^n$
Ausgabe : Das Skalarprodukt $res = fl(x \cdot y)$, in doppelter Arbeitsgenauigkeit
[p,s] = TwoProduct(x_1,y_1)
for *i=2:n* **do**
  [h,r] = TwoProduct(x_i,y_i)
  [p,q] = TwoSum(p,h)
  s = s + (q+r)
res = p + s

Algorithmus 3.8: Dot2

Es lässt sich zeigen [106], dass für das Ergebnis res von Algorithmus 3.8 die Abschätzung

$$|res - x^T y| \leq eps|x^T y| + \gamma_n^2 |x^T||y|$$

gilt, sofern kein Unterlauf auftritt und $n \cdot eps < 1$. Ein Unterlauf hat nur innerhalb der Aufrufe von *TwoProduct* eine Auswirkung und kann wie schon erwähnt grob durch $5eta$ pro Aufruf nach oben abgeschätzt werden. Bei Auftreten von Unterlauf gilt damit die Abschätzung

$$|res - x^T y| \leq eps|x^T y| + \gamma_n^2 |x^T||y| + 5n \cdot eta.$$

Diese Abschätzung zeigt wiederum, dass das berechnete Ergebnis die gleiche Qualität hat, als wäre das Skalarprodukt in doppelter Arbeitsgenauigkeit berechnet und anschließend in die Arbeitsgenauigkeit gerundet worden.

Nun fehlt nur noch der Schritt zu K-facher Genauigkeit, für den ähnlich zur Grundidee aus Algorithmus 3.7, *Dot2* unter Benutzung des Summierungsalgorithmus *SumK* angepasst wird. Dieses Vorgehen wird in Algorithmus 3.9 (*DotK*) verwendet.

Eingabe : $x, y \in \mathbb{F}^n$, gewünschte Genauigkeit K
Ausgabe : Das Skalarprodukt $fl(x \cdot y)$, in K-facher Arbeitsgenauigkeit
[p,r_1] = TwoProduct(x_1,y_1)
for *i=2:n* **do**
  [h,r_i] = TwoProduct(x_i,y_i)
  [p,r_{n+i-1}] = TwoSum(p,h)
r_{2n} = p
res = SumK(r, K-1)

Algorithmus 3.9: DotK

Nach der ersten Umwandlung des Skalarproduktes in eine Summe der Länge $2n$ wird diese Summe also mit Hilfe von *SumK* mit einer Genauigkeit von $K - 1$ berechnet. Wiederum lässt sich zeigen [106], dass dann für das berechnete Endergebnis res die Abschätzung

$$|res - x^T y| \leq (eps + 2\gamma_{4n-2}^2)|x^T y| + \gamma_{(4n-2)}^K |x^T||y|$$

gilt, sofern $8n \cdot eps \leq 1$ und kein Unterlauf auftritt. Bei Auftreten von Unterlauf, welcher wiederum nur innerhalb der Aufrufe von *TwoProduct* eine Auswirkung hat, lässt sich dessen Effekt wieder grob nach oben mit $5n \cdot eta$ abschätzen, um die Abschätzung

$$|res - x^T y| \leq (eps + 2\gamma_{4n-2}^2)|x^T y| + \gamma_{(4n-2)}^K |x^T||y| + 5n \cdot eta$$

für das berechnete Ergebnis zu erhalten.

Diese Abschätzung zeigt, dass das berechnete Ergebnis die gleiche Qualität aufweist, als wäre es in K-facher Arbeitsgenauigkeit berechnet und anschließend in die Arbeitsgenauigkeit gerundet worden. Der Aufwand für Algorithmus 3.9 beträgt $6K(2n-1)+n+5$ Gleitkommaoperationen.

Für eine sinnvolle Verwendung des DotK-Algorithmus in der Verifikationsnumerik und insbesondere bei der Nutzung von Intervallarithmetik fehlt nun nur noch die Möglichkeit,

eine verlässliche Fehlerschranke für das Endergebnis nur mit Hilfe von Gleitkommaoperationen und ohne Rundungsmodusmanipulationen berechnen zu können. Aufgrund ihrer Wichtigkeit soll die Herleitung der Fehlerschranke [106] hier genauer angegeben werden. Zunächst wird dazu nur der Algorithmus *Dot2* betrachtet.

Mit den Größen res, p und s, sowie s_1, q_i und r_i aus Algorithmus 3.8 (s_1 steht für den der Variablen s beim ersten Aufruf von *TwoProduct* zugewiesenen Wert, q_i und r_i stehen für den Wert der Variablen q und r im i-ten Schleifendurchlauf) und unter Berücksichtigung von Unterlauf gilt zunächst die Abschätzung

$$\begin{array}{rcl} |x^T y - res| & \leq & eps|res| + |x^T y - p - s| + 5n \cdot eta \\ & \leq & fl(eps|res|) + |x^T y - p - s| + (5n+1) \cdot eta. \end{array} \tag{3.1}$$

Weiterhin gilt

$$\begin{array}{rcl} |x^T y - p - s| & = & \left| s_1 + \sum_{i=2}^{n} (q_i + r_i) - fl\left(s_1 + \sum_{i=2}^{n} (q_i + r_i) \right) \right| \\ & \leq & \gamma_{n-1} \left(|s_1| + \sum_{i=2}^{n} |fl(q_i + r_i)| \right). \end{array} \tag{3.2}$$

Mit

$$e := fl\left(|s_1| + \sum_{i=2}^{n} |q_i + r_i| \right)$$

und $\delta = \frac{n \cdot eps}{1 - 2n \cdot eps}$ lässt sich (3.2) nun weiterführen als

$$\begin{array}{rcl} \gamma_{n-1} \left(|s_1| + \sum_{i=2}^{n} |fl(q_i + r_i)| \right) & \leq & (1+eps)^{n-1} \gamma_{n-1} fl\left(|s_1| + \sum_{i=2}^{n} |q_i + r_i| \right) \\ & \leq & \gamma_{n-1}\gamma_{n-1} e \\ & \leq & \frac{(n-1)eps}{1-(2n-2)eps} e \\ & \leq & (1+eps)^{-3} \frac{n \cdot eps}{1 - 2n \cdot eps} e \\ & \leq & (1+eps)^{-2} \delta e. \end{array} \tag{3.3}$$

Einsetzen in (3.2) liefert dann

$$\begin{array}{rcl} |x^T y - res| & \leq & fl(eps|res|) + (1+eps)^{-2} \delta e + (5n+1) \cdot eta \\ & \leq & fl(eps|res|) + (1+eps)^{-1} fl(\delta e) + (5n+2) \cdot eta. \end{array} \tag{3.4}$$

Unter der Voraussetzung, dass $2n \cdot eps < 1$ folgt nun aus (3.4)

$$\begin{array}{rcl} |x^T y - res| & \leq & fl(eps|res|) + (1+eps)^{-1}(fl(\delta e) + 3eta/eps) \\ & \leq & fl(eps|res|) + fl(\delta e + 3eta/eps). \end{array} \tag{3.5}$$

Mit $\alpha = fl(eps|res| + \delta e + 3eta/eps)$ gilt schließlich

$$\begin{array}{rcl} |x^T y - res| & \leq & (1-eps)^{-1} \alpha \\ & \leq & (1-eps)\alpha/(1-2eps) \\ & \leq & fl(\alpha/(1-2eps)) \\ & =: & err. \end{array} \tag{3.6}$$

Damit folgt nun, dass für ein mit dem Algorithmus *Dot2* berechnetes Skalarprodukt $x^T y$ die Ungleichung

$$res - err \leq x^T y \leq res + err$$

erfüllt und somit das tatsächliche Resultat, auch unter Berücksichtigung von Unterlauf, verlässlich eingeschlossen werden kann. Eine entsprechend angepasste Version von *Dot2* wird in Algorithmus 3.10 (*Dot2err*) angegeben.

Eingabe : Zwei Vektoren $x, y \in \mathbb{F}^n$
Ausgabe : Das Skalarprodukt $fl(x \cdot y)$, in doppelter Arbeitsgenauigkeit, sowie eine Fehlerschranke *err* für das Ergebnis
[p,s] = TwoProduct(x_1,y_1)
e = abs(s)
for *i=2:n* **do**
    [h,r] = TwoProduct(x_i,y_i)
    [p,q] = TwoSum(p,h)
    t = q + r
    s = s + t
    e = e + abs(t)
res = p + s
$\delta = (n \cdot eps)/(1 - 2n \cdot eps)$
$\alpha = eps \cdot abs(res) + (\delta e + 3eta/eps)$
err = $\alpha/(1 - 2eps)$

Algorithmus 3.10: Dot2err

Der Aufwand für den so veränderten Algorithmus beträgt nun $27n + 4$ statt vorher $25n - 7$ Operationen. Die Fehlerschranke ist also ausschließlich mit normalen Gleitkommaoperationen und vergleichsweise geringem Aufwand berechenbar.

Als letztes fehlt nun noch eine entsprechende Fehlerabschätzung für den Algorithmus *DotK*, also die Berechnung in K-facher Genauigkeit. Auf eine solche Fehlerschranke wird im Originalartikel zum DotK-Algorithmus [106] nicht explizit eingegangen, es ist aber leicht einzusehen, dass sich die Berechnung der Fehlerschranke aus Algorithmus 3.10 direkt auf den allgemeinen Fall für K-fache Genauigkeit übertragen lässt.

Hierzu muss man sich vor Augen halten, dass der erste Schritt sowohl des Algorithmus *Dot2* als auch des Algorithmus *DotK* die Umwandlung des Skalarprodukts mit Hilfe der Algorithmen *TwoSum* und *TwoProduct* in eine Summe der Länge $2n$ ist. Der *DotK*-Algorithmus berechnet diese Summe dann aber in höherer Genauigkeit mit Hilfe von *SumK*. Dabei wird die Summe, abhängig von der verwendeten Genauigkeit, mehrfach *fehlerfrei* (auch bei Auftreten von Unterlauf) in eine neue Summe mit besserer Kondition umgewandelt.

Für die Berechnung der Fehlerschranke ist es also unerheblich, ob der Algorithmus *Dot2* oder *DotK* verwendet wird, da am Ende stets die gleiche Situation vorliegt: Die transformierte Summe der Länge $2n$ wird gleitkommamäßig berechnet und die Fehlerschranke muss den dabei auftretenden Fehler einschließen. Der bei der ursprünglichen

Umwandlung durch eventuellen Unterlauf verursachte Fehler lässt sich in beiden Fällen gleich abschätzen. Die Fehlerschranke des Algorithmus *DotK* für $K \geq 3$ wird automatisch besser (oder wenigstens gleichwertig) sein, da die Summe, in welche das Skalarprodukt transformiert wurde, besser konditioniert ist.

Um nun auch für den *DotK* Algorithmus eine verlässliche Fehlerschranke zu erhalten, reicht es aus, eine speziell angepasste Version des Algorithmus *SumK* zu formulieren und diese in Algorithmus 3.9 zu verwenden. Diese angepasste Version von *SumK* zeigt Algorithmus 3.11 (*SumKerr*).

Eingabe : Der Vektor $r \in \mathbb{F}^n$ aus Algorithmus 3.9
Ausgabe : Die Summe $fl(\sum r_i)$, in K-facher Arbeitsgenauigkeit, sowie eine Fehlerschranke err für das zugehörige Skalarprodukt
for *k = 1 : K-1* **do**
  for *i = 2 : n* **do**
    $[r_i, r_{i-1}] = \text{TwoSum}(r_i, r_{i-1})$

$\text{res} = \text{fl}(\sum_{i=1}^{n-1} p_i + p_n)$
$e = \sum_{i=1}^{n} |r_i|$
$\delta = (n \cdot eps)/(1 - 2n \cdot eps)$
$\alpha = eps \cdot abs(res) + (\delta e + 3eta/eps)$
$\text{err} = \alpha/(1 - 2eps)$

Algorithmus 3.11: SumKerr

Die Fehlerschranke wird hier sogar etwas zu hoch sein, da im Algorithmus *Dot2err* in der Schleife $|q_i + r_i|$ zu e hinzu addiert werden, während das Vorgehen in Algorithmus *SumKerr* dem hinzu Addieren von $|q_i| + |r_i|$ in *Dot2err* entsprechen würde. Dies hat aber nur geringe Auswirkungen auf die Fehlerschranke. Der Aufwand erhöht sich gegenüber der einfachen Version von *SumK* um $n + 14$ Operationen, d.h. der Mehraufwand ist unabhängig von K. Für den Algorithmus DotK bedeutet die Berechnung der Fehlerschranke einen Mehraufwand von $2n + 14$ Operationen.

Insgesamt stehen nun also die Grundalgorithmen für reelle Skalarprodukte mit doppelter oder sogar K-facher Genauigkeit zur Verfügung. Die tatsächliche Implementierung, sowie insbesondere auch die Anpassungen an komplexe oder Intervall-Skalarprodukte, die Einbettung in C-XSC und die dabei vorgenommenen Modifikationen werden im nächsten Abschnitt besprochen.

3.1.3. Implementierung der neuen Skalarproduktalgorithmen

Der beschriebene DotK-Algorithmus wurde bereits in einer früheren Arbeit [146] in Form eigener Klassen in einem separaten Paket für C-XSC implementiert. Die hier beschriebene Implementierung baut zwar grundsätzlich auf dieser früheren Version auf, ist aber in vielen Punkten erweitert und verbessert. Insbesondere ist die neue Implementierung

direkt in die C-XSC Bibliothek integriert und erweitert die vorhandenen `dotprecision`-Klassen. Dadurch steht die Funktionalität der Skalarprodukte in K-facher Genauigkeit jedem Benutzer der Bibliothek automatisch zur Verfügung. Weiterhin ermöglicht dies auch die Verwendung der alternativen Skalarproduktalgorithmen bei Benutzung von Operatoren, z.B. beim Matrix-Vektor-Produkt.

Zunächst wird die Implementierung der Algorithmen *TwoSum*, *Split* und *TwoProduct* für die fehlerfreien Transformationen betrachtet. Aufgrund ihrer Einfacheit können diese im Prinzip direkt in C++ übernommen werden, es ist jeweils nur auf einige wenige Details zu achten. Listing 3.2 zeigt zunächst die Implementierung von *TwoSum*.

```
1 static inline void TwoSum(real x, real y, real &a, real &b) {
2   a = x + y;
3   b = a - x;
4   b = ((x - (a - b)) + (y - b));
5 }
```

Listing 3.2: Implementierung von *TwoSum*

Alle Funktionen für die fehlerfreien Transformationen, sowie auch allgemein alle Funktionen für den DotK-Algorithmus, werden in einer eigenen Datei definiert, welche jeweils in den Quelldateien der betreffenden Datentypen (also den betroffenen Matrix- und Vektordatentypen sowie den `dotprecision`-Klassen) eingebunden wird. Dieses Vorgehen ermöglicht es dem Compiler, die Funktionen inline auszuführen, d.h. den eigentlichen Funktionsaufruf durch eine Kopie des tatsächlichen Quellcodes zu ersetzten, um so die Laufzeitkosten für den (vergleichsweise) teuren Verwaltungsaufwand eines normalen Funktionsaufrufs zu sparen. Hierzu werden die Funktionen auch als `inline` deklariert, um dem Compiler anzuzeigen, dass diese Funktionen möglichst auf diese Weise kompiliert werden sollen (für mehr Details siehe Abschnitt 2.5).

Weiterhin werden alle Funktionen auch als `static` deklariert. Dies ist nötig, da der entsprechende Quellcode in mehreren Quelldateien der C-XSC Bibliothek inkludiert wird, was normalerweise zu Konflikten führen würde, da dann dieselben Funktionen in mehreren Übersetzungseinheiten definiert wäre. Durch das Schlüsselwort `static` wird dem Compiler angezeigt, dass die betreffende Funktion nur für die aktuelle Übersetzungseinheit genutzt werden soll. Die Kompilierung wird dann so vorgenommen, als handelte es sich um unterschiedliche Funktionen, so dass es später beim Linken zu keinem Konflikt kommt.

Bei der Implementierung von *TwoSum* fallen noch einige weitere Details auf. Zum einen werden die Parameter `x` und `y` nicht als Referenzen, sondern als Kopien an die Funktion übergeben. Dies erscheint auf den ersten Blick als Nachteil, da eine Referenzübergabe in den meisten Fällen deutlich schneller ist. Der Datentyp `real` ist allerdings nur ein Wrapper für den Standarddatentyp `double` und hat daher genau wie dieser eine Größe von 8 Byte. Der entsprechende Referenzdatentyp ist auf 64-Bit-Systemen ebenfalls 8 Byte groß, so dass es hier keinerlei Auswirkungen auf die Laufzeit gibt. Auf 32-Bit-Systemen ist ein Pointer hingegen nur 4 Byte groß, so dass es hier zu einem gewissen Laufzeitverlust kommen kann, der sich in der Praxis aber als vernachlässigbar erweist.

Davon abgesehen würde eine Referenzübergabe später auch zu falschem Verhalten führen, da es bei den eigentlichen Skalarproduktberechnungen öfter zu Funktionsaufrufen

der Form `TwoSum(p[i-1],p[i],p[i-1],p[i]);` kommt, d.h. die Ein- und Ausgabeparameter `x` und `a` bzw. `y` und `b` beziehen sich jeweils auf dieselbe Variable. Bei Referenzübergabe würden daher die tatsächlichen Werte hinter `x` und `y` überschrieben werden, bevor in der letzten Zeile noch einmal Bezug auf die Originalwerte genommen wird. Hier tritt auch der interessante Fall auf, dass bei Verwendung von konstanten Referenzen für `x` und `y` die `const`-Deklaration über Umwege quasi umgangen werden würde, ein fehlerträchtiges Verhalten, welches auf den ersten Blick nicht unbedingt auffällt.

Außerdem wird als kleine Änderung gegenüber dem Originalalgorithmus keine temporäre Variable `c` für das Ergebnis der Berechnung in der zweiten Zeile verwendet. Stattdessen kann hier der Parameter `b`, welcher bis dahin noch nicht verwendet wurde, als Zwischenspeicher genutzt werden, wodurch das Anlegen einer zusätzlichen Variablen eingespart werden kann.

Als nächstes wird die Implementierung des Algorithmus *Split* betrachtet. Listing 3.3 zeigt den zugehörigen Quellcode.

```
1 static inline void Split(const real &x, real &x_h, real &x_t) {
2   x_t = Factor * x;
3   x_h = x_t - (x_t - x);
4   x_t = x - x_h;
5 }
```

Listing 3.3: Implementierung von *Split*

Auch bei der Implementierung von *Split* wird auf die Deklaration einer zusätzlichen Variablen für die Speicherung des Zwischenergebnisses aus der ersten Zeile verzichtet. Stattdessen wird hier der bis dahin nicht verwendete Parameter `x_t` genutzt. `Factor` ist hier als Konstante implementiert, so dass der Wert nicht bei jedem Aufruf neu berechnet werden muss. Die Schlüsselwörter `static` und `inline` werden hier ebenso wie bei der Implementierung des Algorithmus *TwoProduct* in Listing 3.4 verwendet.

```
1  static inline void TwoProduct(const real &x, const real &y,
2                                                real &a, real &b) {
3    #ifdef CXSC_USE_FMA
4      a = x * y;
5      b = fma(_double(x), _double(y), _double(-a));
6    #else
7      real x1,x2,y1,y2;
8      a = x * y;
9      Split(x,x1,x2);
10     Split(y,y1,y2);
11     b = x2 * y2 - (((a - x1 * y1) - x2 * y1) - x1 * y2);
12   #endif
13 }
```

Listing 3.4: Implementierung von *TwoProduct*

Die Implementierung ist eine direkte Umsetzung der Algorithmen *TwoProduct* und *TwoProductFMA*. Bei der Kompilierung von C-XSC kann die Nutzung von FMA über das Definieren der Präprozessorvariablen `CXSC_USE_FMA` aktiviert werden. Die FMA-Operation wird dabei über die Standardfunktion `fma` ausgeführt. Bei einer Benutzung

ist darauf zu achten, dass diese in der verwendeten Standardbibliothek korrekt implementiert ist. Ferner sollte diese Option nur mit entsprechender Hardwareunterstützung gewählt werden, da sich die Verwendung von `fma` sonst negativ auf die Laufzeit auswirkt.

Da der DotK-Algorithmus nicht nur für einfache reelle Skalarprodukte, sondern auch für komplexe und intervallwertige Skalarprodukte genutzt werden soll, sind spezielle Implementierungen einiger Funktionen für diese Datentypen nötig. Komplexe und intervallwertige Skalarprodukte lassen sich zwar mit mehreren reellen Skalarprodukten ausdrücken, für die Laufzeit sind direkte Anpassungen aber oftmals sinnvoller. Von den Funktionen für die fehlerfreien Transformationen gibt es daher spezielle Intervallversionen, welche auf die bereits beschriebenen Implementierungen für einfache reelle Zahlen zurückgreifen. Listing 3.5 zeigt die Implementierung von *TwoSum* für zwei Intervalle.

```
1 static inline void TwoSum(const interval &x, const interval &y,
2                           interval &a, real &b_i, real &b_s) {
3   real a_i,a_s;
4   TwoSum(Inf(x),Inf(y),a_i,b_i);
5   TwoSum(Sup(x),Sup(y),a_s,b_s);
6   a = interval(a_i,a_s);
7 }
```

Listing 3.5: Implementierung von *TwoSum* für Intervalle

Diese Implementierung von *TwoSum* ruft also zweimal die reelle Version auf, jeweils einmal für Infimum und Supremum. Das Gleitkommaergebnis ist dabei wieder vom Typ `interval` und wird in dem als Referenz übergebenen Parameter `a` gespeichert. Die Korrekturwerte können allerdings nicht als Intervall gespeichert werden, da der Korrekturwert des Infimums durchaus größer als der des Supremums sein kann. Sie werden daher mittels zweier unabhängiger Referenzparameter vom Typ `real` zurückgegeben. Der Aufwand für einen Aufruf der *TwoSum*-Implementierung für Intervalle ist also in etwa doppelt so hoch wie bei der Standardversion (die Erzeugung der temporären Variablen und der Aufruf des Konstruktors von `interval` sind vernachlässigbar gegenüber dem Aufwand der beiden Funktionsaufrufe).

Auch bei der Intervallversion von *TwoProduct* wird auf die Standardimplementierung Bezug genommen. Die Implementierung orientiert sich, wie jede allgemeine Multiplikationsroutine für zwei Intervalle in Infimum-Supremum-Darstellung, an der Multiplikationstabelle für Intervalle (Tabelle 2.1). Durch die vielen notwendigen Fallunterscheidungen ist ein Aufruf der Intervallversion von *TwoProduct* daher auch deutlich teurer als ein Aufruf der Standardversion. Die entsprechende Implementierung ist in Listing 3.6 gegeben.

```
1 static inline void TwoProduct(const interval &x, const interval &y,
2                               interval &a, real &b_inf, real &b_sup) {
3   real a_inf, a_sup;
4
5   if(Inf(x) >= 0  &&  Sup(x) >= 0) {
6
7     if(Inf(y) >= 0  &&  Sup(y) >= 0) {
8       TwoProduct(Inf(x), Inf(y), a_inf, b_inf);
9       TwoProduct(Sup(x), Sup(y), a_sup, b_sup);
```

```
10      } else if(Inf(y) < 0  &&  Sup(y) >= 0) {
11        TwoProduct(Sup(x), Inf(y), a_inf, b_inf);
12        TwoProduct(Sup(x), Sup(y), a_sup, b_sup);
13      } else {
14        TwoProduct(Sup(x), Inf(y), a_inf, b_inf);
15        TwoProduct(Inf(x), Sup(y), a_sup, b_sup);
16      }
17
18    } else if(Inf(x) < 0  &&  Sup(x) >= 0) {
19
20      if(Inf(y) >= 0  &&  Sup(y) >= 0) {
21
22        TwoProduct(Inf(x), Sup(y), a_inf, b_inf);
23        TwoProduct(Sup(x), Sup(y), a_sup, b_sup);
24
25      } else if(Inf(y) < 0  &&  Sup(y) >= 0) {
26
27        real ta1,ta2,tb1,tb2;
28
29        TwoProduct(Inf(x), Sup(y), ta1, tb1);
30        TwoProduct(Sup(x), Inf(y), ta2, tb2);
31        if((ta1 < ta2) || (ta1 == ta2 && tb1 < tb2)) {
32          a_inf = ta1;
33          b_inf = tb1;
34        } else {
35          a_inf = ta2;
36          b_inf = tb2;
37        }
38
39        TwoProduct(Inf(x), Inf(y), ta1, tb1);
40        TwoProduct(Sup(x), Sup(y), ta2, tb2);
41        if((ta1 > ta2) || (ta1 == ta2 && tb1 > tb2)) {
42          a_sup = ta1;
43          b_sup = tb1;
44        } else {
45          a_sup = ta2;
46          b_sup = tb2;
47        }
48
49      } else {
50
51        TwoProduct(Sup(x), Inf(y), a_inf, b_inf);
52        TwoProduct(Inf(x), Inf(y), a_sup, b_sup);
53
54      }
55
56    } else {
57
58      if(Inf(y) >= 0  &&  Sup(y) >= 0) {
59        TwoProduct(Inf(x), Sup(y), a_inf, b_inf);
60        TwoProduct(Sup(x), Inf(y), a_sup, b_sup);
61      } else if(Inf(y) < 0  &&  Sup(y) >= 0) {
```

```
62          TwoProduct(Inf(x), Sup(y), a_inf, b_inf);
63          TwoProduct(Inf(x), Inf(y), a_sup, b_sup);
64        } else {
65          TwoProduct(Sup(x), Sup(y), a_inf, b_inf);
66          TwoProduct(Inf(x), Inf(y), a_sup, b_sup);
67        }
68
69      }
70
71      SetInf(a,a_inf); SetSup(a,a_sup);
72    }
```

Listing 3.6: Implementierung von *TwoProduct* für Intervalle

Genau wie bei der Implementierung von *TwoSum* für Intervalle wird das Gleitkommaergebnis wieder als `interval` zurückgeliefert, während die Korrekturwerte als zwei unabhängige `real`-Werte gespeichert werden. Besonders zu beachten ist der Fall, in dem in beiden Faktoren die Null enthalten ist (der „mittlere“ Fall in der Multiplikationstabelle). Hier müssen bei der Berechnung von $\boldsymbol{x} \cdot \boldsymbol{y}$ jeweils $min(\underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y})$ sowie $max(\underline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y})$ berechnet werden. In dieser Implementierung wird dazu die fehlerfreie Transformation des jeweiligen Produktes mit *TwoProduct* vorgenommen und dann das Gleitkommaergebnis geprüft. Sind beide Gleitkommaergebnisse gleich, so wird mittels des Korrekturwertes der kleinere bzw. größere Wert ermittelt. Sollte bei der Berechnung der Korrekturwerte ein Unterlauf auftreten, so kann es hierbei zu einem falschen Ergebnis kommen. Dies wird aber später, bei der eigentlichen Berechnung der Skalarprodukte, über die Fehlerschranke wieder ausgeglichen, also dort mitberücksichtigt.

Neben den zusätzlichen Funktionen für zwei Intervalle existieren auch solche für „Mischfälle“, also z.B. die Multiplikation eines Intervalls und einer reellen Zahl. Diese sind direkte Anpassungen der bisher schon erläuterten Implementierungen.

Die Implementierung soll die bestehenden `dotprecision`-Klassen, welche ursprünglich zur Berechnung von Skalarprodukten und Skalarproduktausdrücken mittels des Akkumulators vorgesehen waren, um die alternative Berechnung mit dem *DotK*-Algorithmus sowie zusätzlich um die Möglichkeit, einfache Gleitkommarechnung für Skalarprodukte zu nutzen, erweitert werden. Bevor daher näher auf die Implementierung der eigentlichen Berechnungsroutinen eingegangen wird, werden zunächst die Modifikationen an den bestehenden `dotprecision`-Klassen erläutert.

Ursprünglich sind die `dotprecision`-Klassen ein Wrapper für den in C-Code implementierten Akkumulator aus dem sogenannten *Run-Time-System* von PASCAL-XSC. Sie haben daher als einzige Membervariable einen Pointer auf eine entsprechende Datenstruktur für den Akkumulator. Da nun aber die Berechnung in K-facher *double*-Präzision vorgenommen werden kann, muss zum einen die gewünschte Präzision K gespeichert werden. Dafür wird eine zusätzliche Membervariable `int k` eingeführt. Außerdem wird das Ergebnis nun nicht mehr exakt berechnet, d.h. es kann ein Fehler auftreten, welcher ebenfalls gespeichert werden muss, um später (falls gewünscht) die Berechnung einer Einschließung des korrekten Ergebnisses zu ermöglichen. Dafür wird eine zusätzliche Membervariable `real err` verwendet. Ebenso gibt es für beide Membervariablen entsprechende Zugriffsfunktionen.

Die Fehlervariable muss auch in einigen anderen Funktionen berücksichtigt werden, in welchen mit den `dotprecision`-Typen gearbeitet wird. Zum einen betrifft dies die relationalen Operatoren, welche nun nicht nur den im Akkumulator gespeicherten Wert heranziehen dürfen, sondern den Fehler mit einbeziehen müssen, so dass die verschiedenen Vergleiche eher dem entsprechenden Vorgehen bei Intervallen entsprechen. Zum anderen muss die Funktion `rnd` angepasst werden.

Diese diente ursprünglich dazu, das im Akkumulator exakt vorliegende Ergebnis in die Arbeitsgenauigkeit (also *double*-Genauigkeit) zu runden. Die entsprechenden Funktionen sind nun so angepasst, dass sie nicht nur diese Rundung vornehmen, sondern zusätzlich die jeweilige Fehlervariable addieren (bei Aufwärtsrundung) oder subtrahieren (bei Abwärtsrundung). Wird eine Intervalleinschließung mit `rnd` berechnet, so wirkt sich die Fehlervariable auf beide Grenzen aus.

Soll nun ein Skalarprodukt über die `accumulate` Funktion berechnet werden, so wird intern die Funktion `addDot` aufgerufen, welche gemäß der aktuell gewählten Präzision das weitere Vorgehen bestimmt. Da der Quelltext dieser Funktion sehr umfangreich ist, wird er im Folgenden in die wesentlichen Fälle

- $K = 0$: Benutzung des Akkumulators,
- $K = 1$: Reine Gleitkommarechnung,
- $K = 2$: Optimierter DotK-Algorithmus für doppelte Präzision und
- $K \geq 3$: DotK-Algorithmus für K-fache Präzision

aufgeteilt. Zunächst werden nur reelle Skalarprodukte betrachtet. Listing 3.7 zeigt den Fall $K = 0$.

```
1  for(int i=1 ; i<=n ; i++)
2    accumulate(val, x[i+lb1-1], y[i+lb2-1]);
```

Listing 3.7: Ausschnitt aus der Funktion *addDot* für $K = 0$

Hierbei (und im Folgenden) ist `val` vom Typ `dotprecision`, `x` und `y` sind die Vektoren, für welche das Skalarprodukt berechnet werden soll, `n` die Dimension der Vektoren und `lb1` bzw. `lb2` der jeweilige Startindex der beiden Vektoren. In diesem Fall werden also die beiden Vektoren einfach mit einer for-Schleife durchlaufen und die einzelnen Produkte des zu berechnenden Skalarprodukts werden einzeln akkumuliert. Etwas komplexer wird es für den Fall $K = 1$, wie in Listing 3.8 zu sehen.

```
1  real resd = 0.0, resu = 0.0;
2
3  #ifndef _CXSC_DOTK_ROUND_SOFT
4    setround(-1);
5    for(int i=1 ; i<=n ; i++)
6    resd += x[i+lb1-1] * y[i+lb2-1];
7
8    setround(1);
9    for(int i=1 ; i<=n ; i++)
10     resu += x[i+lb1-1] * y[i+lb2-1];
```

```
11
12    setround(0);
13    res = resd+(resu-resd)*0.5;
14
15    setround(1);
16    val.err += (resu-res);
17
18  #else
19
20    for(int i=1 ; i<=n ; i++)
21      resd = addd(resd, muld(x[i+lb1-1], y[i+lb2-1]) );
22
23    for(int i=1 ; i<=n ; i++)
24      resu = addu(resu, mulu(x[i+lb1-1], y[i+lb2-1]) );
25
26    res = resd+(resu-resd)*0.5;
27
28    val.err = addu(val.err, subu(resu,res) );
29  #endif
30
31  val += res;
```

Listing 3.8: Ausschnitt aus der Funktion *addDot* für $K = 1$

Hier wird das Skalarprodukt mit einfacher Gleitkommarechnung berechnet. Um dennoch eine verlässliche Fehlerschranke berechnen zu können, wird der Rundungsmodus des Prozessors umgeschaltet, d.h. das Skalarprodukt wird einmal mit Aufwärts- und einmal mit Abwärtsrundung berechnet, wobei sich der Fehler dann aus dem Radius des so berechneten Intervalls ergibt. C-XSC unterstützt auch weiterhin einen Modus für Software-Rundung (aktiviert durch Definition der Präprozessorvariablen `_CXSC_DOTK_ROUND_SOFT`, welche während der Installation bei Wahl von Software-Rundung automatisch gesetzt wird), der insbesondere für Plattformen gedacht ist, auf denen keine Rundungsmodusumschaltung in Hardware verfügbar ist (z.B. Cygwin). Hier wird eine Einschließung durch Benutzung der C-XSC Versionen zur Addition bzw. Multiplikation mit Rundung nach oben bzw. unten berechnet. Dieser Modus ist deutlich langsamer und hauptsächlich aus Kompatibilitätsgründen vorhanden.

Listing 3.9 zeigt das Vorgehen bei Verwendung des Algorithmus *Dot2*, also bei Wahl von $K = 2$.

```
1  real p, s, h, r, q, t;
2
3  TwoProduct(x[lb1],y[lb2],p,s);
4
5  err += abs(s);
6
7  for(int i=2 ; i<=n ; i++) {
8    TwoProduct(x[lb1+i-1],y[lb2+i-1],h,r);
9    TwoSum(p,h,p,q);
10   t = q + r;
11   s += t;
12   err += abs(t);
```

```
13 }
14
15 val += p;
16 val += s;
17
18 res = p+s;
19 real alpha, delta, error;
20
21 delta = (n*Epsilon) / (1.0-2*n*Epsilon);
22 alpha = (Epsilon*abs(res)) + (delta*err+3*MinReal/Epsilon);
23 error = alpha / (1.0 - 2*Epsilon);
24
25 val.err = addu(val.err, error);
```

Listing 3.9: Ausschnitt aus der Funktion *addDot* für $K = 2$

Dieser Teil der Funktion `addDot` ist im Wesentlichen eine direkte Umsetzung des Algorithmus *Dot2err*. Ein wichtiger Unterschied ist allerdings, dass das Ergebnis hier innerhalb eines Akkumulators (`val`) gespeichert wird. Dabei werden das reine Gleitkommaergebnis `p` und die Summe der Korrekturwerte `s` *einzeln* zum Akkumulator hinzu addiert. Die Summe der beiden Werte liegt also im Akkumulator exakt vor, insbesondere hat das Ergebnis dadurch doppelte *double*-Länge. Die berechnete Fehlerschranke wird zu der Membervariable `err` des Akkumulators hinzu addiert (mit Rundung nach oben, um in jedem Fall eine verlässliche Schranke zu erhalten).

Ähnlich wird auch bei der Implementierung für K-fache Genauigkeit vorgegangen, wie Listing 3.10 zeigt.

```
1  real r = 0, h;
2  real* t = new real[2*n];
3
4  for(int i=1 ; i<=n ; i++) {
5    TwoProduct(x[lb1+i-1], y[lb2+i-1], h, t[i-1]);
6    TwoSum(r, h, r, t[n+i-2]);
7  }
8
9  t[2*n-1] = r;
10 SumK(t, 2*n, val.k-1, err, val);
11
12 val.err = addu(val.err, err);
13
14 delete[] t;
```

Listing 3.10: Ausschnitt aus der Funktion *addDot* für $K \geq 3$

Hier wird zunächst nur die Transformation des Skalarprodukts der Länge n in eine Summe der Länge $2n$ vorgenommen und bereits mittels `TwoSum` eine erste Aufsummierung vorgenommen, ähnlich wie beim Vorgehen für doppelte Genauigkeit. Die Summanden der resultierenden Summe werden in einem dynamisch angelegten `real`-Array gespeichert. Zur Berechnung des Ergebnisses wird dann die Funktion `SumK` mit $K - 1$-facher Genauigkeit aufgerufen, welche in Listing 3.11 zu sehen ist.

```
1  static inline void SumK(real* p, int n, int k, real &err,
2                                                  dotprecision& val) {
3    real corr = 0.0, tmperr = 0.0, res = 0.0;
4    val += p[n-1];
5    res += p[n-1];
6
7    for(int j=1 ; j<k ; j++) {
8      for(int i=1 ; i<n-1 ; i++)
9        TwoSum(p[i],p[i-1],p[i],p[i-1]);
10
11     val += p[n-2];
12     res += p[n-2];
13     p[n-2] = 0.0;
14   }
15
16   for(int i=0 ; i<n-2 ; i++) {
17     corr += p[i];
18     tmperr += abs(p[i]);
19   }
20
21
22   real alpha, delta, error;
23
24   delta = (n*Epsilon) / (1.0-2*n*Epsilon);
25   alpha = (Epsilon*abs(res)) + (delta*tmperr+3*MinReal/Epsilon);
26   error = alpha / (1.0 - 2*Epsilon);
27
28   err = addu(err, error);
29
30   val += corr;
31 }
```

Listing 3.11: Implementierung von *SumK*

Gegenüber dem Originalalgorithmus gibt es einige Anpassungen. Zunächst ist *SumK* hier speziell für die Benutzung innerhalb des DotK-Algorithmus angepasst, insbesondere wird die Fehlerschranke für das vom DotK-Algorithmus berechnete Skalarprodukt direkt in *SumK* berechnet. Der Funktion übergeben werden ein Pointer `p` auf das Array mit den Summanden, welches sich durch die Transformation des ursprünglichen Skalarprodukts ergeben hat. Weiterhin werden die Größe `n` des Arrays (also die doppelte Dimension des ursprünglichen Skalarprodukts), die gewünschte Präzision `k`, ein Referenzparameter zum Speichern der berechneten Fehlerschranke und ein `dotprecision`-Objekt zur Speicherung des Endergebnisses übergeben.

Auch hier werden zu Beginn der Funktion in jedem Durchlauf der ersten for-Schleife und am Ende der Funktion das aktuelle Gleitkomma-Zwischenergebnis und die Summe der verbleibenden Korrekturwerte *einzeln* zum Akkumulator hinzu addiert. Dadurch erhält man wiederum im Akkumulator die exakte Summe der einzelnen Werte. Da die Korrekturwerte bei jedem Durchlauf in etwa um einen Faktor *eps* kleiner werden, erhält man so im Akkumulator das Ergebnis des DotK-Algorithmus in K-facher *double*-Länge. Bei expliziter Verwendung von `dotprecision`-Variablen bei der Berechnung von Skalarpro-

duktausdrücken können diese zusätzlichen Informationen für deutlich genauere Ergebnisse sorgen als bei Verwendung einer normalen Implementierung des DotK-Algorithmus. Näheres dazu folgt später in diesem Abschnitt.

Der ursprüngliche Algorithmus und die bisher erläuterte verbesserte Implementierung sind nur auf Skalarprodukte mit reellen Punktvektoren ausgelegt, d.h. es wurden bisher weder komplexe Vektoren noch (komplexe) Intervallvektoren betrachtet. Ein komplexes Skalarprodukt der Dimension n wird in zwei reelle Skalarprodukte der Länge $2n$ umgewandelt, jeweils eines für den Real- und Imaginärteil, für welche sich der DotK-Algorithmus anwenden lässt. Bei der Implementierung der komplexen Version werden daher zwei Skalarprodukte simultan berechnet, d.h. für jedes Element der Eingangsvektoren werden mittels `TwoProduct` und `TwoSum` die jeweils zwei Produkte für Realteil und Imaginärteil verarbeitet. Die Fehlerschranke wird ebenfalls getrennt für Real- und Imaginärteil berechnet. Listing 3.12 verdeutlicht die notwendigen Änderungen am Beispiel des Falls $K = 2$. Für die anderen Fälle sind analoge Änderungen vorzunehmen.

```
1  real p_re = 0.0, p_im = 0.0;
2  real s_re = 0.0, s_im = 0.0, h, r, q, t;
3  real err_re = 0.0, err_im = 0.0;
4
5  for(int i=0 ; i<n ; i++) {
6    TwoProduct(Re(x[lb1+i]), Re(y[lb2+i]), h, r);
7    TwoSum(p_re, h, p_re, q);
8    t = q + r;
9    s_re += t;
10   err_re += abs(t);
11
12   TwoProduct(-Im(x[lb1+i]), Im(y[lb2+i]), h, r);
13   TwoSum(p_re, h, p_re, q);
14   t = q + r;
15   s_re += t;
16   err_re += abs(t);
17
18   TwoProduct(Re(x[lb1+i]), Im(y[lb2+i]), h, r);
19   TwoSum(p_im, h, p_im, q);
20   t = q + r;
21   s_im += t;
22   err_im += abs(t);
23
24   TwoProduct(Im(x[lb1+i]), Re(y[lb2+i]), h, r);
25   TwoSum(p_im, h, p_im, q);
26   t = q + r;
27   s_im += t;
28   err_im += abs(t);
29 }
30
31 val += complex(p_re, p_im);
32 val += complex(s_re, s_im);
33
34 //Compute error bound
35 complex res = complex(p_re+s_re, p_im+s_im);
```

```
36 real alpha_re, alpha_im, delta, error_re, error_im;
37 delta = (2*n*Epsilon) / (1.0-4*n*Epsilon);
38
39 alpha_re = (Epsilon*abs(Re(res))) + (delta*err_re+3*MinReal/Epsilon);
40 error_re = alpha_re / (1.0 - 2*Epsilon);
41 Re(val).err = addu(Re(val).err, error_re);
42
43 alpha_im = (Epsilon*abs(Im(res))) + (delta*err_im+3*MinReal/Epsilon);
44 error_im = alpha_im / (1.0 - 2*Epsilon);
45 Im(val).err = addu(Im(val).err, error_im);
```

Listing 3.12: Ausschnitt aus der Funktion *addDot* für komplexe Skalarprodukte und $K = 2$

Für Intervallskalarprodukte werden die speziellen Intervallvarianten von `TwoSum` aus Listing 3.5 und `TwoProduct` aus Listing 3.6 verwendet. Außerdem wird jeweils getrennt für Infimum und Supremum eine Fehlerschranke berechnet. Der eigentliche Ablauf der Berechnung ist ansonsten äquivalent zum einfachen reellen Skalarprodukt. Listing 3.13 zeigt beispielhaft die Implementierung für den Fall $K = 2$.

```
1  interval p,h;
2  real s_inf, s_sup, r_inf, r_sup;
3  real p_inf, p_sup, q_inf, q_sup;
4  real t;
5
6  TwoProduct(x[lb1], y[lb2], p, s_inf, s_sup);
7
8  err_inf += abs(s_inf);
9  err_sup += abs(s_sup);
10
11 for(int i=2 ; i<=n ; i++) {
12   TwoProduct(x[lb1+i-1], y[lb2+i-1], h, r_inf, r_sup);
13   TwoSum(p, h, p, q_inf, q_sup);
14
15   t = q_inf + r_inf;
16   s_inf += t;
17   err_inf += abs(t);
18
19   t = q_sup + r_sup;
20   s_sup += t;
21   err_sup += abs(t);
22 }
23
24 dotprecision& val_inf = Inf(val);
25 dotprecision& val_sup = Sup(val);
26
27 val_inf += Inf(p);
28 val_inf += s_inf;
29
30 val_sup += Sup(p);
31 val_sup += s_sup;
32
33 real alpha, delta, error;
```

```
34 delta = (n*Epsilon) / (1.0-2*n*Epsilon);
35
36 //Error infimum
37 alpha = (Epsilon*abs(Inf(p))) + (delta*err_inf+3*MinReal/Epsilon);
38 error = alpha / (1.0 - 2*Epsilon);
39 val_inf -= error;
40
41 //Error supremum
42 alpha = (Epsilon*abs(Sup(p))) + (delta*err_sup+3*MinReal/Epsilon);
43 error = alpha / (1.0 - 2*Epsilon);
44 val_sup += error;
```

Listing 3.13: Ausschnitt aus der Funktion *addDot* für Intervallskalarprodukte und $K = 2$

Für Intervallskalarprodukte mit einfacher Genauigkeit ist hingegen ein deutlich anderes Vorgehen als in den anderen Fällen nötig. Da C-XSC für Intervalle grundsätzlich die Infimum-Supremum-Darstellung verwendet (eine Ausnahme bilden einige der optionalen Berechnungsalgorithmen unter Verwendung der BLAS-Bibliothek, welche als Zwischenschritt eine Mittelpunkt-Radius-Darstellung verwenden, siehe dazu Abschnitt 3.2), ist die Berechnung des Intervallskalarproduktes auch bei einfacher Genauigkeit sehr aufwändig, da die Umsetzung der Multiplikationstabelle für solche Intervalle eine ständige Umschaltung des Rundungsmodus erfordert, was sich äußerst negativ auf die Performance auswirkt (siehe Abschnitt 2.5.3). Dieser Effekt kann so stark sein, dass eine solche Implementierung sogar deutlich langsamer als die Nutzung des langen Akkumulators ausfällt.

Um diesen Effekt zu Umgehen, wird für Intervalle eine Vorsortierung des Skalarproduktes verwendet. Dabei wird das Intervallskalarprodukt in zwei reelle Skalarprodukte aufgeteilt, jeweils eines für Infimum und Supremum, wobei mittels der Multiplikationstabelle für Intervalle elementweise geprüft wird, ob jeweils die Ober- oder Untergrenze für die Berechnung verwendet werden muss und die ursprünglichen zwei Intervallvektoren so in vier reelle Punktvektoren aufgeteilt werden. Diese Sortierung ist zwar, bedingt durch die vielen Fallunterscheidungen, ebenfalls deutlich langsamer als ein einfaches Gleitkommaskalarprodukt, aber in der Regel immer noch deutlich schneller als ein ständiger Wechsel des Rundungsmodus.

Für komplexe Intervalle wird eine Kombination der Vorgehensweisen für komplexe und Intervallskalarprodukte genutzt. Dabei werden die Intervallversionen von `TwoSum` und `TwoProduct` genutzt, um, ähnlich wie in Listing 3.12 getrennt nach Real- und Imaginärteil, das Skalarprodukt zu berechnen. Dabei sind vier verschiedene Fehlerschranken zu berechnen, jeweils eine für das Infimum und Supremum von Real- und Imaginärteil.

Benutzung der Skalarproduktalgorithmen

Generell gibt es in C-XSC zwei Möglichkeiten, Skalarprodukte zu berechnen: Zum einen über eine `dotprecision`-Variable, wodurch insbesondere längere Skalarproduktausdrücke ohne Zwischenrundung berechnet werden können, zum anderen über die entsprechenden Operatoren der Matrix- und Vektordatentypen. In beiden Fällen ist die gewünschte Präzision K (im Verhältnis zu *double*-Präzision) für die Skalarproduktberechnungen

frei wählbar, muss allerdings auf andere Art eingestellt werden. Die folgenden Ausführungen gelten für alle Grunddatentypen von C-XSC (also `real`, `interval`, `complex`, `cinterval`).

Zunächst wird die Berechnung über eine `dotprecision`-Variable betrachtet. Eine solche Variable repräsentiert einen langen Akkumulator, zu welchem mittels der `accumulate`-Funktion die Ergebnisse weiterer Skalarprodukte hinzu addiert werden können. Listing 3.1 zeigte bereits beispielhaft die Berechnung des Residuums $r = b - A\tilde{x}$ für die Näherungslösung $\tilde{x}$ eines linearen Gleichungssystems $Ax = b$.

Mit den neuen Skalarproduktalgorithmen ist es nun möglich, jedes über `accumulate` berechnete Skalarprodukt in K-facher *double*-Präzision zu berechnen. Dazu muss nur die Methode `set_dotprec(int)` von `dotprecision` aufgerufen werden, durch welche die gewünschte Präzision K gesetzt wird. Alle folgenden Skalarprodukte, welche mit Hilfe dieser `dotprecision`-Variablen berechnet werden, werden dann in K-facher Genauigkeit berechnet. So können auch innerhalb eines Skalarproduktausdrucks Skalarprodukte mit unterschiedlicher Präzision berechnet werden.

Dabei ist allerdings zu beachten, dass die Summierung stets mit maximaler Genauigkeit erfolgt. Für obiges Beispiel bedeutet das bei Wahl einer Genauigkeit von $K = 2$ also z.B., dass zwar die Skalarprodukte bei der Berechnung von $A\tilde{x}$ mit doppelter Genauigkeit berechnet werden, die Berechnung der Differenz dann aber mit maximaler Genauigkeit durchgeführt wird. Bei einer Berechnung wie in obigem Beispiel, bei der es in der Regel zu starker Auslöschung kommt, hat dies den Vorteil, dass die K-fache Länge der Ergebnisse der Skalarprodukte im Akkumulator voll genutzt wird. Für eine mit reiner Gleitkommarechnung berechnete Näherungslösung erhält man so bereits mit einer Präzision $K = 2$ eine sehr genaue Annäherung an das tatsächliche Residuum und spart gegenüber der Benutzung des Akkumulators (also Genauigkeit $K = 0$, welche aus Kompatibilitätsgründen mit bestehender Software auch weiterhin die Standardeinstellung ist) deutlich an Laufzeit ein.

Die zweite Möglichkeit zur Nutzung der neuen Skalarproduktalgorithmen besteht über die Verwendung der entsprechenden Operatoren der Matrix- und Vektordatentypen. Hier werden bei Aufruf der Operatoren für das Vektor-Vektor-Produkt, das Matrix-Vektor-Produkt sowie das Matrix-Matrix-Produkt ein oder mehrere Skalarprodukte berechnet, deren Ergebnisse standardmäßig über den langen Akkumulator bestimmt werden. Um die Präzision der Skalarproduktberechnungen für diese Operatoren einzustellen, wird die globale Variable `opdotprec` eingeführt. Mit Hilfe dieser Variablen kann die Präzision für alle Skalarproduktberechnungen innerhalb von Operatoren eingestellt werden. Listing 3.14 zeigt eine entsprechende Berechnung für das Beispiel aus Listing 3.1.

```
1 rmatrix A;
2 rvector r,b,xa;
3
4 //... Matrizen und Vektoren fuellen ...
5
6 opdotprec = 2;
```

```
7  r = b - A*xa;
```

Listing 3.14: Beispiel für die Berechnung des Residuums eines linearen Gleichungssystems in Genauigkeit $K = 2$ mittels Operatoren

Eine Berechnung auf diese Art benötigt offensichtlich deutlich weniger Quellcode, allerdings ist zu Bedenken, dass einige weitere wichtige Unterschiede zwischen beiden Berechnungsarten bestehen:

- Bei der Nutzung von Operatoren werden alle Zwischenergebnisse in Arbeitsgenauigkeit gerundet, die K-fache Länge der Zwischenergebnisse bei der Berechnung der Skalarprodukte wird hier also nicht ausgenutzt.

- Will man eine Intervalleinschließung für einen Punktausdruck berechnen, so ist dies bei Verwendung von Operatoren nur möglich, indem man zu Intervallberechnungen übergeht (in obigem Beispiel könnte man also einen Cast der Matrix A zu einer Intervallmatrix (also zum Datentyp `imatrix`) vornehmen), während bei direkter Verwendung einer `dotprecision`-Variable das Endergebnis einfach nach außen gerundet werden kann. In diesem Fall ist die Operatorberechnung dadurch auch langsamer.

- Im Allgemeinen ist die Verwendung von Operatoren hingegen etwas schneller und in der Verwendung deutlich einfacher.

Zumeist ist die Verwendung der Operatoren ausreichend, will man allerdings genauere Ergebnisse oder mehr Kontrolle über die Berechnung, so ist die explizite Verwendung einer `dotprecision`-Variablen vorzuziehen. Auch bei der Parallelisierung, z.B. mittels OpenMP, kann die Verwendung der `dotprecision`-Typen mit expliziten Aufrufen von `accumulate` vorteilhaft sein, da dann die Parallelisierung an den einzelnen for-Schleifen der Berechnung ansetzen kann. Alternativ sind aber auch einige Operatoren in C-XSC bereits mit OpenMP parallelisiert, siehe Abschnitt 3.5.1.

3.1.4. Kompilierung

In diesem Abschnitt sollen einige kurze Anmerkungen zur Kompilierung der neuen Skalarproduktalgorithmen gemacht werden. Die bei der Kompilierung verwendeten Compileroptionen können großen Einfluss sowohl auf die Laufzeit als auch auf die Korrektheit der Ergebnisse haben. Bei der Kompilierung ist daher große Sorgfalt nötig, um keine falschen Ergebnisse zu erhalten.

Für die Laufzeit von entscheidender Bedeutung bei Verwendung des DotK-Algorithmus und auch der normalen Gleitkommapräzision $K = 1$ ist die Aktivierung von Inlining durch den Compiler (Details hierzu finden sich in Abschnitt 2.5.2). Durch die vielen Funktionsaufrufe, insbesondere der Funktionen für die fehlerfreien Transformationen, entsteht sonst ein deutlicher Geschwindigkeitsverlust. Auch bei der Kompilierung der C-XSC Bibliothek selbst sollte Inlining aktiviert sein. Im Normalfall sollte die volle Optimierungsstufe über die Option `-O3` sowohl bei der Erstellung der Bibliothek als auch

bei der Kompilierung der eigenen Programme verwendet werden, sofern keine anderen Überlegungen, insbesondere in Bezug auf die Korrektheit des gesetzten Rundungsmodus (siehe Abschnitt 2.5.3), dagegen sprechen.

Für die Korrektheit der Ergebnisse entscheidend ist, dass die Berechnungen dem IEEE-754 *double*-Format entsprechen. Die Verwendung von *excess precision* führt bei den Algorithmen zur Berechnung der fehlerfreien Transformationen zu falschen Ergebnissen. Die Algorithmen lassen sich auch nicht einfach an das längere 80 Bit Format anpassen, da bei der Kompilierung (u.a. auch abhängig von den verwendeten Optimierungsoptionen) nicht zwingend Code entsteht, welcher die in den Registern gespeicherten Zwischenergebnisse weiterverwendet, so dass es auch zu Zwischenspeicherungen im Hauptspeicher kommen kann, durch welche die Werte wieder in das übliche *double*-Format gerundet werden würden.

Auf nahezu allen bedeutenden Plattformen gibt es allerdings die Möglichkeit, die fehlerfreien Transformationen korrekt zu berechnen, entweder weil die Register ohnehin die übliche Breite von 64 Bit aufweisen, oder weil Alternativen zur Verfügung stehen. Auf x86-Rechnern stehen hierfür die SSE-Register (*Streaming SIMD Extensions*) zur Verfügung. Die SSE-Befehlssatzerweiterungen verwenden 128-Bit breite Register, welche zwei Gleitkommazahlen im *double*-Format aufnehmen können. Die Berechnungen mit Hilfe der SSE-Einheit sind in der Regel nicht nur schneller als die Nutzung der alten x87-Befehle, sondern nutzen auch das standardkonforme 64-Bit *double*-Format statt des erweiterten 80-Bit-Formates.

Durch Nutzung dieser Register für alle Gleitkommaberechnungen, welche sich über eine entsprechende Compileroption aktivieren lässt bzw. bei den meisten 64-Bit Compilern ohnehin standardmäßig aktiviert ist, liefern die fehlerfreien Transformationen also auch auf x86 Rechnern korrekte Ergebnisse ohne jegliche Geschwindigkeitsverluste. Nötig ist dazu nur die Unterstützung des SSE2-Befehlssatzes, welcher von allen Intel- und AMD-Prozessoren seit dem Pentium III unterstützt wird.

Steht keine solche Option für die genutzte Plattform zur Verfügung, so kann der Compiler in der Regel angewiesen werden, grundsätzlich mit Gleitkommazahlen im 64-Bit-*double*-Format zu rechnen. Dies führt in jedem Fall zu richtigen Ergebnissen, allerdings kann es auf einigen Plattformen nötig sein, hierfür Geschwindigkeitseinbußen in Kauf zu nehmen (z.B. da die Zwischenergebnisse immer in den Hauptspeicher zurückgeschrieben werden müssen). Für solche Plattformen kann es daher in Zukunft vorteilhaft sein, spezielle Assemblerimplementierungen der fehlerfreien Transformationen zu entwickeln.

Bei der Kompilierung von C-XSC wird für alle unterstützten Plattformen automatisch eine entsprechende Option gewählt, um korrekte Ergebnisse zu garantieren. Die DotK-Routinen werden grundsätzlich direkt in die Bibliotheksdatei kompiliert, d.h. sie werden nicht in einem Header definiert. Dies führt zu einem gewissen Laufzeitverlust, da ein evtl. mögliches Inlining der `addDot`-Funktion verhindert wird, ist aber nötig, damit ein Benutzer bei Kompilierung seines Programms nicht gezwungen ist, auf die korrekten Compileroptionen zu achten, was in vielen Fällen zu falscher Kompilierung und damit zu falschen Ergebnissen führen würde.

3.1.5. Numerische Ergebnisse und Zeitvergleiche

In diesem Abschnitt werden die verschiedenen beschriebenen Skalarproduktalgorithmen diversen Tests zur Bestimmung von Laufzeitverhalten und numerischer Genauigkeit unterzogen. Dazu ist es zunächst nötig, entsprechende Testdaten zur Verfügung zu haben. Hierzu werden zwei verschiedene Algorithmen zur Erzeugung von Skalarprodukten mit einer bestimmten Kondition verwendet.

Zum einen wird der Algorithmus 6.1 aus [106], in der Folge mit *GenDot* bezeichnet, verwendet. Dieser erzeugt Skalarprodukte einer vorgegebenen Dimension n mit vorgegebener Kondition, bei welchen die Vektoren keinem offensichtlichen Muster folgen. Nachteil ist allerdings, dass die tatsächliche Kondition des erzeugten Skalarprodukts häufig sehr stark von der eigentlich gewünschten Kondition abweicht. Außerdem dauert die Erzeugung von Skalarprodukten hoher Dimension durch die im Algorithmus notwendigen Berechnungen von exakten Skalarprodukten (in der C-XSC Implementierung wird dazu der lange Akkumulator verwendet) sehr lange. Für Details siehe Ogita et al. [106].

Zum anderen wird ein alternativer Algorithmus nach Oishi et al. [108], in der Folge als *GenDot2* bezeichnet, verwendet. Die Kondition der von diesem Algorithmus erzeugten Skalarprodukte liegt im Allgemeinen deutlich näher an der vorgegebenen Kondition als dies beim Algorithmus *GenDot* der Fall ist. Außerdem ist er deutlich schneller und eignet sich daher auch für die Erzeugung von Skalarprodukten hoher Dimension. Nachteilig ist, dass die erzeugten Vektoren stets eine recht strikt vorgegebene Struktur aufweisen. Weitere Erläuterungen sowie Anpassungen für komplexe und Intervallskalarprodukte werden bei Zimmer [146] beschrieben.

Zunächst soll nun die erreichte Genauigkeit der neuen Skalarproduktalgorithmen getestet werden. Dazu werden mit den Algorithmen *GenDot* und *GenDot2* 1000 reelle Skalarprodukte $x^T y$ der Dimension $n = 1000$ mit Konditionszahlen zwischen 1 und 10^{100} generiert. Danach werden Einschließungen dieser Skalarprodukte mit Genauigkeiten von $K = 1, \ldots, 7$ berechnet. Abbildung 3.2 zeigt die relative Breite

$$\frac{diam(\texttt{res})}{x^T y}$$

der berechneten Einschließung `res` für die mit *GenDot2* generierten Skalarprodukte, Abbildung 3.3 den relativen Fehler des Ergebnisses (also von mid(`res`)) für die mit *GenDot* generierten Skalarprodukte, welcher sich über

$$\frac{\texttt{res} - x^T y}{x^T y}$$

berechnen lässt (die Berechnung des Zählers sowie des Nenners erfolgt dabei exakt mittels des Akkumulators). Relative Fehler größer als 2 werden dabei auf 2 gesetzt, da dann nahezu keine Informationen des korrekten Ergebnisses mehr vorhanden sind. Der Fall $K = 0$ wird hier nicht getestet, da die Implementierung des langen Akkumulators in C-XSC mittlerweile sehr ausgereift ist und daher verlässliche Ergebnisse mit einer Genauigkeit von 1 *ulp* berechnet werden. Weiterhin werden hier nur reelle Skalarprodukte

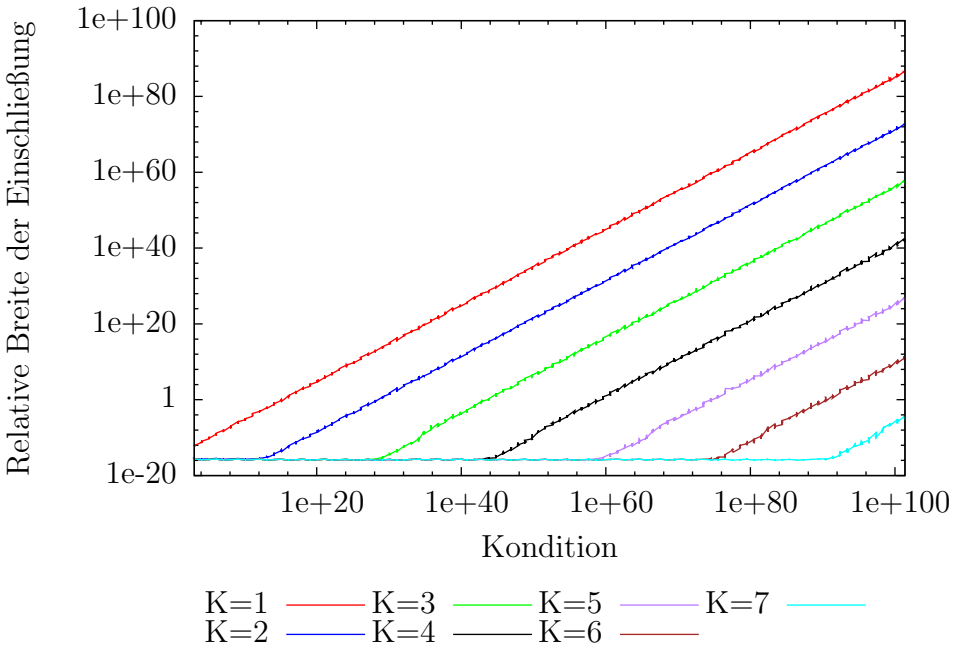


Abbildung 3.2.: Relative Breite der berechneten Einschließungen bei unterschiedlichen Konditionen (Skalarprodukte generiert mit *GenDot2*)

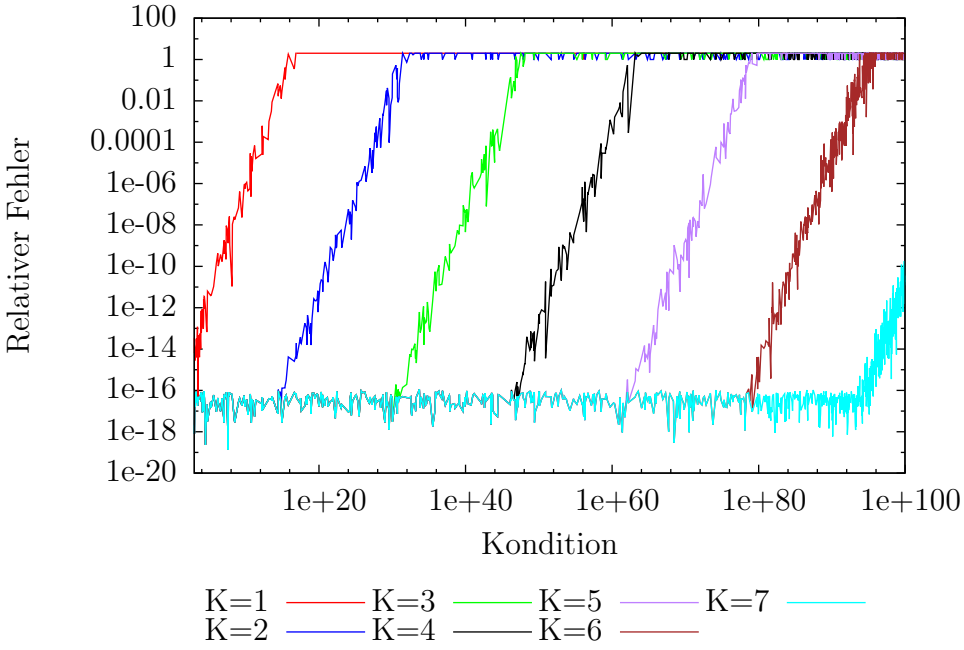


Abbildung 3.3.: Relativer Fehler des berechneten Ergebnisses bei unterschiedlichen Konditionen (Skalarprodukte generiert mit *GenDot2*)

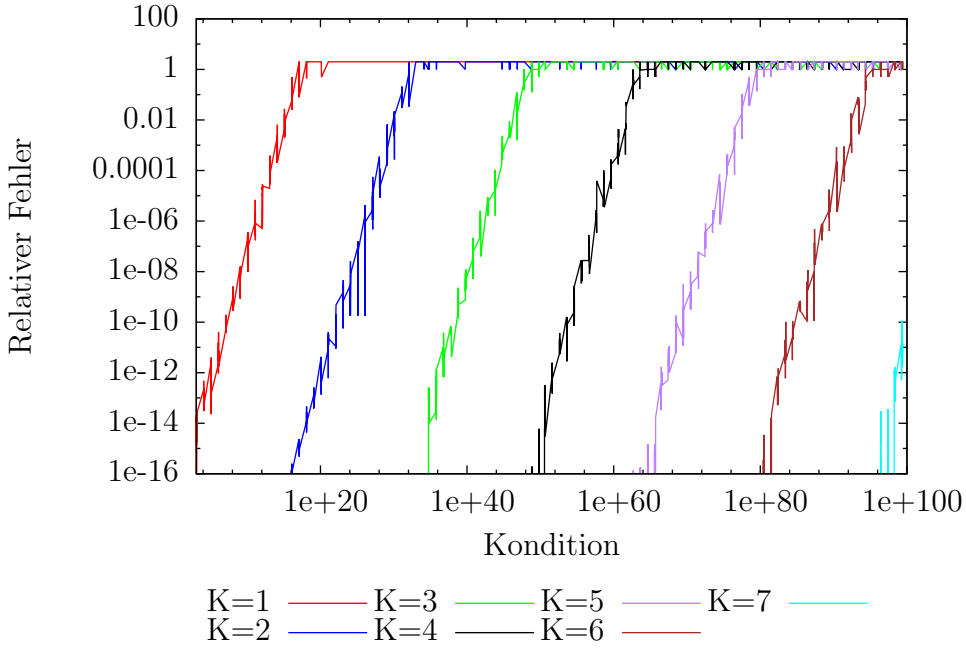


Abbildung 3.4.: Relativer Fehler des berechneten Ergebnisses bei unterschiedlichen Konditionen (Skalarprodukte generiert mit *GenDot*)

betrachtet. Da die Algorithmen für komplexe und Intervallskalarprodukte aber letztlich auf dem reellen Algorithmus basieren, erhält man dort analoge Ergebnisse.

Wie in den Abbildungen 3.2 und 3.3 zu sehen ist, entspricht das Verhalten in der Tat einer Berechnung mit K-facher Präzision. Die Ergebnisse bleiben jeweils bis zu einer Kondition von etwa $10^{(K-1)15}$ so exakt wie möglich (das Endergebnis muss in jedem Fall in *double*-Genauigkeit gerundet werden) und verlieren danach nahezu linear an Genauigkeit. Bei einer Kondition von etwa $10^{K \cdot 15}$ ist dann bei Berechnung mit K-facher Präzision keine Stelle des Ergebnisses mehr korrekt. Die Breite der berechneten Einschließung wächst ab diesem Punkt annähernd linear. Es ist aber zu bemerken, dass die berechnete Einschließung trotzdem immer das korrekte Ergebnis enthält.

Abbildung 3.4 zeigt den relativen Fehler für die mit *GenDot* berechneten Skalarprodukte. Die Ergebnisse entsprechen im Wesentlichen den vorherigen. Allerdings ist hier der Fehler teilweise sogar 0, d.h. das exakte Ergebnis wird berechnet. Da bei Verwendung von *GenDot* das exakte Ergebnis für die gewählte Kondition *cnd* immer cnd^{-1} ist, ist es für die Testfälle, in denen die Kondition eine Zweierpotenz ist, exakt als Gleitkommazahl darstellbar, was bei den mit *GenDot* generierten Skalarprodukten in der Regel nicht der Fall ist.

Als nächstes sollen die erreichten Laufzeiten verglichen werden. Dazu wird zunächst eine Zeitmessung mit Skalarprodukten der Dimension $n = 100000$ für alle Grunddatentypen durchgeführt. Die Ergebnisse werden in Abbildung 3.5 dargestellt, die genauen Werte lassen sich Tabelle 3.1 entnehmen. Alle Skalarprodukte für diese und die folgenden Zeitmessungen wurden mit dem Algorithmus *GenDot2* mit einer gewählten Kondition

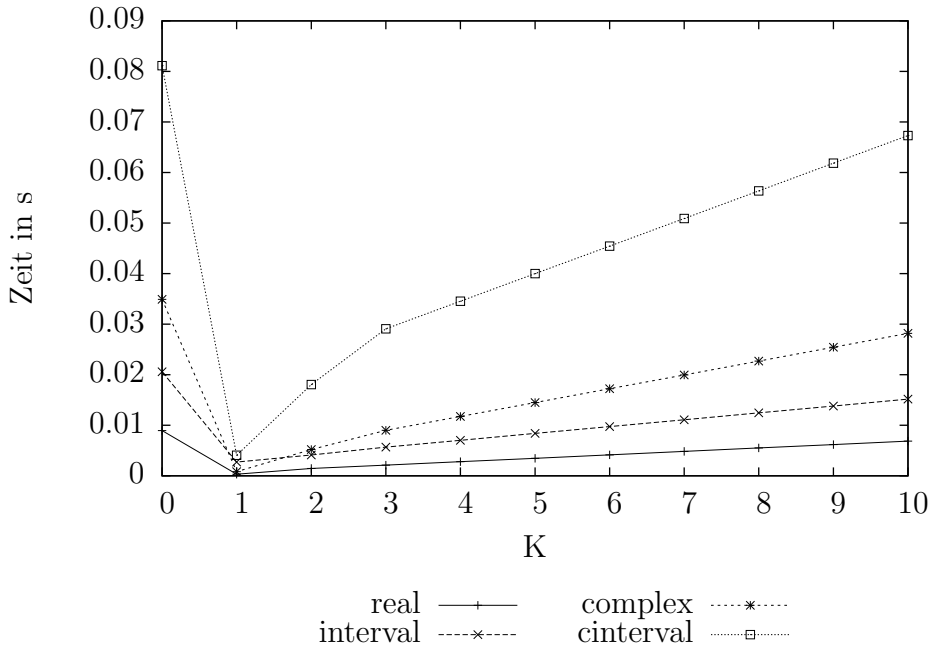


Abbildung 3.5.: Zeitmessung für Skalarprodukte der Dimension $n = 100000$

von 10^{20} generiert (die Kondition hat keinerlei Auswirkung auf die Laufzeit). Jede Messung wurde dabei 100 mal wiederholt, um eventuelle Schwankungen auszugleichen.

Die Zeitmessungen zeigen für alle Datentypen ein ähnliches Verhalten. Während die Berechnung mit dem langen Akkumulator $K = 0$ deutlich länger als mit den übrigen getesteten Präzisionen dauert, ist reine Gleitkommarechnung mit $K = 1$ am schnellsten, auch bei reellen und komplexen Intervallen, bei denen, wie in Abschnitt 3.1.3 erläutert, aufgrund der von C-XSC verwendeten Infimum-Supremum-Darstellung von Intervallen vor der eigentlichen Berechnung ein Sortierungsschritt erfolgt. Die optimierte Version des *DotK*-Algorithmus für $K = 2$ ist im Schnitt etwa 4-5 mal langsamer als reine Gleitkommarechnung. Danach folgt ein größerer Sprung zu $K = 3$, welcher durch die Optimierungen beim Algorithmus für doppelte Genauigkeit verursacht wird. Ab einer Präzision von $K = 3$ wächst der Zeitaufwand linear mit der Präzision, da hier bei jeder Stufe ein weiterer Schleifendurchlauf in der Berechnung hinzukommt (siehe Abschnitt 3.1.2). Festzuhalten ist, dass selbst für eine Präzision von $K = 10$ die Berechnung mit dem *DotK*-Algorithmus immer noch schneller ist, als die Verwendung des langen Akkumulators ($K = 0$).

Um die Laufzeiten für die verschiedenen Datentypen und Präzisionen in Relation zu setzen, wird in Abbildung 3.6 dargestellt, wie sich die Laufzeit für die jeweilige Berechnung in Relation zum reellen Skalarprodukt mit Präzision $K = 1$ verschlechtert (d.h. diese Berechnung wird mit 1 gewichtet). Wie zu erkennen ist, sind erwartungsgemäß für alle Präzisionen reelle Punkt-Skalarprodukte am schnellsten. Reelle Intervallskalarprodukte benötigen etwa doppelt so viel Zeit, mit Ausnahme von Präzision $K = 1$, bei welcher Intervallskalarprodukte durch die nötige Vorsortierung deutlich langsamer sind.

Genauigkeit	**real**	**interval**	**complex**	**cinterval**
$K = 0$	0.008958	0.020547	0.035030	0.081186
$K = 1$	0.000325	0.002731	0.000820	0.003914
$K = 2$	0.001459	0.004113	0.005210	0.018058
$K = 3$	0.002132	0.005662	0.009059	0.028960
$K = 4$	0.002810	0.007022	0.011772	0.034407
$K = 5$	0.003488	0.008379	0.014516	0.039911
$K = 6$	0.004165	0.009733	0.017250	0.045399
$K = 7$	0.004844	0.011091	0.019990	0.050897
$K = 8$	0.005523	0.012449	0.022732	0.056385
$K = 9$	0.006199	0.013807	0.025467	0.061846
$K = 10$	0.006881	0.015168	0.028216	0.067331

Tabelle 3.1.: Zeitmessungen in Sekunden für Skalarprodukte der Dimension $n = 100000$ mit verschiedenen Genauigkeiten

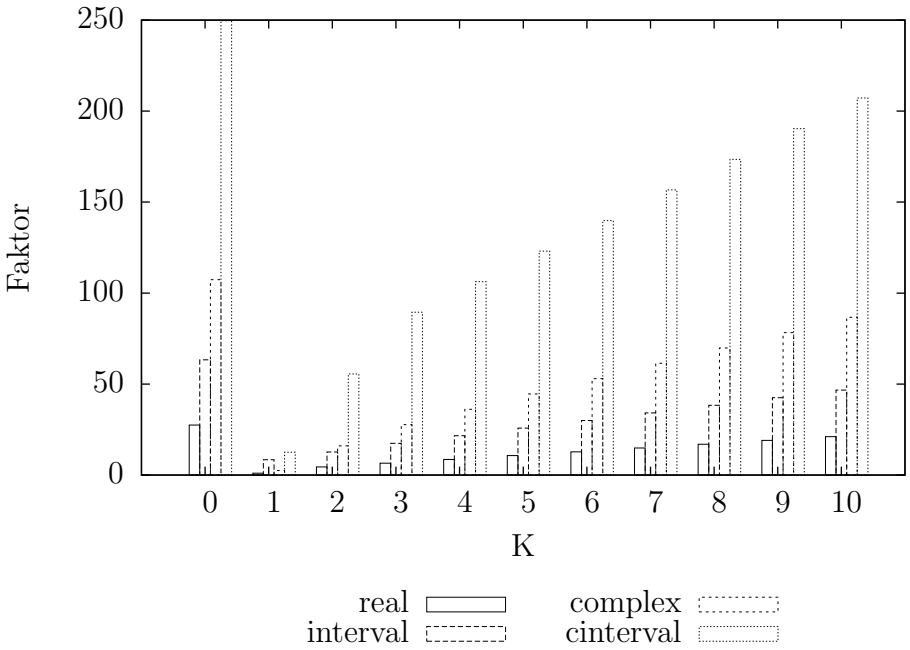


Abbildung 3.6.: Laufzeit relativ zu reellem Skalarprodukt bei Verwendung von `dotprecision`, Dimension $n = 100000$

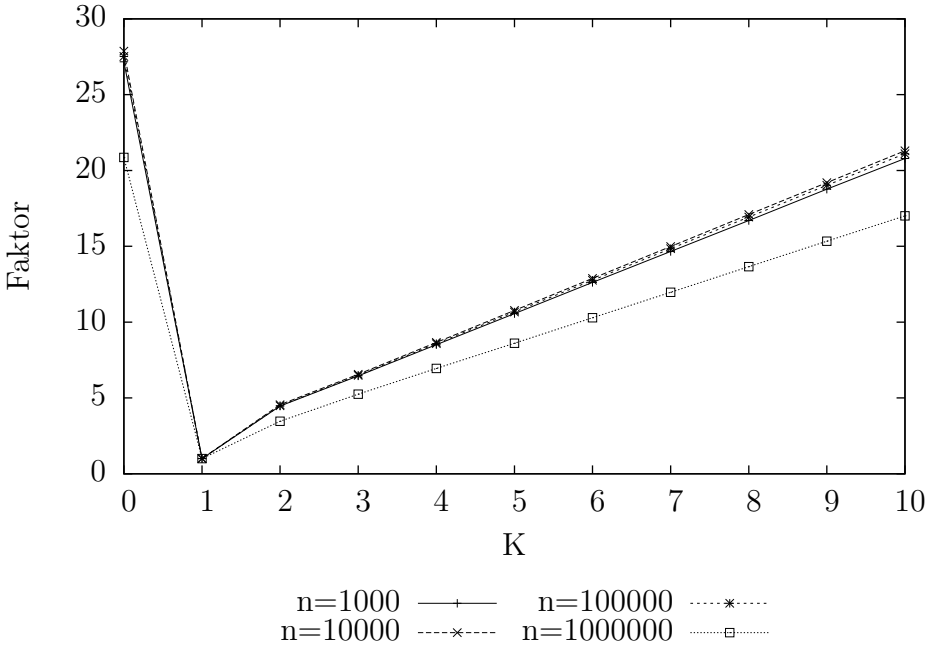


Abbildung 3.7.: Laufzeit für reelle Skalarprodukte bei verschiedenen Präzisionen relativ zu $K = 1$

Als nächstes folgen komplexe Punktskalarprodukte, welche im Allgemeinen langsamer als reelle Punkt- und Intervallskalarprodukte sind. Ausnahme ist auch hier wieder die Präzision $K = 1$, bei welcher Punktberechnungen auch im Komplexen deutlich schneller als Intervallskalarprodukte sind. Deutlich am langsamsten sind, wie zu erwarten, die komplexen Intervallskalarprodukte, bei denen ein Skalarprodukt mit Präzision $K = 0$ etwa 250 mal so lange dauert wie ein einfaches reelles Skalarprodukt.

Eine weitere interessante Frage ist, wie sich die Dimension des berechneten Skalarprodukts auf die eben beschriebenen Laufzeitverhältnisse zwischen den verschiedenen Präzisionen auswirkt. Um diese Frage zu klären, werden entsprechende Messungen mit reellen Skalarprodukten (die Aussagen gelten weitgehend analog für die anderen Datentypen) der Dimension $n = 1000, 10000, 100000, 1000000$ herangezogen. Die erreichten Zeiten werden in Abbildung 3.7 dargestellt, wobei für jede Dimension das mit Präzision $K = 1$ berechnete Skalarprodukt mit 1 gewichtet wird.

Wie zu sehen ist, ist das Laufzeitverhältnis zwischen den verschiedenen Präzisionen im Wesentlichen unabhängig von der Dimension des Skalarproduktes. Allerdings zeigt sich für die Dimension $n = 1000000$, dass die *DotK*-Berechnungen ebenso wie die Berechnungen mit dem langen Akkumulator etwas schneller in Relation zum einfachen Skalarprodukt mit $K = 1$ sind als bei kleineren Dimensionen. Dies liegt nicht etwa daran, dass diese Algorithmen hier im Vergleich schneller wären, sondern interessanterweise daran, dass die Berechnung für $K = 1$ hier unverhältnismäßig viel langsamer wird im Vergleich zu den anderen Dimensionen (beim Sprung von $n = 100000$ auf $n = 1000000$ steigt die nötige Zeit nicht um einen Faktor 10, sondern um einen Faktor von ca. 12). Vermutlich

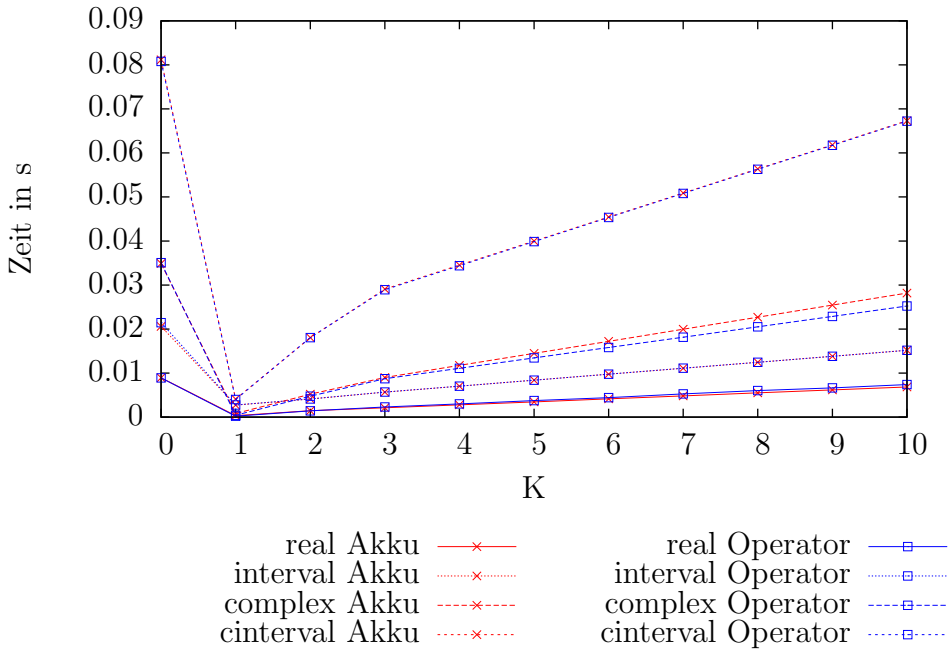


Abbildung 3.8.: Laufzeitvergleich zwischen Verwendung von `dotprecision` und Verwendung von Operatoren, Dimension $n = 100000$

wird hier verstärkter Zugriff auf den Hauptspeicher nötig, da bei dieser Dimension der Cache nicht mehr so gut ausgenutzt werden kann.

Zum Abschluss soll nun noch die Berechnung der Skalarprodukte mittels der jeweiligen Operatoren mit der Berechnung mittels einer `dotprecision`-Variable verglichen werden. Dazu werden Skalarprodukte mit Dimension $n = 100000$ für alle Datentypen mit Präzision $K = 0, \ldots, 10$ berechnet. Die Ergebnisse lassen sich Abbildung 3.8 entnehmen.

Wie zu erwarten, liegen die Ergebnisse im Allgemeinen gleich auf. Für Punktskalarprodukte (also für die Grunddatentypen `real` und `complex`) ist die Berechnung mittels Operator allerdings etwas schneller. Grund hierfür ist, dass bei diesen Operatoren, wie in Abschnitt 3.1.3 beschrieben, keine Fehlerschranke benötigt wird und die dafür normalerweise nötigen Berechnungen daher auch nicht durchgeführt werden. Eine Ausnahme bildet hier der Fall $K = 0$, da eine Berechnung mittels des langen Akkumulators grundsätzlich maximal genau erfolgt und daher auch in beiden Fällen die gleiche Zeit in Anspruch nimmt.

Insgesamt zeigen die Ergebnisse, dass die neuen Skalarproduktalgorithmen Ergebnisse in K-facher *double*-Präzision liefern und dabei bis zu einer Präzision von $K = 10$ immer noch schneller als die Berechnung mittels des langen Akkumulators sind. In vielen praktischen Anwendungsfällen, in denen schon eine Berechnung in doppelter Genauigkeit ausreicht, ist die Nutzung des *DotK*-Algorithmus also eine attraktive Alternative. Weiterhin besteht über die Wahl von $K = 1$ nun auch die Möglichkeit, in C-XSC Skalarprodukte mit reiner Gleitkommarechnung durchzuführen, was z.B. für die in vielen Algorithmen aus der Verifikationsnumerik zunächst notwendige Berechnung einer Nä-

herungslösung eine wichtige Ergänzung ist. Weiterhin wird hierdurch erst die sinnvolle Nutzung von BLAS-Routinen innerhalb der C-XSC-Bibliothek ermöglicht.

3.2. Nutzung von BLAS und LAPACK

In diesem Abschnitt wird die optionale Nutzung der hochoptimierten Bibliotheken BLAS und LAPACK (siehe Abschnitt 2.4) für einige Matrix- und Vektorberechnungen in C-XSC erläutert. Dazu wird zunächst auf passende Algorithmen zur Nutzung der BLAS-Bibliothek im Rahmen der Intervallrechnung, also auf Algorithmen zur Berechnung einer verlässlichen Einschließung mit diesen eigentlich auf einfache Gleitkommarechnung ausgelegten Routinen, eingegangen. Danach wird die konkrete Implementierung und die Benutzung der BLAS-Bibliothek innerhalb der C-XSC Bibliothek besprochen. Ebenso wird eine Schnittstelle zwischen C-XSC und einigen LAPACK-Funktionen beschrieben, welche es ermöglicht, diese LAPACK-Routinen (d.h. normale Gleitkommaalgorithmen, welche nur Näherungsergebnisse berechnen, dies allerdings sehr schnell) direkt mit C-XSC Datentypen aufzurufen. Auch werden einige Tests und Zeitmessungen zur Benutzung von BLAS und LAPACK mit C-XSC angegeben.

3.2.1. Algorithmen

Die hochoptimierten Routinen aus der BLAS-Bibliothek berechnen die jeweilige Problemstellung, also z.B. ein Matrix-Matrix-Produkt, mittels rundungsbehafteter Gleitkommarechnung. Um diese Routinen im Rahmen der Verifikationsnumerik nutzen zu können, müssen daher entsprechende Algorithmen verwendet werden, welche mit Hilfe des Rundungsmodus des Prozessors Einschließungen des korrekten Ergebnisses berechnen können. Im Folgenden werden entsprechende Algorithmen für das Produkt zweier Matrizen vorgestellt. Das Vorgehen für Matrix-Vektor- und Vektor-Vektor-Produkte lässt sich leicht aus diesen ableiten.

Die Algorithmen in diesem Abschnitt basieren auf den von Rump in Intlab [130] verwendeten Algorithmen. Dabei wird eine Funktion `setround` zum Wechseln des Rundungsmodus benutzt. Die Algorithmen werden in einem an C-XSC angelehnten Pseudocode angegeben. Alle in den Algorithmen vorkommenden Matrix-Matrix-Produkte werden gleitkommamäßig berechnet, d.h. sie können auch mittels Aufruf einer entsprechenden BLAS-Routine berechnet werden.

Zunächst wird der einfachste Fall, das Produkt zweier reeller Punktmatrizen, betrachtet (Algorithmus 3.12). Hierbei genügt es, das Matrix-Matrix-Produkt zweimal zu berechnen, einmal mit Rundung nach unten zur Berechnung des Infimums der Einschließung und einmal mit Rundung nach oben zur Berechnung des Supremums. Da hierbei jede Operation entsprechend gerundet wird, erhält man stets eine korrekte Einschließung, unabhängig von der tatsächlichen Reihenfolge der Berechnungen innerhalb der Matrix-Matrix-Produkte und auch bei Auftreten von Über- oder Unterlauf.

Als nächstes wird das Produkt einer reellen Punkt- und einer reellen Intervallmatrix betrachtet (Algorithmus 3.13). Um Berechnungen mit Intervallmatrizen effektiv auf ein-

Eingabe : Zwei reelle Matrizen A und B
Ausgabe : Eine Intervalleinschließung $\boldsymbol{C}$ des Produktes AB

```
setround(-1);
SetInf(C, A*B);
setround(1);
SetSup(C, A*B);
setround(0);
```

Algorithmus 3.12: Berechnung der Einschließung des Produktes zweier reeller Matrizen

fache Gleitkommaprodukte von Punktmatrizen zurückzuführen, ist es nötig, zu einer Mittelpunkt-Radius Darstellung überzugehen. Wie in Algorithmus 3.13 zu sehen, sind durch dieses Vorgehen nur drei Matrix-Matrix-Produkte sowie einige $\mathcal{O}(n^2)$ Operationen nötig.

Eingabe : Eine reelle Matrix A und eine Intervallmatrix B
Ausgabe : Eine Intervalleinschließung $\boldsymbol{C}$ des Produktes AB

```
setround(1);
Bmid = Inf(B) + 0.5*(Sup(B)-Inf(B));
Brad = Bmid - Inf(B);
setround(-1);
C1 = A*Bmid;
setround(1);
C2 = A*Bmid;
Cmid = C1 + 0.5*(C2-C1);
Crad = (Cmid-C1) + abs(A) * Brad;
setround(-1);
SetInf(C, Cmid-Crad);
setround(1);
SetSup(C, Cmid+Crad);
setround(0);
```

Algorithmus 3.13: Berechnung der Einschließung des Produktes einer reellen und einer Intervallmatrix

Bei Verwendung der Infimum-Supremum Darstellung wäre es nötig, zunächst alle Kombinationen von Untergrenze und Obergrenze jeweils einmal mit Rundung nach unten und einmal mit Rundung nach oben zu berechnen und danach durch einen elementweisen Vergleich jeweils das größte und kleinste Element pro Matrixeintrag zu finden. Beim Produkt einer reellen Punkt- und einer reellen Intervallmatrix wären dies also vier statt drei Matrix-Matrix-Produkte (für das Produkt zweier reeller Intervallmatrizen wären sogar acht Matrix-Matrix-Produkte notwendig). Weiterhin ist der abschließende Vergleich durch die vielen Fallunterscheidungen ebenfalls vergleichsweise zeitaufwändig, auch wenn es sich hier um eine $\mathcal{O}(n^2)$ Operation handelt.

Die Mittelpunkt-Radius-Darstellung bietet hier also enorme Geschwindigkeitsvorteile. Ein Nachteil ist, dass dadurch eine Überschätzung um einen Faktor von maximal 1.5 möglich ist (siehe Abschnitt 2.1.2). Da die Nutzung der BLAS-Routinen in C-XSC aber optional ist und insbesondere auf maximale Geschwindigkeit optimiert sein soll, wird die Überschätzung hier in Kauf genommen.

Beim Produkt zweier reeller Intervallmatrizen (Algorithmus 3.14) ist der Vorteil durch Verwendung der Mittelpunkt-Radius-Darstellung nochmals deutlich größer. Die Laufzeit wird hier von den vier notwendigen Matrix-Matrix-Produkten dominiert, im Gegensatz zu acht Matrix-Matrix-Produkten bei Verwendung der Infimum-Supremum-Darstellung. Berücksichtigt man die zusätzlich notwendigen, relativ aufwändigen Vergleiche, so lässt sich feststellen, dass die Berechnung mittels Mittelpunkt-Radius-Darstellung hier mehr als doppelt so schnell ist.

Eingabe : Zwei reelle Intervallmatrizen A und B
Ausgabe : Eine Intervalleinschließung $\boldsymbol{C}$ des Produktes AB

```
setround(1);
Amid = Inf(A) + 0.5*(Sup(A)-Inf(A));
Arad = Amid - Inf(A);
Bmid = Inf(B) + 0.5*(Sup(B)-Inf(B));
Brad = Bmid - Inf(B);
setround(-1);
C1 = Amid*Bmid;
setround(1);
C2 = Amid*Bmid;
Cmid = C1 + 0.5*(C2-C1);
Crad = (Cmid-C1) + Arad * ( abs(Bmid) + Brad ) + abs(Amid) * Brad;
setround(-1);
SetInf(C, Cmid-Crad);
setround(1);
SetSup(C, Cmid+Crad);
setround(0);
```

Algorithmus 3.14: Berechnung der Einschließung des Produktes zweier reeller Intervallmatrizen

Im Komplexen ist das Vorgehen sehr ähnlich zum reellen Fall. Algorithmus 3.15 zeigt zunächst, wie eine Einschließung des Produktes zweier komplexer Punktmatrizen berechnet wird. Wie im Reellen wird hier die entsprechende Berechnung zweimal durchgeführt, einmal mit Rundung nach oben und einmal mit Rundung nach unten. Dadurch ist auch hier, unabhängig von der tatsächlichen Reihenfolge der Berechnungen durch die BLAS-Bibliothek innerhalb der einzelnen Matrix-Matrix-Produkte und einem eventuell auftretenden Unter- oder Überlauf, garantiert, dass eine korrekte Einschließung berechnet wird.

Bei Verwendung komplexer Intervallmatrizen wird, anders als in Intlab, nicht Kreisscheibenarithmetik benutzt. Stattdessen werden Real- und Imaginärteil jeweils in reelle

Eingabe : Zwei komplexe Matrizen A und B
Ausgabe : Eine Intervalleinschließung $\boldsymbol{C}$ des Produktes AB

```
setround(-1);
SetRe(C1, Re(A)*Re(B) + (-Im(A))*Im(B));
SetIm(C1, Re(A)*Im(B) + Im(A)*Re(B));
setround(1);
SetRe(C2, Re(A)*Re(B) + (-Im(A))*Im(B));
SetIm(C2, Re(A)*Im(B) + Im(A)*Re(B));
setround(0);
SetInf(C, C1);
SetSup(C, C2);
```

Algorithmus 3.15: Berechnung der Einschließung des Produktes zweier komplexer Matrizen

Mittelpunkt-Radius-Darstellung umgewandelt und es wird entsprechend dem reellen Fall vorgegangen. Da C-XSC für komplexe Intervalle Rechteckarithmetik verwendet, würde durch eine Umwandlung in Kreisscheibenarithmetik für die Berechnung mit anschließender Konversion zurück in ein einschließendes Rechteck in der Regel eine große zusätzliche Überschätzung eingeführt werden. Zunächst müsste ein Kreis in der komplexen Ebene berechnet werden, welcher das betrachtete Rechteckintervall einschließt und nach der Berechnung wiederum ein Rechteck, welches den Ergebniskreis einschließt. Die dadurch möglichen Auswirkungen zeigt folgendes Beispiel:

Beispiel 3.1. *Gegeben sei das Rechteck* $z = [0, 10a] + i[0, a]$, $a \in \mathbb{R}$, *in der komplexen Ebene. Es soll von einer Kreisscheibe eingeschlossen werden, welche dann mit* 1 *multipliziert wird. Anschließend wird wieder zur Rechteckdarstellung zurückgekehrt, der Ergebniskreis wird also engstmöglich von einem Rechteck eingeschlossen. Der engstmögliche Kreis, welcher* z *einschließt, hat den Mittelpunkt* $m = (5a, a/2)$ *und den Radius*

$$r = \sqrt{25a^2 + \frac{a^2}{4}}.$$

Eine Multiplikation mit 1 *ändert den Kreis nicht. Nach der Berechnung erfolgt die Umwandlung zurück in ein Rechteck. Das engste einschließende Rechteck des Kreises hat die Seitenlänge* $2r$. *Der Flächeninhalt des Ergebnisrechtecks beträgt damit*

$$(2r)^2 = 4\left(25a^2 + \frac{a^2}{4}\right) = 101a^2,$$

während der Flächeninhalt des ursprünglichen Rechtecks $10a^2$ *beträgt. Der Flächeninhalt hat sich also nur aufgrund der Konversion zwischen Kreisscheiben- und Rechteckdarstellung mehr als verzehnfacht.*

Aus diesen Gründen wird die Berechnung mit komplexen Intervallmatrizen in C-XSC auf die Algorithmen zur Multiplikation von reellen Intervallmatrizen zurückgeführt. Al-

gorithmus 3.16 zeigt beispielhaft das Vorgehen bei der Multiplikation zweier komplexer Intervallmatrizen. Die Matrix-Matrix-Produkte in diesem Algorithmus werden dabei mittels Algorithmus 3.14 berechnet.

Eingabe : Zwei komplexe Intervallmatrizen A und B
Ausgabe : Eine Intervalleinschließung $\boldsymbol{C}$ des Produktes AB

```
SetRe(C1, Re(A)*Re(B) + (-Im(A))*Im(B));
SetIm(C1, Re(A)*Im(B) + Im(A)*Re(B));
SetRe(C2, Re(A)*Re(B) + (-Im(A))*Im(B));
SetIm(C2, Re(A)*Im(B) + Im(A)*Re(B));
SetInf(C, C1);
SetSup(C, C2);
```

Algorithmus 3.16: Berechnung der Einschließung des Produktes zweier komplexer Intervallmatrizen

Für das Produkt zweier komplexer Intervallmatrizen werden also vier reelle Intervall-Matrix-Matrix-Produkte mittels Algorithmus 3.14 oder 16 reelle Punktprodukte berechnet. Eine Berechnung in Kreisscheibenarithmetik, wie sie in Intlab benutzt wird, würde nur 10 Produkte reeller Punktmatrizen benötigen. Durch die oben beschriebenen Probleme ist die Nutzung der Kreisscheibenarithmetik für die BLAS-Routinen in C-XSC allerdings trotzdem nicht sinnvoll.

3.2.2. Implementierung

In diesem Abschnitt wird die Verwendung der BLAS- und LAPACK-Routinen in C-XSC erläutert. Zunächst wird dabei auf die Implementierung der BLAS-basierten Matrix-Matrix-, Matrix-Vektor- und Vektor-Vektor-Produkte, welche sich einzeln oder mittels der entsprechenden Operatoren ausführen lassen, eingegangen. Danach wird die LAPACK-Schnittstelle von C-XSC genauer erläutert, welche die für die Arbeit nötigen Routinen auf einfache Art für die C-XSC Matrixdatentypen zur Verfügung stellt.

Nutzung der BLAS-Routinen

Wie in Abschnitt 2.4.1 beschrieben, werden BLAS-Routinen in FORTRAN implementiert. Ein direkter Zugriff auf diese Routinen von C oder C++ aus ist nach einer entsprechenden Deklaration möglich. Allerdings taucht dabei das Problem auf, dass die Routinen je nach verwendetem FORTRAN-Compiler einer anderen Namenskonvention folgen. Details zu diesem Problem folgen im Abschnitt zur Nutzung von LAPACK, wo diese Methode mangels Alternativen angewandt werden muss.

Für die BLAS-Routinen existiert eine standardisierte C-Schnittstelle namens CBLAS. Hier werden C-Funktionen zur Verfügung gestellt, welche einen Wrapper um die jeweilige BLAS-Routine darstellen. Die in Abschnitt 2.4.1 beschriebenen Namenskonventionen für die Routinen werden dabei im Wesentlichen beibehalten, allerdings wird den Funktionsnamen jeweils noch ein `cblas_` vorangestellt.

Weiterhin nutzt CBLAS einige Aufzählungsdatentypen, welche bei manchen Funktionen als Konfigurationsparameter übergeben werden. Für diese Arbeit von Interesse sind dabei die folgenden beiden Enumerations-Datentypen, welche bei Funktionen, die mit Matrizen arbeiten, benutzt werden:

- `enum CBLAS_ORDER {CblasRowMajor, CblasColMajor}`
- `enum CBLAS_TRANSPOSE {CblasNoTrans, CblasTrans, CblasConjTrans}`

`CBLAS_ORDER` gibt dabei an, ob eine Matrix zeilen- (also im C-Stil) oder spaltenweise (also im FORTRAN-Stil) im Speicher vorliegt. Mit Hilfe dieses Parameters müssen also bei Aufrufen der BLAS-Routinen von C/C++ aus die benutzten Matrizen nicht erst transponiert werden, um die unterschiedliche Speicherung auszugleichen. `CBLAS_TRANSPOSE` wird bei den CBLAS-Aufrufen statt dem in BLAS-Routinen sonst üblichen `char` (`'N'` für nicht transponiert bzw. `'T'` für transponiert) benutzt, um zu kennzeichnen, ob mit transponierten Matrizen gerechnet werden soll oder nicht.

Ein weiterer Unterschied ist, dass Funktionen, welche eine komplexe Zahl als Ergebnis zurückgeben (also z.B. `zdotu` zur Berechnung eines komplexen Skalarproduktes), durch sogenannte Subroutinen ersetzt werden, welche das Ergebnis in einem Parameter speichern, statt es als Funktionsergebnis zurückzuliefern. Dies war bei der Festlegung des CBLAS-Standards notwendig, da es in C, anders als in FORTRAN, lange Zeit keinen komplexen Datentyp in der Standardbibliothek gab. Dies hat sich zwar mit dem C99-Standard [65] geändert, das entsprechende Vorgehen in CBLAS wurde aber bis heute beibehalten. Die BLAS-Routine `zdotu` wird in CBLAS daher durch folgende Funktion repräsentiert:

```
void cblas_zdotu_sub(const int N, const void *X, const int incX,
                     const void *Y, const int incY, void *dotu);
```

Dabei ist `N` die Dimension, `X` und `Y` sind Zeiger auf die beteiligten Vektoren. Bei komplexen Berechnungen muss es sich dabei um 2`N` `double`-Werte handeln, wobei der Realteil des i-ten Elementes (bei Indizierung ab 0) an Position $2i$ und der Imaginärteil an Position $2i + 1$ steht. Die Parameter `incX` und `incY` geben die Schrittweite zum nächsten Element an (normalerweise sind also beide Parameter gleich 1), und in `dotu` wird das Ergebnis gespeichert. Der hier übergebene Zeiger muss dabei auf reservierten Speicher für zwei `double`-Werte zeigen, welche dann den Real- und Imaginärteil des Ergebnisses speichern.

Insgesamt werden die in Listing 3.15 angegebenen CBLAS-Funktionen für die Implementierung des C-XSC Interfaces benötigt. Es folgt jeweils (außer für die bereits behandelte Funktion `cblas_zdotu_sub`) eine kurze Erläuterung.

```
1 double cblas_ddot(const int N, const double *X, const int incX,
2                    const double *Y, const int incY);
3
4 void cblas_zdotu_sub(const int N, const void *X, const int incX,
5                       const void *Y, const int incY, void *dotu);
6
```

```
7  void cblas_dgemv(const enum CBLAS_ORDER order,
8                   const enum CBLAS_TRANSPOSE TransA, const int M,
9                   const int N, const double alpha, const double *A,
10                  const int lda, const double *X, const int incX,
11                  const double beta, double *Y, const int incY);
12
13 void cblas_zgemv(const enum CBLAS_ORDER order,
14                  const enum CBLAS_TRANSPOSE TransA, const int M,
15                  const int N, const void *alpha, const void *A,
16                  const int lda, const void *X, const int incX,
17                  const void *beta, void *Y, const int incY);
18
19 void cblas_dgemm(const enum CBLAS_ORDER Order,
20                  const enum CBLAS_TRANSPOSE TransA,
21                  const enum CBLAS_TRANSPOSE TransB, const int M,
22                  const int N, const int K, const double alpha,
23                  const double *A, const int lda, const double *B,
24                  const int ldb, const double beta, double *C,
25                  const int ldc);
26
27 void cblas_zgemm(const enum CBLAS_ORDER Order,
28                  const enum CBLAS_TRANSPOSE TransA,
29                  const enum CBLAS_TRANSPOSE TransB, const int M,
30                  const int N, const int K, const void *alpha,
31                  const void *A, const int lda, const void *B,
32                  const int ldb, const void *beta, void *C,
33                  const int ldc);
```

Listing 3.15: Benötigte CBLAS-Routinen

- `cblas_ddot`: Berechnet ein reelles Skalarprodukt. Die Parameter entsprechen in ihrer Bedeutung denen der Funktion `cblas_zdotu_sub`, das Ergebnis wird hier allerdings als Rückgabewert der Funktion zurückgegeben.

- `cblas_dgemv`: Berechnet den Ausdruck $y = \alpha Ax + \beta y$, wobei A eine $m \times n$ Matrix, x und y Vektoren der Dimension n und α und β Skalare sind. Die Parameter `order` und `TransA` der Funktion geben an, ob A zeilen- oder spaltenweise vorliegt und ob mit der Transponierten von A gerechnet werden soll. Die Matrix A und die Vektoren x und y werden beschrieben durch die Parameter `A`, `X` und `Y`, ihre Dimension wird durch `M` und `N` angegeben, α und β werden durch die Parameter `alpha` und `beta` festgelegt. Durch `incX` und `incY` wird wie zuvor die Schrittweite beim Durchlaufen von `X` und `Y` angegeben. Der Parameter `lda` schließlich gibt die tatsächliche erste Dimension der gespeicherten Matrix an (falls ein $m \times n$ Block aus einer größeren Matrix verwendet werden soll).

- `cblas_zgemv`: Für diese Funktion gilt das Gleiche wie für die Funktion `cblas_dgemv`, nur dass A, x, y, α und β komplex sind.

- `cblas_dgemm`: Berechnet den Ausdruck $C = \alpha AB + \beta C$, wobei A eine $m \times n$ Matrix, B eine $n \times k$ Matrix und C eine $m \times k$ Matrix ist. α und β sind wiederum

Skalare. Die Parameter `order`, `TransA` und `TransB` beschreiben wieder, wie die gespeicherten Werte für die Matrizen zu interpretieren sind, während über die Parameter `A`, `B`, `C`, `M`, `N`, `K`, `lda`, `ldb` und `ldc` die Dimension und die Einträge der drei Matrizen beschrieben werden. Weiterhin geben `alpha` und `beta` wieder die Werte der beiden Skalare an.

- `cblas_zgemm`: Bei dieser Funktion handelt es sich wiederum um die komplexe Version von `cblas_dgemm`, sie arbeitet daher analog.

Für die Nutzung der BLAS-Routinen mit den C-XSC Datentypen werden die folgenden drei Funktionen verwendet, welche jeweils für die diversen zutreffenden Datentypen überladen werden und die Algorithmen aus Abschnitt 3.2.1 implementieren:

- `blasdot`: Berechnet das Skalarprodukt der beiden übergebenen Vektoren. Das Ergebnis wird in einem dritten, per Referenz übergebenem Parameter gespeichert. Für Punkt-Skalarprodukte gibt es dabei auch eine Version, der zur Speicherung des Ergebnisses ein entsprechender Intervalldatentyp übergeben wird. Diese Version berechnet dann mit Hilfe des Rundungsmodus eine Einschließung des Skalarproduktes.

- `blasmvmul`: Berechnet das Matrix-Vektor-Produkt zwischen einer übergebenen Matrix und einem übergebenen Vektor. Der Ergebnisvektor wird wiederum in einem dritten Parameter gespeichert. Auch hier gibt es jeweils eine reelle und komplexe Version zur Berechnung der Einschließung der entsprechenden Punktberechnung.

- `blasmatmul`: Berechnet das Produkt zweier Matrizen. Wie bei den anderen Funktionen gibt es auch hier eine überladene Version der Funktion, welche die Einschließung für das Produkt zweier Punktmatrizen berechnet.

Die Funktionen können über den Header `cxsc_blas.hpp` eingebunden werden. Zur Verdeutlichung der Implementierung wird im Folgenden jeweils für jede Funktion die Version für einfache reelle Punktprodukte angegeben. Die Versionen für Intervallberechnungen sind auf entsprechende Art mit Hilfe der Algorithmen aus Abschnitt 3.2.1 implementiert, ebenso lassen sich Anpassungen für die komplexen Versionen vornehmen. Da an dieser Stelle der Schwerpunkt aber auf Implementierungsdetails liegen soll und die (komplexen) Intervallversionen der jeweiligen Funktion sehr umfangreich sind, wird hier auf deren Angabe verzichtet.

```
1 inline void blasdot(const rvector& x, const rvector& y, real& res) {
2    int n = VecLen(x);
3    int inc=1;
4
5    double* xd = (double*) x.to_blas_array();
6    double* yd = (double*) y.to_blas_array();
7
8    res = cblas_ddot(n, xd, inc, yd, inc);
```

```
9 }
```

Listing 3.16: Reelles Skalarprodukt mittels BLAS

Listing 3.16 zeigt zunächst die Funktion zur Berechnung eines reellen Skalarproduktes. Entscheidend bei allen Funktionen ist die Frage, wie die C-XSC Datentypen (wie `rvector` in diesem Beispiel) in das für den CBLAS Aufruf nötige Format überführt werden können. Dadurch, dass C-XSC-Vektoren und -Matrizen intern ebenfalls als eindimensionale Arrays von `double`-Werten gespeichert werden, ist bei Punktmatrizen und -vektoren keine explizite Konversion (also ein Umkopieren der Werte) erforderlich. Mittels der Memberfunktion `to_blas_array` liefern diese einen Pointer, der direkt auf die entsprechende Speicherstelle zeigt und im CBLAS Aufruf verwendet werden kann.

Dies gilt auch im Komplexen, da der C-XSC Datentyp `complex` letztlich nur aus zwei im Speicher hintereinander liegenden `double`-Werten besteht und somit die entsprechenden Matrix- und Vektortypen ebenfalls direkt das passende Format für den CBLAS-Aufruf haben. Bei Verwendung von Matrizen muss auf ein korrektes Setzen der Konfigurationsparameter für die Orientierung der Matrix im Speicher und die Transponierung geachtet werden, wie beim Matrix-Vektor-Produkt in Listing 3.17 und beim Matrix-Matrix-Produkt in Listing 3.18 zu sehen.

```
1  inline void blasmvmul(const rmatrix& A, const rvector& x, rvector& r) {
2
3     double* DA = (double*) A.to_blas_array();
4     double* dx = (double*) x.to_blas_array();
5     double* dr = (double*) r.to_blas_array();
6
7     const int m = Ub(A,1) - Lb(A,1) + 1;
8     const int n = Ub(A,2) - Lb(A,2) + 1;
9     const double alpha = 1.0, beta = 0.0;
10    const int inc = 1;
11
12    cblas_dgemv(CblasRowMajor, CblasNoTrans, m, n,
13                alpha, DA, n, dx, inc, beta, dr, inc);
14 }
```

Listing 3.17: Reelles Matrix-Vektor-Produkt mittels BLAS

```
1  inline void blasmatmul(const rmatrix &A, const rmatrix &B, rmatrix &C) {
2
3     double* DA = (double*) A.to_blas_array();
4     double* DB = (double*) B.to_blas_array();
5     double* DC = (double*) C.to_blas_array();
6
7     const int m = Ub(A,1) - Lb(A,1) + 1;
8     const int n = Ub(A,2) - Lb(A,2) + 1;
9     const int o = Ub(C,2) - Lb(C,2) + 1;
10    const double alpha = 1.0, beta = 0.0;
11
12    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, o,
13                n, alpha, DA, n, DB, o, beta, DC, o);
14 }
```

Listing 3.18: Reelles Matrix-Matrix-Produkt mittels BLAS

Bei beiden Funktionen muss der Parameter für zeilenweise Speicherung der beteiligten Matrizen (`CblasRowMajor`) benutzt werden, da C-XSC seine Matrizen entsprechend der C-Konvention zeilenweise speichert. Weiterhin darf daher auch nicht mit transponierten Matrizen gerechnet werden, es muss also für jede Matrix jeweils `CblasNoTrans` gesetzt werden.

Bei Intervallvektoren und -matrizen (im Reellen wie im Komplexen) kann nicht direkt auf den Speicher des C-XSC Datentyps zugegriffen werden. Dies ist aber ohnehin nicht nötig, da in diesem Fall durch die Konversion in Mittelpunkt-Radius-Darstellung zunächst ein Mittelpunktsvektor bzw. eine Mittelpunktsmatrix sowie ein Radiusvektor bzw. eine Radiusmatrix berechnet werden müssen. Diese werden dann direkt in von CBLAS nutzbaren `double`-Arrays gespeichert.

Um die BLAS-Funktionen in C-XSC zu nutzen, gibt es folgende Möglichkeiten:

- Direkter Aufruf der entsprechenden Funktion.
- Über die jeweiligen Operatoren der Matrix- und Vektordatentypen.

In letzterem Fall werden die BLAS-Routinen nur dann verwendet, wenn die Präzision für Operatoren auf 1 gesetzt ist (über die globale Variable `opdotprec`, siehe Abschnitt 3.1) und die Präprozessorvariable `CXSC_USE_BLAS` gesetzt wurde, entweder direkt im Programm oder durch Verwendung einer entsprechende Option bei der Kompilierung. Zur Berechnung der Einschließung eines Punktproduktes muss entweder direkt die entsprechende Funktion aufgerufen werden, oder einer der Operanden muss in einen passenden Intervalldatentyp (dessen Elemente dann ausschließlich Punktintervalle sind) umgewandelt werden, da die Operatoren auf den vorgegebenen Rückgabetyp festgelegt sind (das Produkt aus zwei reellen Punktmatrizen ist bei Verwendung des zugehörigen Operators grundsätzlich ebenfalls eine reelle Punktmatrix). Zur Verdeutlichung zeigt Listing 3.19 ein Beispielprogramm.

```
1  #include <imatrix.hpp>
2
3  using namespace cxsc;
4
5  int main() {
6    const int n = 1000;
7    interval i;
8    ivector x(n), y(n);
9    rmatrix A(n,n), B(n,n), C(n,n);
10   imatrix CI(n,n);
11
12   //...mit beliebigen Werten fuellen...
13
14   //einfach Genauigkeit ist Voraussetzung
15   opdotprec = 1;
16
17   //Folgende Berechnungen werden mit BLAS-Unterstuetzung durchgefuehrt
18   i = x * y;
```

```
19    y = A * x;
20    C = A * B;
21    //Berechnung der Einschliessung ueber Typumwandlung (ineffizient)
22    CI = A * imatrix(B);
23
24    opdotprec = 2;
25    //Die naechste Anweisung wird NICHT mit BLAS ausgefuehrt,
26    //da Genauigkeit fuer Operatoren nun 2 ist
27    C = A * B;
28
29    //Die BLAS-Berechnung kann aber direkt aufgerufen werden
30    blasmatmul(A,B,C);
31    //Ausserdem kann so eine Einschliessung berechnet werden
32    blasmatmul(A,B,CI); //Effizienter im Vergleich zu Zeile 22
33
34    return 0;
35  }
```

Listing 3.19: Beispielprogramm zur Verwendung der BLAS-Routinen

In jedem Fall ist bei der Kompilierung außer dem evtl. nötigen Setzen der Präprozessorvariablen (in der Regel geschieht dies über die Option `-DCXSC_USE_BLAS`) zu beachten, dass zu einer passenden BLAS- sowie zu einer CBLAS-Bibliothek gelinkt werden muss.

Weiterhin ist darauf zu achten, dass die verwendete BLAS-Bibliothek keinen Einfluss auf die aktuelle Konfiguration für Gleitkommaoperationen im entsprechenden Prozessor-Kontrollwort nimmt, da ansonsten der Wechsel des Rundungsmodus durch die C-XSC Funktionen unter Umständen rückgängig gemacht wird und somit falsche Ergebnisse berechnet werden. In Tests als geeignet erwiesen haben sich aktuelle Versionen der Intel Math Kernel Library (MKL) [64], der von Apple in Mac OS X mitgelieferten optimierten BLAS-Bibliothek sowie von ATLAS BLAS [141]. Letztere Bibliothek optimiert sich bei der Kompilierung selbstständig für das jeweilige System. Somit sollte auf jeder Plattform zumindest eine optimierte Version der BLAS-Bibliothek zur Verfügung stehen, die sich für C-XSC benutzen lässt. Als nicht geeignet erwiesen hat sich GOTO-BLAS (getestet mit Version 1.23) [47].

Nutzung von LAPACK

Um in C-XSC die Möglichkeit zu haben, mit den Standard-C-XSC-Datentypen auf einfache Weise performante Implementierungen der wichtigsten numerischen Algorithmen aus der linearen Algebra nutzen zu können, ist eine Schnittstelle zu den Funktionen aus LAPACK sinnvoll, welche z.B. auch in Matlab genutzt werden. Dadurch können die stark optimierten Routinen aus LAPACK zur Berechnung der in vielen verifizierenden Algorithmen nötigen Näherungslösungen und/oder Präkonditionierer genutzt werden.

Für diese Arbeit von Bedeutung sind dabei die Berechnung einer Näherungsinversen einer Matrix, welche für die dicht besetzten Löser aus Kapitel 4 benötigt wird, sowie die Berechnung der näherungsweisen QR-Zerlegung einer Matrix, welche für einen Teil des in Abschnitt 5.1.1 beschriebenen Löser benötigt wird. Für beide Berechnungen wird daher eine Schnittstelle zur entsprechenden LAPACK-Funktion implementiert.

Weiterhin stehen für beide Algorithmen zusätzlich zur Schnittstelle zum jeweiligen LAPACK-Aufruf auch einfache Varianten der Algorithmen direkt mit den C-XSC Datentypen zur Verfügung, so dass die Funktionalität auch genutzt werden kann, wenn keine LAPACK-Bibliothek auf dem jeweiligen System vorhanden ist. Diese zusätzlichen Implementierungen sind allerdings, obwohl sie teilweise mittels OpenMP parallelisiert sind, deutlich langsamer als die stark optimierten LAPACK-Routinen und sollten daher nur in Ausnahmefällen, in denen keinerlei LAPACK-Implementierung zur Verfügung steht, genutzt werden.

Im Folgenden soll beispielhaft das Vorgehen zur Erstellung der Schnittstelle für die Routine zur Matrixinversion verdeutlicht werden, die Implementierung der QR-Zerlegung erfolgt nach dem gleichen Prinzip. Wie bei der Benutzung der BLAS-Bibliothek, muss auch die Nutzung von LAPACK über das Setzen einer Präprozessorvariablen (in diesem Fall `CXSC_USE_LAPACK`) aktiviert werden. Für die oben aufgeführten Funktionalitäten stehen entsprechende, für die verschiedenen Datentypen überladene, Funktionen zur Verfügung, die, abhängig davon, ob die nötige Präprozessorvariable gesetzt ist, entweder die zugehörige LAPACK-Routine zur Berechnung starten oder eine selbstimplementierte Variante der jeweiligen Funktionalität ausführen. Listing 3.20 zeigt die Implementierung der Funktion `MatInvAprx` zur Matrixinversion.

```
1  inline void MatInvAprx( rmatrix &A, rmatrix &R, int& error ) {
2  #ifdef CXSC_USE_LAPACK
3      int n = RowLen(A);
4      int nb = 256, ld = n, info, lwork = n*nb;
5      error = 0;
6
7      double *DA = new double[n*n];
8      double *work = new double[lwork];
9      int *ipiv  = new int[n];
10
11     //Kopie von rmatrix in double array
12     for(int i=1 ; i<=n ; i++) {
13        for(int j=1 ; j<=n ; j++) {
14           DA[(j-1)*n+(i-1)] = _double(A[Lb(A,1)+i-1][Lb(A,2)+j-1]);
15        }
16     }
17
18     //LU-Zerlegung mit LAPACK
19     dgetrf_(&n, &n, DA, &ld, ipiv, &info);
20
21     if(info != 0) {
22       error = 1;
23       delete[] DA;
24       delete[] ipiv;
25       delete[] work;
26       return;
27     }
28
29     //Berechnung der Inversen mit LAPACK aus LU-Zerlegung
30     dgetri_(&n, DA, &ld, ipiv, work, &lwork, &info);
31
```

```
32      //Kopie der berechneten Inversen in rmatrix R
33      for(int i=1 ; i<=n ; i++) {
34          for(int j=1 ; j<=n ; j++) {
35              R[Lb(R,1)+j-1][Lb(R,2)+i-1] = DA[(i-1)*n+(j-1)];
36          }
37      }
38
39      if(info != 0) {
40        error = 1;
41      }
42
43      //Freigeben der double arrays
44      delete[] DA;
45      delete[] ipiv;
46      delete[] work;
47
48      return;
49
50  #else
51
52      //Eigene Implementierung....
```

Listing 3.20: Implementierung der Matrixinversion als LAPACK-Schnittstelle

Die eigene Implementierung ohne LAPACK-Unterstützung basiert auf dem Gauß-Jordan-Algorithmus und stützt sich auf die Implementierung aus der C-XSC-Toolbox [53], ohne allerdings den langen Akkumulator zu verwenden (es wird nur reine Gleitkommarechnung benutzt). Weiterhin werden einige for-Schleifen mittels OpenMP parallelisiert.

Die Funktion `MatInvAprx` erhält als Parameter die zu invertierende Matrix `A` sowie eine Matrix `R`, in welcher die Inverse gespeichert werden soll (`R` muss dabei schon bei Aufruf der Funktion die korrekte Größe haben). Weiterhin wird eine Fehlervariable mit übergeben, welche einen Fehlercode zurück gibt, falls die Invertierung fehlschlägt (z.B. falls `A` singulär ist). Die Verwendung von Fehlervariablen statt Exceptions erfolgt aus Gründen der Kompatibilität mit den bestehenden Toolbox-Algorithmen.

Für den Aufruf der LAPACK-Funktionen müssen die FORTRAN-Routinen direkt aufgerufen werden, da für LAPACK keine standardisierte C-Schnittstelle zur Verfügung steht. Abhängig vom verwendeten FORTRAN-Compiler müssen die Funktionen dabei über einen leicht anderen Namen deklariert und ausgeführt werden. Listing 3.21 zeigt, wie die hier benötigten Funktionen deklariert werden.

```
1  #ifdef Unchanged
2    #define dgetri_ dgetri
3    #define dgetrf_ dgetrf
4  #elif defined(Add_)
5    //Nothing to do
6  #elif defined(Uppercase)
7    #define dgetri_ DGETRI
8    #define dgetrf_ DGETRF
9  #endif
10
```

```
11 //Declaration of LAPACK-routines
12 extern "C" {
13   void dgetri_(int *n, double* A, int *lda, int* ipiv, double* work,
14                int *lwork, int *info);
15   void dgetrf_(int *m, int *n, double* A, int *lda, int* ipiv,
16                int *info);
17 }
```

Listing 3.21: Beispielprogramm zur Verwendung der BLAS-Routinen

Über das Setzen einer Präprozessorvariablen kann hier also die korrekte Namensgebung aktiviert werden. Die Funktionsnamen können dabei entweder unverändert bleiben (`Unchanged`), um einen Unterstrich erweitert werden (`Add_`) oder in Großbuchstaben umgewandelt werden (`Uppercase`). In jedem Fall müssen die Funktionen danach mittels `extern "C"` deklariert werden. Der Linker bindet bei der späteren Programmerstellung dann den richtigen Code aus der LAPACK Bibliothek an den Funktionsaufruf.

Da hier direkt FORTRAN-Routinen aufgerufen werden, muss die Matrix `A` spaltenweise im Speicher vorliegen. Daher wird sie zu Beginn transponiert in ein entsprechendes `double`-Array umkopiert (Zeilen 11-16 in Listing 3.20). Danach folgt zunächst ein Aufruf der LAPACK-Funktion `dgetrf` zur LU-Zerlegung. Mittels der berechneten LU-Zerlegung wird über `degtri` dann die Inverse bestimmt. Nach beiden Funktionsaufrufen wird jeweils abgefragt, ob ein Fehler aufgetreten ist. Bei Auftreten eines Fehlers wird die Fehlervariable auf 1 gesetzt, die belegten Ressourcen werden freigegeben und die Funktion wird abgebrochen. Bei erfolgreicher Invertierung wird zum Abschluss die berechnete Inverse dann in die als Parameter übergebene Matrix `R` kopiert.

Wird in einem Programm eine LAPACK-Schnittstelle benutzt, so muss zur Verwendung der LAPACK-Funktionen bei der Kompilierung neben dem Setzen der Präprozessorvariable `CXSC_USE_LAPACK` auch zu einer passenden LAPACK-Bibliothek und auch zu einer BLAS-Bibliothek (da LAPACK auf BLAS aufbaut) gelinkt werden. Dabei ist auch auf das zusätzliche Setzen einer der drei oben erwähnten Präprozessorvariablen zu achten, um die korrekte Benennung der Fortran-Funktionen zu verwenden. Falls die LAPACK-Unterstützung nicht genutzt wird, so sollte auf Mehrkern- oder Mehrprozessorsystemen zumindest OpenMP über einen entsprechenden Compilerschalter aktiviert werden, um so von der Parallelisierung der alternativen Implementierung ohne LAPACK-Unterstützung Gebrauch zu machen.

3.2.3. Tests und Zeitmessungen

In diesem Abschnitt wird der Performancegewinn bei Benutzung der Schnittstelle zu BLAS und LAPACK gegenüber den herkömmlichen Berechnungsmethoden in C-XSC genauer untersucht. Weiterhin wird ein kurzer Vergleich mit Intlab durchgeführt und es wird die Auswirkung der BLAS-Unterstützung auf die Breite der berechneten Einschließung geprüft. Alle Tests in diesem Abschnitt erfolgen auf einem System mit 2 Intel Xeon E5520 CPUs mit jeweils 4 Kernen, welche mit 2.27 GHz getaktet sind, und 24 GB Hauptspeicher (DDR3). Als Compiler wurde der Intel Compiler 11.1 sowie als BLAS-Bibliothek die Intel Math Kernel Library Version 10.2 genutzt. Für alle Tests wurden

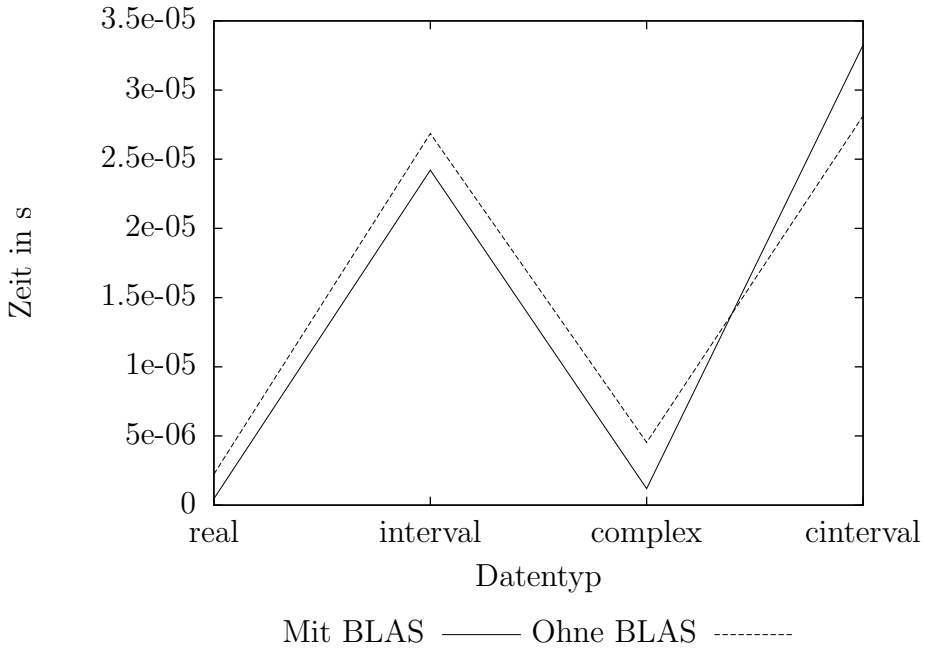


Abbildung 3.9.: Zeitvergleich Skalarprodukte

volle Compileroptimierungen mittels der Option `-O3` aktiviert. Die MKL macht für alle BLAS- und LAPACK-Routinen, soweit sinnvoll, von allen acht Kernen des Systems Gebrauch, die normalen C-XSC-Implementierungen hingegen verwenden nur einen Kern.

Zunächst werden Zeitvergleiche für Vektor-Vektor- und Matrix-Vektor-Produkte für Vektoren der Dimension $n = 1000$ und Matrizen der Dimension $n \times n = 1000 \times 1000$ mit und ohne BLAS-Unterstützung für alle Datentypen durchgeführt. Alle Berechnungen in diesem Abschnitt werden mit *double*-Präzision ($K = 1$) vorgenommen. Jede Messung wurde mehrfach wiederholt (Vektor-Vektor-Produkte 100000 mal, Matrix-Vektor-Produkte 1000 mal und die später getesteten Matrix-Matrix-Produkte 10 mal), anschließend wurde der Durchschnitt aller Messungen als Wert übernommen. Die Abbildungen 3.9 und 3.10 zeigen jeweils die Ergebnisse.

In beiden Fällen bietet die BLAS-Version leichte Geschwindigkeitsvorteile von etwa 10-20%. Eine Ausnahme bilden die komplexen Intervalle, bei welchen die nötigen Umwandlungen von Real- und Imaginärteil in Mittelpunkt-Radius-Darstellung im Vergleich zur eigentlichen Berechnung recht aufwändig sind. Dadurch werden die (hier sowieso nur leichten) Geschwindigkeitsvorteile durch die BLAS-Berechnung wieder aufgehoben. Wie in Abschnitt 2.4.1 erläutert, sind Vektor-Vektor- sowie Matrix-Vektor-Produkt Routinen aus den BLAS-Leveln 1 bzw. 2 und bieten daher von vorneherein kein allzu großes Optimierungspotential, weshalb der Geschwindigkeitsgewinn auch relativ gering bleibt. Ein deutlich anderes Bild ergibt sich für Matrix-Matrix-Produkte. Die Ergebnisse der entsprechenden Zeitmessungen lassen sich Abbildung 3.11 entnehmen.

In dieser Abbildung ist zu beachten, dass die y-Achse (Zeit in Sekunden) hier eine logarithmische Skala verwendet, die in der Abbildung zu erkennenden Unterschiede

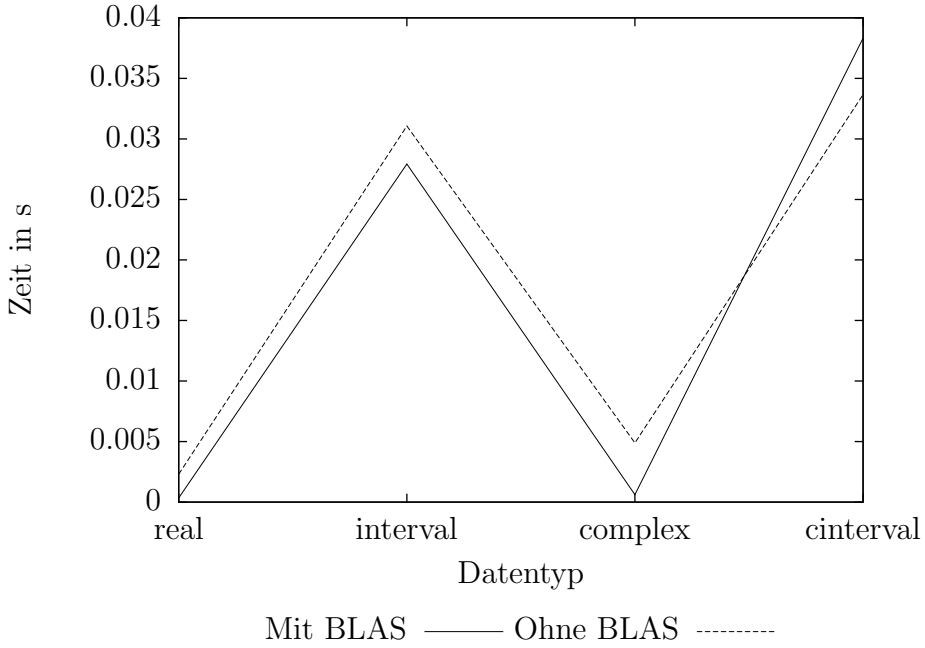


Abbildung 3.10.: Zeitvergleich Matrix-Vektor-Produkte

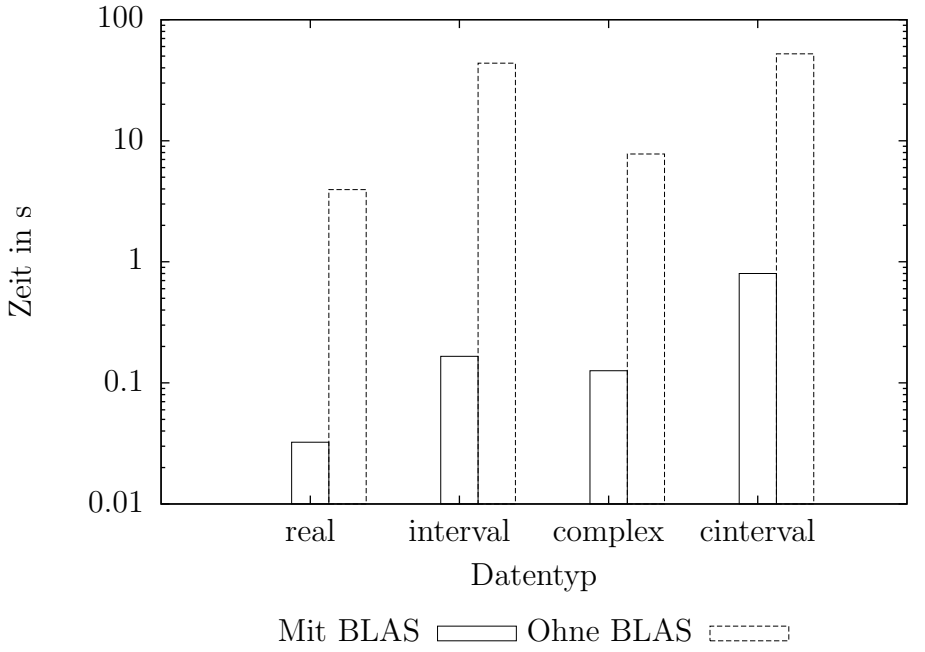


Abbildung 3.11.: Zeitvergleich Matrix-Matrix-Produkte

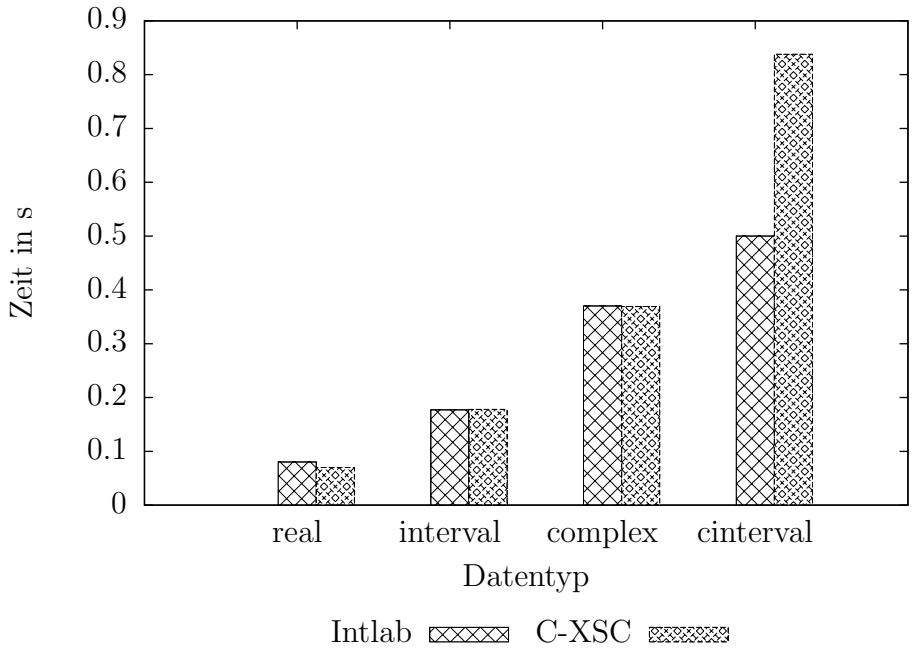


Abbildung 3.12.: Zeitvergleich Matrix-Matrix-Produkte mit Intlab

sind also noch wesentlich deutlicher als auf den ersten Blick ersichtlich. Beim Matrix-Matrix-Produkt handelt es sich um eine Routine aus BLAS-Level 3, welche extrem gut optimierbar und auch parallelisierbar ist, d.h. die auf dem Testsystem zur Verfügung stehenden 8 Kerne können hier sehr gut ausgenutzt werden.

Für reelle und komplexe Punktmatrizen wird die Berechnung dadurch um einen Faktor von mehr als 100 schneller, für reelle Intervallmatrizen sogar um einen Faktor von mehr als 250. Diese starke Beschleunigung bei Intervallmatrizen kommt dadurch zustande, dass die Standard-C-XSC Implementierung das Produkt elementweise berechnet, wodurch sehr viele Wechsel des Rundungsmodus nötig werden, welche sich negativ auf die Geschwindigkeit auswirken (siehe Abschnitt 51).

Für komplexe Intervallmatrizen wiederum ist der Geschwindigkeitszuwachs mit einem Faktor von ca. 50 deutlich geringer. Dieser geringere Zuwachs kommt durch die aufwändigere Berechnung bei der Benutzung der BLAS-Routinen zustande, welche durch den Verzicht auf die Nutzung der Kreisscheibenarithmetik im Komplexen (wie in Abschnitt 3.2.1 erläutert) nötig wird. Trotzdem bleibt der Performancegewinn auch in diesem Fall enorm.

Als nächstes wird die Geschwindigkeit des Matrix-Matrix-Produktes, der in diesem Zusammenhang wichtigsten Operation, mit Intlab verglichen. Im Unterschied zum vorherigen Test wird dabei für die beiden Punktmatrix-Produkte eine Intervalleinschließung des korrekten Ergebnisses berechnet. Mit Intlab hat Rump als erster die effektive Benutzung von BLAS-Routinen in der Intervallrechnung demonstriert. Wie schon erwähnt basiert auch die in diesem Abschnitt beschriebene C-XSC Implementierung auf den in Intlab verwendeten Algorithmen [130]. Abbildung 3.12 zeigt die Ergebnisse des Verglei-

Datentyp	BLAS (8 Threads)	BLAS (1 Thread)	Ohne BLAS
real	$1.69 \cdot 10^{-13}$	$1.36 \cdot 10^{-12}$	$2.69 \cdot 10^{-12}$
interval	$2.55 \cdot 10^{-10}$	$2.56 \cdot 10^{-10}$	$2.57 \cdot 10^{-10}$
complex	$4.30 \cdot 10^{-13}$	$3.42 \cdot 10^{-12}$	$6.74 \cdot 10^{-12}$
cinterval	$6.38 \cdot 10^{-10}$	$6.41 \cdot 10^{-10}$	$6.43 \cdot 10^{-10}$

Tabelle 3.2.: Vergleich der relativen Einschließungsbreite mit und ohne BLAS-Unterstützung

ches.

Für den Test wurde Intlab 6 mit Matlab 7.9.0 verwendet. Um einen korrekten Vergleich zu ermöglichen, wurde jeweils die gleiche Hauptversion der BLAS-Bibliothek (Intel MKL 10.2) verwendet. Wie die Abbildung zeigt, liegt C-XSC im Reellen und bei komplexen Punktmatrizen in etwa gleichauf mit Intlab oder hat leichte Geschwindigkeitsvorteile. Diese dürften insbesondere durch den (in diesem Fall aber nur leichten) Interpretations-Overhead von Intlab zustande kommen (der Intlab Code muss erst von Matlab interpretiert werden). Wiederum ergibt sich ein deutlicher Unterschied bei den komplexen Intervallen, wo Intlab durch die native Verwendung der Kreisscheibenarithmetik einen klaren Vorteil hat.

Als nächstes soll geprüft werden, ob sich die Nutzung der BLAS-Routinen spürbar auf die Genauigkeit der Resultate auswirkt. Dazu werden Einschließungen für Matrix-Matrix-Produkte mit einer Reihe von Testmatrizen berechnet und anschließend der durchschnittliche Durchmesser der Einschließung bestimmt. Als Beispiel zeigt Tabelle 3.2 die Ergebnisse bei Berechnung mit Zufallsmatrizen (für die Intervallberechnungen werden diese Punktmatrizen um einen Faktor 10^{-13} aufgebläht), welche repräsentativ für alle Testergebnisse sind. Für komplexe Matrizen wird jeweils der relative Durchmesser von Real- und Imaginärteil getrennt betrachtet und der größere der beiden Werte angegeben.

Wie zu sehen ist, hat die Benutzung der BLAS-Routinen in der Praxis im Allgemeinen keinen negativen Effekt, sondern erhöht im Gegenteil die Genauigkeit oftmals sogar leicht, insbesondere bei Verwendung von 8 Threads. Grund dafür dürfte die andere Berechnungsreihenfolge (vor allem bedingt durch die Verteilung der Berechnung auf mehrere Threads) bei den BLAS-Routinen sein, die sich im Allgemeinen positiv auf die Genauigkeit auswirkt. Insgesamt zeigen die Tests also, dass die C-XSC-BLAS-Routinen, insbesondere bei Berechnung von Matrix-Matrix-Produkten, von sehr großem Wert bei der Erstellung von laufzeitoptimierten Programmen mit der C-XSC Bibliothek sein können.

Zum Abschluss dieses Abschnitts folgt nun noch ein Zeitvergleich für die LAPACK-Schnittstelle. Dazu wird die Matrixinvertierung für reelle Matrizen (Abbildung 3.13) und für komplexe Matrizen (Abbildung 3.14) verglichen. Dabei wird die Zeit für die Berechnung mittels LAPACK, ohne LAPACK und ohne LAPACK aber mit Benutzung

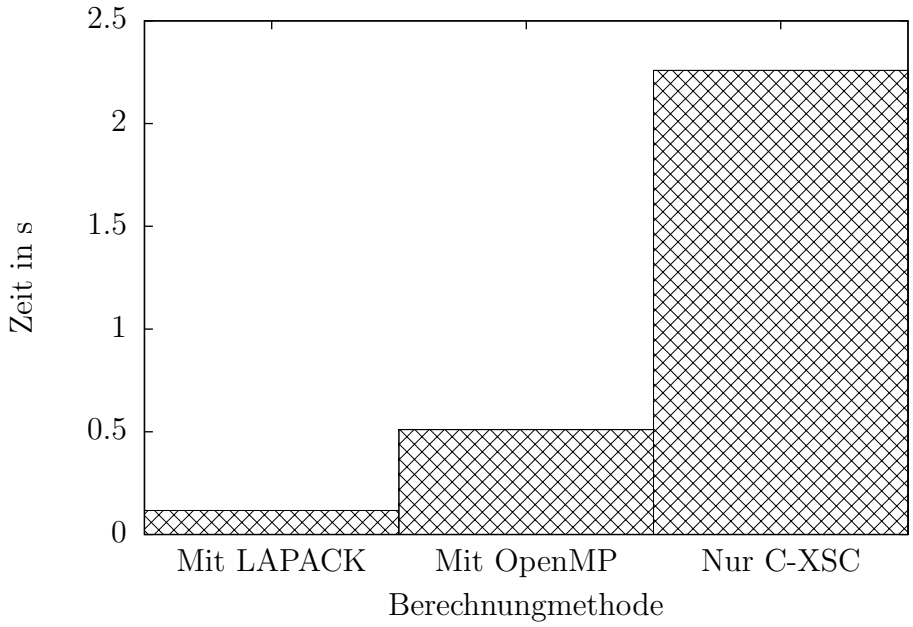


Abbildung 3.13.: Zeitvergleich Matrixinvertierung reell

von OpenMP (d.h. einer Parallelisierung „von Hand“) verglichen.

Es ergibt sich im Reellen wie im Komplexen ein ähnliches Bild: Die Verwendung von LAPACK ist erwartungsgemäß die deutlich schnellste Lösung. Allerdings liefert auch die selbstimplementierte Version mit Benutzung von OpenMP einen guten Speed-Up gegenüber der seriellen Version. Falls keine LAPACK-Bibliothek, aber ein OpenMP-fähiger Compiler und ein Mehrkern- bzw. Mehrprozessorsystem zur Verfügung steht, ist also trotzdem eine relativ performante Berechnungsmöglichkeit gegeben.

3.3. Datentypen für dünn besetzte Vektoren und Matrizen

C-XSC bietet eine Vielzahl von mächtigen Matrix- und Vektordatentypen für alle Grunddatentypen (siehe Abschnitt 2.3). Neben der einfachen Benutzung durch überladene Operatoren sind auch der lesende und schreibende Zugriff auf Teilmatrizen und -vektoren, die freie Indexwahl sowie die Möglichkeit, die Genauigkeit von Skalarprodukten zur Laufzeit festzulegen (Abschnitt 3.1) und optional optimierte BLAS-Routinen nutzen zu können (Abschnitt 3.2) erwähnenswert. Mit Hilfe der `dotprecision`-Datentypen ist es möglich, ganze Skalarproduktausdrücke in höherer oder sogar maximaler Genauigkeit zu berechnen.

Die Standard Matrix- und Vektordatentypen sind allerdings auf dicht besetzte Matrizen und Vektoren ausgelegt. Enthält ein Vektor, und insbesondere eine Matrix, sehr viele Nulleinträge (so viele, dass sich deren Ausnutzung in Bezug auf Rechenzeit und

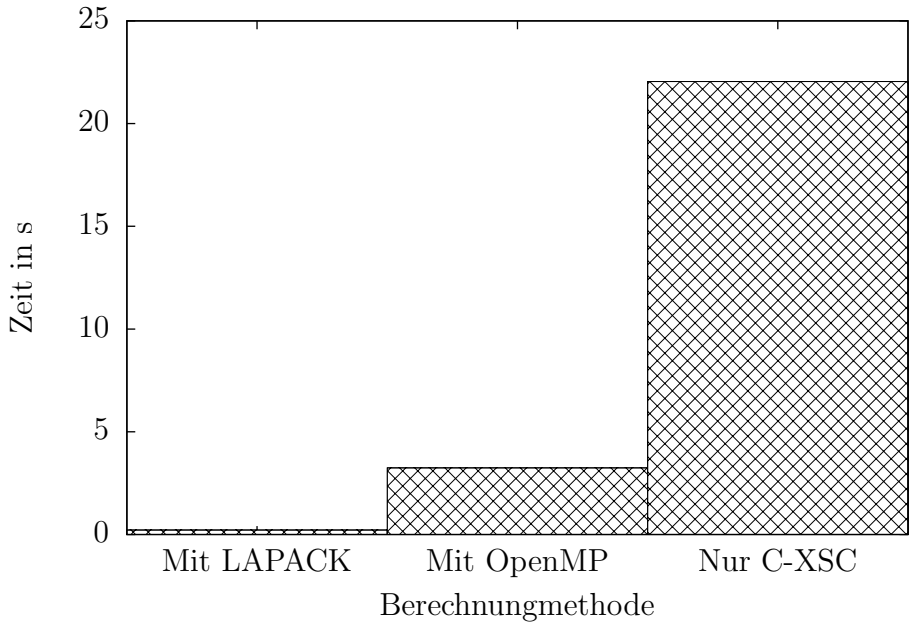


Abbildung 3.14.: Zeitvergleich Matrixinvertierung komplex

Speicherbedarf lohnt), so spricht man von dünn besetzten Matrizen (und Vektoren). Für diese sind eigene Datentypen erforderlich, welche die spärliche Besetztheit durch Nutzung einer speziellen Datenstruktur ausnutzen und so bei der Arbeit mit solchen Matrizen und Vektoren in der Regel deutliche Vorteile sowohl hinsichtlich Rechenzeit- als auch Speicherbedarf mit sich bringen. Die Verfügbarkeit solcher Datentypen ist auch eine Grundvoraussetzung für die in Kapitel 5 beschriebenen Löser für dünn besetzte Systeme.

Im Folgenden werden spezielle Datentypen für die Benutzung von dünn besetzten Vektoren und Matrizen und ihre Einbindung in C-XSC beschrieben. Eine wichtige Prämisse für die neuen Datentypen ist, dass sie soweit wie möglich die gleiche Schnittstelle mit der gleichen Funktionalität bieten sollen wie die dicht besetzten Typen. Dies soll zum einen die Benutzung für erfahrene C-XSC Anwender erleichtern, zum anderen aber auch die Erstellung von Template-basiertem Code ermöglichen, der dann sowohl für dicht als auch für dünn besetzte Datentypen ausgeprägt werden kann. Dabei ist allerdings zu beachten, dass für ein effektives Arbeiten mit dünn besetzten Matrizen oftmals speziell angepasste Algorithmen erforderlich sind, weshalb ein solches Vorgehen nicht immer sinnvoll ist.

Aus diesem Grund ist es von großer Bedeutung, eine Datenstruktur zu wählen, die auch in vielen anderen Softwarepaketen verwendet wird, um so für bestimmte Problemstellungen auf einfache Art eine Schnittstelle zwischen C-XSC und der jeweiligen Software herstellen zu können.

Zusammenfassend sollen also folgende Punkte durch die neuen Datentypen abgedeckt werden:

- Unterstützung der gleichen Operatoren und Funktionen wie bei den dicht besetzten Typen.
- Skalarproduktausdrücke mit beliebig wählbarer Präzision (über Operatoren oder `dotprecision`).
- Lesender und schreibender Zugriff auf Teilmatrizen und -vektoren.
- Beliebig wählbare Indizierung.
- Verwendung einer weit verbreiteten Datenstruktur, um problemlos Schnittstellen zu bestehenden Softwarepaketen erstellen zu können.
- Allgemein ist stets eine möglichst effiziente Implementierung sowohl in Bezug auf den Speicher- als auch auf den Rechenzeitbedarf anzustreben.

In der Folge wird zunächst allgemein auf Datenstrukturen für dünn besetzte Vektoren und Matrizen eingegangen. Die in der Implementierung verwendete Datenstruktur sowie entsprechende Algorithmen für die wichtigsten Grundoperationen werden anschließend genauer erläutert. Danach wird die tatsächliche Implementierung in C-XSC detailliert beschrieben. Am Ende dieses Abschnitts werden schließlich einige Tests und Zeitmessungen, inklusive Vergleichen mit anderen Softwarepaketen, vorgenommen. Einige der in der Folge verwendeten Algorithmen und deren Implementierung, vor allem für das Produkt zweier dünn besetzter Matrizen, folgen dabei dem Buch von Davis [35].

3.3.1. Datenstrukturen und Algorithmen

Im Gegensatz zu dicht besetzten Datentypen, bei denen die Einträge in einem Speicherblock der Größe $n \times TSize$ (Vektoren) bzw. $n \times n \times TSize$ (Matrizen) Byte gespeichert werden ($TSize$ ist hierbei die Größe des Grunddatentyps, also z.B. `double`, in Byte), erfordern dünn besetzte Datentypen eine spezielle und deutlich kompliziertere Datenstruktur. Hierfür gibt es eine Vielzahl verschiedener Ansätze, die alle das grundsätzliche Ziel verfolgen, möglichst nur Position und Wert der Nicht-Null-Einträge zu speichern. Ein zweiter wichtiger Gesichtspunkt bei der Wahl der passenden Datenstruktur muss aber auch sein, dass sich bei ihrer Verwendung entsprechende Algorithmen, z.B. für die Matrix-Matrix-Multiplikation, effizient formulieren lassen.

Alle Algorithmen in diesem Abschnitt werden in einer Notation ähnlich C/C++ angegeben, also sehr nahe an der tatsächlichen Implementierung, da die Effizienz der Algorithmen hier wesentlich von einer geschickten Implementierung abhängt. Die Indizierung der Vektor- und Matrixelemente beginnt in den folgenden Erläuterungen daher auch grundsätzlich bei 0. Das Anhängen neuer Elemente an ein Array wird in Anlehnung an die C++-Standardbibliothek mit *push_back* notiert, der Zugriff auf die Elemente einer Datenstruktur (also eines Vektors oder einer Matrix) wird mittels der auch bei C++-Objekten verwendeten Punktnotation kenntlich gemacht.

Vektoren

Der Aufbau einer für dünn besetzte Vektoren geeigneten Datenstruktur ist relativ einfach. Der Vektor wird in Form von zwei Arrays der Länge nnz (Zahl der Nicht-Null-Elemente) gespeichert, wobei ein Array p den Index und ein zweites Array x den Wert des betreffenden Eintrags speichert. Arrayelemente mit dem gleichen Index i, $i = 0, \ldots, nnz-1$ beziehen sich dabei auf das gleiche Element des Vektors (der Vektor hat also an Position $p[i]$ einen Eintrag mit dem Wert $x[i]$). Dieses Vorgehen wird in Beispiel 3.2 demonstriert.

Beispiel 3.2. *Der Vektor*

$$x = (\ 0 \quad 2 \quad 0 \quad 0 \quad 5 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0\ )^T$$

kann unter Ausnutzung seiner spärlichen Besetztheit in den folgenden zwei Arrays abgespeichert werden:

$$\begin{aligned} p &= [\ 1 \quad 4 \quad 6\ ], \\ x &= [\ 2 \quad 5 \quad 1\ ]. \end{aligned}$$

Diese Datenstruktur hat also einen Speicherbedarf von $nnz \times TSize$, zuzüglich $nnz+2$ Integerwerten (die Dimension und die Gesamtzahl der Nichtnullen müssen gesondert gespeichert werden). Ein hierbei (wie bei den meisten anderen dünn besetzten Datenstrukturen) wichtiger zu klärender Punkt ist, ob die Einträge in p und x sortiert sein müssen, d.h. ob gefordert wird, dass $p[i] < p[i+1]$, $i \in \{0, \ldots, nnz-1\}$. Falls dies gefordert wird, können viele Algorithmen einfacher formuliert werden, allerdings muss dann beim Einfügen von neuen Einträgen darauf geachtet werden, dass die Speicherung an der korrekten Position in p und x erfolgt. In der in 3.3.2 beschriebenen Implementierung wird die sortierte Variante verwendet.

Mit dieser Datenstruktur können nun Algorithmen für verschiedene Operationen formuliert werden. Dabei muss unterschieden werden, ob beide Operanden dünn besetzt sind oder einer dicht und einer dünn besetzt ist. Zunächst wird ein Algorithmus für elementweise Berechnungen am Beispiel der Vektoraddition betrachtet. Algorithmus 3.17 dient der Addition zweier dünn besetzter Vektoren.

Hierzu werden die Indizes der beiden Vektoren mit zwei separaten Laufvariablen durchlaufen und jeweils auf Gleichheit geprüft. Sind die Indizes gleich, so wird die Addition der beiden Werte im Ergebnisvektor gespeichert, ansonsten nur der Wert mit dem jeweils kleinsten Index, da der Wert des anderen Vektors für diesen Index gleich 0 ist. Sobald das Ende eines der beiden Vektoren erreicht ist, müssen die verbliebenen Einträge des anderen Vektors noch übernommen werden, weshalb in zwei zusätzlichen Schleifen die restlichen Einträge der beiden Vektoren durchlaufen werden. Dieses Vorgehen lässt sich für alle elementweisen Operationen anwenden, falls die Vektoreinträge aufsteigend nach Index sortiert gespeichert sind.

Algorithmus 3.18 addiert einen dünn und einen dicht besetzten Vektor. Da davon ausgegangen werden kann, dass fast alle Elemente des dicht besetzten Vektors ungleich Null sind, wird der Ergebnisvektor als Kopie des dicht besetzten Vektors initialisiert. Nun müssen nur die Nicht-Null-Elemente des dünn besetzten Vektors durchlaufen und an der

Eingabe : Zwei dünn besetzte Vektoren a und b
Ausgabe : Das Ergebnis c der Berechnung $a + b$

```
//c ist ein Vektor!
c = 0
i = 0
j = 0
while i < a.nnz und j < b.nnz do
    if a.p[i] == b.p[j] then
        c.p.push_back(a.p[i])
        c.x.push_back(a.x[i]+b.x[i])
        i++
        j++

    else if a.p[i] < b.p[j] then
        c.p.push_back(a.p[i])
        c.x.push_back(a.x[i])
        i++

    else if a.p[i] > b.p[j] then
        c.p.push_back(b.p[j])
        c.x.push_back(b.x[j])
        j++


//Ende eines Vektors erreicht, anderen Vektor zu Ende durchlaufen
while i<a.nnz() do
    c.p.push_back(a.p[i])
    c.x.push_back(a.x[i])
    i++
while j<b.nnz() do
    c.p.push_back(b.p[j])
    c.x.push_back(b.x[j])
    j++
```

Algorithmus 3.17: Elementweise Addition zweier dünn besetzter Vektoren

jeweils passenden Stelle zum dicht besetzten Vektor hinzu addiert werden. Dieses Vorgehen lässt sich wiederum auf alle elementweise anzuwendenden Operationen übertragen.

Die Algorithmen 3.19 und 3.20 zeigen die Berechnung des Skalarproduktes. Diese Algorithmen gehen ähnlich vor wie die elementweise operierenden Algorithmen, also mit zwei unabhängigen Laufvariablen. Allerdings ist hier am Ende des Skalarproduktes zweier dünn besetzter Vektoren, nach dem ein Vektor komplett durchlaufen wurde, kein weiteres Durchlaufen des anderen Vektors nötig, da der entsprechende Beitrag zum Skalarprodukt in jedem Falle gleich Null wäre.

```
Eingabe : Ein dünn besetzter Vektor a und ein dicht besetzter Vektor b
Ausgabe : Das Ergebnis c der Berechnung a + b
c = b
for i := 0 to a.nnz-1 do
    c[a.p[i]] += a.x[i]
```

Algorithmus 3.18: Elementweise Addition eines dünn und eines dicht besetzten Vektors

```
Eingabe : Zwei dünn besetzte Vektoren a und b
Ausgabe : Das Ergebnis res des Skalarprodukts a · b
res = 0
i = 0
j = 0
while i < a.nnz und j < b.nnz do
    if a.p[i] == b.p[j] then
        res += a.x[i] * b.x[j]
        i++
        j++

    else if a.p[i] < b.p[j] then
        i++

    else if a.p[i] > b.p[j] then
        j++
```

Algorithmus 3.19: Skalarprodukt zweier dünn besetzter Vektoren

Eingabe : Ein dünn besetzter Vektor a und ein dicht besetzter Vektor b
Ausgabe : Das Ergebnis *res* des Skalarprodukts $a \cdot b$
res = 0;
for *i := 0* **to** *a.nnz-1* **do**
  res += a.x[i] * b[a.p[i]]

Algorithmus 3.20: Skalarprodukt eines dünn und eines dicht besetzten Vektors

Matrizen

Datenstrukturen und Algorithmen für dünn besetzte Matrizen sind in der Regel deutlich komplexer als solche für Vektoren. Der einfachste Ansatz für eine entsprechende Datenstruktur ist eine Erweiterung der oben beschriebenen Vektordatenstruktur. In diesem Fall speichert man die Matrix in drei Arrays der Länge nnz, mit einem Array *row* für den Zeilenindex, einem Array *col* für den Spaltenindex und einem Array x für den eigentlichen Matrixeintrag. Für die Nicht-Null-Elemente einer Matrix A gilt dann also $A_{row[i],col[i]} = x[i]$, $i \in 0, \ldots, nnz - 1$, wie in Beispiel 3.3 demonstriert. Dieses Format wird auch *Triplet-Format* genannt.

Beispiel 3.3. *Die Matrix*

$$\begin{pmatrix} 0 & 0 & 0 & 1.1 & 0 \\ 0 & 2.3 & 0 & 0 & 1.0 \\ 0 & 0 & 0 & 0 & 0 \\ 3.3 & 0 & 2.1 & 0 & 0 \\ 0 & 0 & 6.5 & 0 & 0 \end{pmatrix}$$

kann mittels drei Arrays auf folgende Weise gespeichert werden (Triplet-Format):

$$\begin{array}{lcl} row & = & [\ 0 \quad 1 \quad 1 \quad 3 \quad 3 \quad 4\ ], \\ col & = & [\ 3 \quad 1 \quad 4 \quad 0 \quad 2 \quad 2\ ], \\ x & = & [\ 1.1 \quad 2.3 \quad 1.0 \quad 3.3 \quad 2.1 \quad 6.5\ ]. \end{array}$$

Das Triplet-Format hat einen Speicheraufwand von $nnz \times TSize$ zuzüglich $2nnz + 3$ Integern (zwei Integer für die Dimension der Matrix sowie einer für nnz). In obigem Beispiel ist die Datenstruktur nach Zeilen- und Spalteneinträgen sortiert, dies ist aber keine zwingende Voraussetzung. Problematisch am Triplet-Format ist, dass meist nur schwer performante Algorithmen, z.B. zur Matrix-Matrix-Multiplikation, für dieses Format formuliert werden können. Außerdem benötigt es im Vergleich zu anderen Formaten mehr Speicher, wie im Folgenden zu sehen sein wird.

Neben dem Triplet-Format für allgemeine Matrizen gibt es auch Datenstrukturen, die auf eine bestimmte Struktur der Matrix ausgerichtet sind. So können z.B. Bandmatrizen als ein Array von vollbesetzten Vektoren der Dimension n gespeichert werden, mit jeweils einem Vektor für jedes Band, bzw. mit einer voll besetzten Matrix der Dimension $n \times (l + u)$ bei oberer Bandbreite u und unterer Bandbreite l.

Weiterhin können bei symmetrischen Matrizen nur die Einträge der unteren oder oberen Dreiecksmatrix (inklusive der Diagonalen) gespeichert werden. Dies ist prinzipiell mit jeder beliebigen Datenstruktur möglich, wobei der Umstand, dass es sich um eine symmetrische Matrix handelt, dann über ein Flag festgehalten und bei den zugehörigen Algorithmen berücksichtigt werden kann.

Für die dünn besetzten Datentypen in C-XSC wird allerdings eine allgemein verwendbare Datenstruktur benötigt, welche idealerweise auch in vielen anderen Softwarepaketen Verwendung finden sollte, um die einfache Erstellung von Schnittstellen zu ermöglichen. Daher wird für die C-XSC Datentypen das sogenannte *Compressed Column Storage* (CCS) Format verwendet.

Beim CCS-Format wird die Matrix spaltenweise betrachtet. Die Datenstruktur besteht aus einem Array *ind* mit nnz Elementen, welches die Zeilenindizes der Einträge speichert, sowie einem Array x der Länge nnz, welches die eigentlichen Matrixeinträge speichert, beides analog zum Triplet-Format. Anders als beim Triplet-Format wird aber ein Array p der Länge $n + 1$ (n gibt die Zahl der Spalten der Matrix an) genutzt, in welchem festgehalten wird, zwischen welchen Elementen innerhalb der Arrays *ind* und nnz die Einträge zu einer bestimmten Spalte gespeichert werden. Für die Nicht-Null-Einträge in einer Spalte j werden dabei diejenigen Einträge mit Index i betrachtet, für welche $p[j] \leq i < p[j + 1]$ gilt. Daraus ergibt sich auch, dass der letzte Eintrag des Arrays p gleichzeitig die Zahl der Nicht-Null-Einträge nnz der Matrix angibt. Die Datenstruktur wird in Beispiel 3.4 demonstriert.

Beispiel 3.4. *Die Matrix*

$$\begin{pmatrix} 0 & 0 & 0 & 1.1 & 0 \\ 0 & 2.3 & 0 & 0 & 1.0 \\ 0 & 3.1 & 0 & 0 & 0 \\ 0 & 4.0 & 2.1 & 0 & 0 \\ 0 & 0 & 6.5 & 0 & 0 \end{pmatrix}$$

kann im Compressed Column Storage Format auf folgende Weise abgespeichert werden:

$$\begin{array}{lcl} p & = & [\ 0 \quad 0 \quad\quad\quad 3 \quad\quad 5 \quad 6 \quad 7\ ], \\ ind & = & [\ \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 0 \quad 1 \quad\ ], \\ x & = & [\ \quad 2.3 \quad 3.1 \quad 4.0 \quad 2.1 \quad 6.5 \quad 1.1 \quad 1.0 \quad\ ]. \end{array}$$

Das CCS-Format hat einen Speicheraufwand von $nnz \times TSize$ sowie $nnz + (n + 1) + 2$ Integern (zwei Integer werden zur Speicherung der Dimension benötigt, nnz lässt sich direkt über das letzte Element von p ablesen), benötigt also in der Regel weniger Speicher als das Triplet-Format. Außerdem eignet sich dieses Format deutlich besser für die Formulierung eleganter und effektiver Berechnungsalgorithmen, wie im Folgenden zu sehen sein wird. Analog zum CCS-Format lässt sich auch das *Compressed Row Storage* Format definieren, bei welchem die Rollen von Spalten und Zeilen gegenüber dem CCS-Format vertauscht sind. Das CCS-Format ist allerdings verbreiteter und wird daher auch in der tatsächlichen C-XSC Implementierung verwendet.

Im Folgenden werden nun einige der wichtigsten Algorithmen für dünn besetzte Matrizen auf Basis des CCS-Formats angegeben. Bei allen hier angegebenen Algorithmen wird der Einfachheit halber von quadratischen Matrizen ausgegangen, eine Anpassung auf nicht quadratische Matrizen ist jeweils direkt möglich. Algorithmus 3.21 zeigt zunächst, stellvertretend für alle elementweise auszuführenden Operationen, die Addition zweier dünn besetzter Matrizen.

Dieser Algorithmus ist eine direkte Erweiterung von Algorithmus 3.17 für dünn besetzte Vektoren, bei dem die beiden Matrizen A und B spaltenweise durchlaufen werden und für jede Spalte das Vorgehen aus Algorithmus 3.17 einzeln angewendet wird. Ähnlich verhält es sich beim Algorithmus zur Addition einer dünn und einer dicht besetzten Matrix, der daher hier nicht extra angegeben wird.

Als nächstes wird das Produkt zweier dünn besetzter Matrizen betrachtet (Algorithmus 3.22). Bei der Berechnung des Produktes zweier dünn besetzter Matrizen A und B taucht hier das Problem auf, dass auf die Zeilen von A zugegriffen werden muss (denn das Element in Zeile i und Spalte j der Ergebnismatrix wird über das Skalarprodukt $A_{i\bullet} \cdot B_{\bullet j}$ berechnet), die zugrunde liegende Datenstruktur aber spaltenorientiert ist. Direkte Zugriffe auf die einzelnen Zeilen sind daher sehr aufwändig. Ein erster, naiver Ansatz wäre nun, die Matrix A zu transponieren und dann auf die Spalten der transponierten Matrix zuzugreifen. Dies ist aber nur in Ausnahmefällen eine praktikable Lösung, da der Aufwand für die Transponierung im Allgemeinen zu groß ist. Algorithmus 3.22 geht daher auf andere Art vor.

Hier wird die Ergebnismatrix spaltenweise aufgebaut. Dazu wird ein Hilfsarray *work* der Länge n (Datentyp ist der entsprechende Ergebnisdatentyp), initialisiert mit 0, und ein Hilfsarray w der Länge n von ganzen Zahlen, initialisiert mit -1, benötigt. Die Matrix B wird dann spaltenweise durchlaufen. Für jede Spalte k von B werden die Spalten von A mit Index l durchlaufen, für die $B_{kl} \neq 0$ ist (auf die Matrix A muss auf diese Weise immer nur spalten- und nicht zeilenweise zugegriffen werden).

Das Hilfsarray w dient nun dazu festzuhalten, ob für den entsprechenden Zeilenindex der aktuell betrachteten Spalte der Ergebnismatrix schon ein Zwischenergebnis vorliegt oder nicht. Ist für einen Zeilenindex z und einen Spaltenindex s der Eintrag $w[z] < s$, so liegt noch kein Zwischenergebnis vor, ansonsten wurde in vorherigen Schleifendurchläufen schon ein Beitrag zum jeweiligen Eintrag der Ergebnismatrix berechnet. Die Zwischenergebnisse für die aktuell betrachtete Spalte der Ergebnismatrix werden im Hilfsarrray *work* gespeichert. Entsprechend dem Eintrag im Hilfsarray w wird dann der neue Beitrag für die aktuell betrachteten Indizes zum jeweiligen Eintrag im *work* Array hinzu addiert oder es wird ein neuer Beitrag initialisiert.

Die Nutzung des Hilfsarrays w verhindert, dass das *work* Array für jeden Durchlauf der äußersten for-Schleife (welche über die Spalten der Ergebnismatrix läuft) wieder mit 0 initialisiert werden muss. Diese Initialisierung würde einen Aufwand $\mathcal{O}(n)$ zu jedem Schleifendurchlauf hinzufügen. Ein entsprechender Algorithmus für das Produkt einer dicht und einer dünn besetzten Matrix ist direkt aus diesem Vorgehen ableitbar und wird daher hier nicht angegeben.

Zum Abschluss dieses Abschnitts betrachten wir nun noch das Matrix-Vektor Produkt $A \cdot v$ zwischen einer dünn besetzten Matrix A und einem dünn besetzten Vektor v

```
Eingabe : Zwei dünn besetzte Matrizen A und B der Dimension n × n im
          CCS-Format
Ausgabe : Die dünn besetzte Matrix C = A + B im CCS-Format
nnz = 0
for j=0 to n-1 do
    k = A.p[j]
    l = B.p[j]
    while k<A.p[j+1] und l<B.p[j+1] do
        if A.ind[k] == B.ind[l] then
            C.ind.push_back(A.ind[k])
            C.x.push_back(A.x[k] + B.x[l])
            k++
            l++
        else if A.ind[k] < B.ind[l] then
            C.ind.push_back(A.ind[k])
            C.x.push_back(A.x[k])
            k++
        else
            C.ind.push_back(B.ind[l])
            C.x.push_back(B.x[l])
            l++
        nnz++
    while k<A.p[j+1] do
        C.ind.push_back(A.ind[k])
        C.x.push_back(A.x[k])
        nnz++
        k++
    while l<B.p[j+1] do
        C.ind.push_back(B.ind[l])
        C.x.push_back(B.x[l])
        nnz++
        l++
    C.p[j+1] = nnz
```

Algorithmus 3.21: Addition zweier dünn besetzter Matrizen im CCS-Format

```
Eingabe : Zwei dünn besetzte Matrizen A und B der Dimension n × n im
          CCS-Format
Ausgabe : Die dünn besetzte Matrix C = AB im CCS-Format
real work(n) = 0.0
int w(n) = -1
nnz = 0
for i=0 to n-1 do
    for k=B.p[i] to B.p[i+1]-1 do
        for l=A.p[B.ind[k]] to A.p[B.ind[k]+1]-1 do
            if w[A.ind[l]] < i then
                w[A.ind[l]] = i
                C.ind.push_back(A.ind[l])
                work[A.ind[l]] = A.x[l] * B.x[k]
                nnz++
            else
                work[A.ind[l]] += A.x[l] * B.x[k]
    for j=C.p[i] to nnz do
        C.x.push_back(work[C.ind[j]])
    C.p[i+1] = nnz
```

Algorithmus 3.22: Produkt zweier dünn besetzter Matrizen im CCS-Format

```
Eingabe : Eine dünn besetzte Matrix A der Dimension n × n im CCS-Format und
          ein dünn besetzter Vektor v
Ausgabe : Der dünn besetzte Vektor x = Av
real work(n) = 0.0
for i=0 to v.nnz-1 do
    for k=A.p[v.p[i]] to A.p[v.p[i]+1]-1 do
        work[A.ind[k]] += A.x[k] * v.x[i]
for i=0 to A.m-1 do
    if work[i] != 0.0 then
        x[i+1] = work[i]
```

Algorithmus 3.23: Produkt einer dünn besetzten Matrix im CCS-Format und eines dünn besetzten Vektors v

(Algorithmus 3.23).

Es wird wieder ein Hilfsarray *work* benötigt. Die Matrix A wird spaltenweise durchlaufen und der entsprechende Teil des jeweiligen Skalarproduktes wird berechnet und im Hilfsarray zwischengespeichert. Nach Durchlauf aller Spalten enthält das *work*-Array dann den Ergebnisvektor x und muss nur noch in ein dünn besetztes Format übertragen werden. Die Fälle mit dicht besetzter Matrix bzw. dicht besetztem Vektor sind ähnlich umzusetzen.

3.3.2. Implementierung

In diesem Abschnitt soll nun die tatsächliche Implementierung der Datentypen für dünn besetzte Vektoren und Matrizen in C-XSC erläutert werden. Dabei wird zunächst der grundsätzliche Aufbau der Klassen, auch im Vergleich zu den Datentypen für dicht besetzte Vektoren und Matrizen, erklärt. Danach folgen Überlegungen zum Einsatz der *Standard Template Library* (STL) und der Nutzung von Templates in der Implementierung.

Grundlegende Überlegungen

Eine wesentliche Anforderung bei der Erstellung der dünn besetzten Datentypen ist, eine möglichst große Ähnlichkeit zu den dicht besetzten Typen bezüglich Aufbau und Schnittstellen zu erreichen. Dies hat zum einen den Vorteil, dass Benutzer, die schon mit C-XSC vertraut sind, die neuen Datentypen direkt problemlos verwenden können (von einigen Besonderheiten und Fallstricken abgesehen, die am Ende dieses Abschnitts erläutert werden). Zum anderen ermöglicht es auch die weitgehend problemlose Erstellung von Templates, welche dann sowohl für dicht besetzte als auch für dünn besetzte Datentypen ausgeprägt werden können. Bei der Nutzung in Templates ist allerdings darauf zu achten, dass, bedingt durch die zugrunde liegende Datenstruktur, für Berechnungen mit dünn besetzten Matrizen zumeist speziell angepasste Algorithmen benötigt werden und daher die Benutzung der dünn besetzten Datentypen mit Algorithmen, die eigentlich für dicht besetzte Datentypen konzipiert sind, oftmals zu enormen Geschwindigkeitseinbußen führen kann.

Bei der Konstruktion der Klassen tauchen zwei wichtige Fragestellungen auf: Zum einen, ob es es sinnvoll ist, eine Vererbungshierarchie mit einer (abstrakten) Oberklasse für alle Matrizen/Vektoren einzuführen, zum anderen in welcher Form und in welchem Umfang Templates bei der Implementierung benutzt werden können. Blicken wir zunächst auf die Frage nach einer Vererbungshierarchie für alle Matrix- und Vektordatentypen.

Der Grundgedanke dabei wäre, jeweils eine Oberklasse für alle Matrizen und für alle Vektoren zu definieren. Da diese Oberklasse dann keine tatsächliche Matrix repräsentieren könnte (denn es wäre ja nicht klar, welchen Datentyp die Einträge hätten), müsste es sich um eine abstrakte Klasse handeln, welche nur mit rein virtuellen Funktionen eine Schnittstelle für alle Matrizen vorgibt. Gegen dieses Vorgehen sprechen allerdings einige Punkte.

Zunächst besteht die Schnittstelle der Matrix- und Vektorklassen überwiegend aus überladenen Operatoren. Viele dieser Operatoren werden aber nicht als Member-, sondern als `friend`- oder einfache Funktionen definiert. Diese können daher auch nicht als rein virtuelle Funktionen in die Schnittstellenvorgabe mit aufgenommen werden. Allgemeine Operatoren für die Oberklasse einzufügen ist ebenfalls nicht möglich, da die eigentlichen Einträge aufgrund der unterschiedlichen Datentypen der Unterklassen, welche den eigentlich entscheidenden Unterschied zwischen den verschiedenen Klassen ausmachen, nicht in der Oberklasse auftauchen. Weiterhin würde ein solches Vorgehen auch ein komplettes Umschreiben des bestehenden, gut ausgetesteten und stabilen Quellcodes für die dicht besetzten Datentypen erfordern.

Aus diesen Gründen wurde auf die Definition einer (abstrakten) Oberklasse für alle Matrix- und Vektorklassen verzichtet. Stattdessen wird analog zu den dicht besetzten Typen vorgegangen: Es gibt je vier unabhängige Matrix- und Vektorklassen für jeden der vier Grunddatentypen. Diese wurden analog zu den dicht besetzten Typen benannt, die dünne Besetztheit wird durch ein vorangestelltes `s` im Namen des Datentyps verdeutlicht. Die neuen Datentypen haben also folgende Bezeichnungen:

- Matrizen: `srmatrix`, `simatrix`, `scmatrix`, `scimatrix`
- Vektoren: `srvector`, `sivector`, `scvector`, `scivector`

Analog zu den dicht besetzten Typen gibt es auch eigene, intern verwendete Typen für *Slices* und *Subvektoren*, also mittels `()`- oder `[]`-Operator aus einer Matrix oder einem Vektor herausgetrennte Bereiche. Auf diese wird später in diesem Abschnitt genauer eingegangen.

Nutzung von Templates

Der nächste wichtige Punkt ist die Frage nach der Benutzung von Templates. Offensichtlich sind die verschiedenen Matrix- und Vektortypen, egal ob dünn oder dicht besetzt, vom Grundaufbau her sehr ähnlich, d.h. sie stellen in weiten Teilen die gleichen Operatoren zur Verfügung, in welchen dann auch oftmals (durch die Nutzung der diversen überladenen Operatoren der vier Grunddatentypen) identischer Quellcode verwendet wird. Aus diesem Grund bietet sich die Benutzung von Templates an, wobei die Erstellung je einer einzigen Template-Klasse für Vektoren und Matrizen, welche dann für alle Grunddatentypen ausgeprägt wird, die offensichtlichste Möglichkeit darstellt.

Bei der Benutzung einer Template-Klasse würde man allerdings wiederum auf einige Probleme stoßen. Auch hier muss bedacht werden, dass viele Operationen und auch Funktionen (wie z.B. `Inf` oder `Re`) nicht als Klassenmember definiert sind. Diese Funktionen müssten also in Template-Funktionen umgewandelt werden. Da die Definitionen aus Performancegründen unbedingt im Header stehen müssen, um Inlining zu erlauben, würde dies allerdings bedeuten, dass diese Templates (versehentlich) für beliebige Datentypen und nicht nur für die C-XSC Grunddatentypen ausgeprägt werden können. Dadurch kann es zu falscher Anwendung durch und verwirrenden Compiler-Fehlermeldungen für den Benutzer kommen (mittels Funktionalitäten aus der Boost-Bibliothek ließe sich dem

zwar entgegen wirken, allerdings ist es von jeher ein grundlegendes Ziel, für die C-XSC Kernbibliothek nicht die Verfügbarkeit externer Bibliotheken zwingend vorauszusetzen).

Aus diesem Grund, und insbesondere um ein einheitliches Vorgehen und somit leichtere Wartbarkeit innerhalb der C-XSC Bibliothek sicherzustellen, wird stattdessen ein ähnlicher Ansatz wie bei den dicht besetzten Datentypen verfolgt: Die Matrix- und Vektorklassen selbst sind keine Templates, allerdings rufen sie innerhalb der verschiedenen Operatoren und Funktionen Template-Funktionen auf, welche die eigentliche Berechnung durchführen. Im Normalfall wird der zusätzliche Funktionsaufruf sich nicht auf die Laufzeit auswirken, da er vom Compiler bei Benutzung von Compileroptimierungen in den allermeisten Fällen inline kompiliert wird. Um das erläuterte Vorgehen zu verdeutlichen, betrachten wir als Beispiel den Operator zur Addition eines dünn und eines dicht besetzten Intervallvektors. In Listing 3.22 ist eine übliche Implementierung ohne Benutzung von Templates gegeben.

```
1  inline ivector operator+(const sivector& v1, const ivector& v2) {
2    ivector res(v2); //res mit den Werten von v2 initialisieren
3    int lb = Lb(res);
4
5    for(int i=0 ; i<v1.get_nnz() ; i++)
6      res[v1.p[i]+lb] += v1.x[i];
7
8    return res;
9  }
```

Listing 3.22: Operator zur Addition eines `sivector`s und eines `ivector`s ohne Template-Nutzung

Die Implementierung dieses Operators ist, durch die Nutzung des jeweiligen überladenen Operators `+=`, offensichtlich unabhängig vom Grunddatentyp der beiden Vektoren `v1` und `v2`. Entscheidend für die Implementierung ist nur, dass `v1` ein dünn und `v2` ein dicht besetzter Vektor ist. Diese Beobachtung gilt analog für nahezu alle Operatoren und Funktionen. Es ist also im hier betrachteten Beispiel möglich, eine Template-Funktion zum Ausführen der Berechnung für alle Operatoren zu definieren, bei denen ein dünn und ein dicht besetzter Vektor addiert werden. Eine solche Template-Funktion ist in Listing 3.23 gegeben.

```
1  template<class Tx, class Ty, class Tres>
2  inline Tres spf_vv_add(const Tx& v1, const Ty& v2)
3  #if(CXSC_INDEX_CHECK)
4          throw(OP_WITH_WRONG_DIM)
5  #else
6          throw()
7  #endif
8  {
9  #if(CXSC_INDEX_CHECK)
10   if(v1.n!=VecLen(v2)) cxscthrow(
11     OP_WITH_WRONG_DIM("operator+(const␣"+nameof(v1)+
12                       "␣&,␣const␣"+nameof(v2)+"␣&)")
13   );
14 #endif
```

```
15    Tres res(v2);
16    int lb = Lb(res);
17
18    for(int i=0 ; i<v1.get_nnz() ; i++)
19      res[v1.p[i]+lb] += v1.x[i];
20
21    return res;
22  }
```

Listing 3.23: Template-Funktion zur Addition eines dünn und eines dicht besetzten Vektors

In dieser Template-Funktion wurde auch Code zur optionalen Prüfung der Länge der beiden Vektoren ergänzt. C-XSC ermöglicht dem Benutzer, solche Dimensionsprüfungen durch Setzen der Präprozessorvariable `CXSC_INDEX_CHECK` für alle Matrix- und Vektoroperationen beim Kompilieren zu aktivieren. Tritt ein Fehler auf, so wird dann eine entsprechende Exception geworfen. Diese Prüfungen sind vor allem während der Entwicklung von C-XSC Programmen nützlich, werden aber aus Performancegründen standardmäßig nicht aktiviert. Ansonsten entspricht der eigentliche Quellcode dem Code aus Listing 3.22 in einer Template-Version. Es werden dabei drei Template-Parameter benötigt: Je einer für den linken und rechten Operanden sowie einer für den Rückgabetyp.

Der Name der Template-Funktion (und aller anderen Template-Berechnungsfunktionen) setzt sich nach einem bestimmten Muster zusammen. Der erste Teil des Namens gibt die zugrunde liegende Datenstruktur von linkem und rechtem Operanden an. Möglich sind dabei:

- `sp`: Es handelt sich um eine dünn besetzte Datenstruktur.
- `f`: Es handelt sich um eine dicht besetzte Datenstruktur.
- `sl`: Es handelt sich um einen Ausschnitt aus einer dünn besetzten Datenstruktur („Slice“). Auf solche Ausschnitte wird im weiteren Verlauf dieses Abschnitts noch genauer eingegangen.

Im betrachteten Beispiel ist der linke Operand dünn und der rechte Operand dicht besetzt, der Funktionsname beginnt daher mit `spf`. Nach einem Unterstrich folgt eine Angabe zur Art der Operanden. Mögliche Angaben hierbei sind:

- `m`: Es handelt sich um eine Matrix.
- `v`: Es handelt sich um einen Vektor.
- `s`: Es handelt sich um ein Skalar.

Im Beispiel werden zwei Vektoren addiert, der entsprechende Teil des Funktionsnamens lautet also `vv`. Schließlich folgt nach einem weiteren Unterstrich noch die Bezeichnung der zugrunde liegenden Operation. Dies können z.B. folgende Angaben sein:

- `add`: Addition.

- `sub`: Subtraktion.
- `mult`: Multiplikation bzw. Skalarprodukt.
- `assign`: Zuweisung.
- `addassign`: Addition und gleichzeitige Zuweisung.
- ...

Im Beispiel ist der letzte Teil des Bezeichners also `add` und es ergibt sich insgesamt als Name für die Berechnungsfunktion zur Berechnung der Addition eines dünn und eines dicht besetzten Vektors: `spf_vv_add`. Einige der Operationen bzw. Funktionen benötigen nur einen Operanden bzw. Parameter, in einem solchen Fall wird der Name der Berechnungsfunktion entsprechend angepasst, also z.B. `sp_v_input` für den Eingabeoperator eins dünn besetzten Vektors.

Mit diesen Template-Berechnungsfunktionen können die einzelnen Operatoren und Funktionen nun wie in Listing 3.24 definiert werden. Zu beachten ist, dass die Template-Parameter beim Aufruf immer direkt angegeben werden müssen, da der Rückgabetyp, welcher auch ein Template-Parameter ist, vom Compiler nicht aus dem Zusammenhang des jeweiligen Aufrufs heraus hergeleitet werden kann.

```
1 inline ivector operator+(const sivector& v1, const ivector& v2) {
2   return spf_vv_add<sivector,sivector,ivector>(v1,v2);
3 }
```

Listing 3.24: Operator zur Addition eines `sivector` und eines `ivector` mit Template-Nutzung

Auf diese Weise können also alle Operatoren und viele Funktionen der neuen Datentypen für dünn besetzte Vektoren und Matrizen mit Hilfe von Template-Berechnungsfunktionen definiert werden. Der Quellcode für die eigentlichen Berechnungen kann so vergleichsweise klein gehalten und innerhalb von nur zwei Dateien zentral definiert werden (dies sind die Dateien `sparsevector.inl` und `sparsematrix.inl`). Dies vereinfacht die Fehlerversuche und Erweiterung für diese neuen Datentypen enorm und führt auch ohne Verwendung von Template-Klassen zu einem, in Anbetracht der Komplexität der neuen Datentypen, schlanken und übersichtlichen Quellcode.

Aufbau der Klassen

Im Folgenden soll nun der interne Aufbau der Klassen für dünn besetzte Vektoren bzw. Matrizen näher betrachtet werden. Alle Datentypen für dünn besetzte Vektoren haben folgenden Datenmember:

- `int n`: Die Dimension des Vektors.
- `int lb, ub`: Untere und obere Indizierungsgrenze (s.u.).
- `vector<int> p`: Analog zum in Abschnitt 3.3.1 beschriebenen Array p.

- `vector<T> x`: Analog zum in Abschnitt 3.3.1 beschriebenen Array x (`T` ist hier der Grunddatentyp des Vektors, also z.B. `interval` für `sivector`).

Die Datentypen für dünn besetzte Matrizen haben folgende Datenmember:

- `int m, n`: Die Dimension der Matrix.
- `int lb1, lb2, ub1, ub2`: Untere und obere Indizierungsgrenzen (s.u.).
- `vector<int> p`: Analog zum in Abschnitt 3.3.1 beschriebenen Array p.
- `vector<int> ind`: Analog zum in Abschnitt 3.3.1 beschriebenen Array *ind*.
- `vector<T> x`: Analog zum in Abschnitt 3.3.1 beschriebenen Array x (`T` ist hier der Grunddatentyp der Matrix, also z.B. `interval` für `simatrix`).

Die interne Indizierung ist dabei, unabhängig von der für das jeweilige Objekt gewählten Indizierung, Null-basiert. Anstatt von Arrays wird der Datentyp `vector` aus der STL zur Speicherung der Indizes und der Werte genutzt. Dies hat den Vorteil, dass auch die in der Regel sehr robusten und performanten Algorithmen (z.B. zur Sortierung eines `vector`-Objektes) aus der STL genutzt werden können. Vor allem aber wird die Speicherverwaltung komplett im Datentyp `vector` geregelt, wodurch sich die Fehleranfälligkeit vermindert und die Wartbarkeit des Codes deutlich verbessert.

Ein Argument gegen die Verwendung des Datentyps `vector` liegt in der potentiellen Geschwindigkeitseinbuße: Durch die Nutzung objektorientierter Konzepte kommt es zu vielen zusätzlichen Funktionsaufrufen, z.B. bei einer Anweisung wie `p[1]=1;`, welche bei Verwendung eines Arrays für `p` einem direkten Zugriff auf eine Speicherstelle entsprechend würde, bei Verwendung eines STL-`vector`s aber einen impliziten Funktionsaufruf des `operator[]` zur Folge hat. Allerdings wird dieser Effekt bei Verwendung von Compiler-Optimierungen durch Inlining in weiten Teilen aufgehoben, so dass die Vorteile diesen Nachteil deutlich überwiegen.

Zum Anlegen der dünn besetzten Vektoren und Matrizen stehen mehrere Konstruktoren zur Auswahl. Für dünn besetzte Vektoren sind dabei im Wesentlichen folgende Konstruktoren vorgesehen (hier am Beispiel der Klasse `srvector` für reelle Vektoren gezeigt, die anderen Klassen bieten Konstruktoren analog zu diesen):

- `srvector()`: Standardkonstruktor, legt einen Vektor der Länge 0 an.
- `explicit srvector(const int n)`: Konstruktor zum Anlegen eines Vektors der Länge `n`. Weiterhin wird Speicher für eine Besetztheit von 10% reserviert.
- `srvector(const int n, const int nnz)`: Konstruktor zum Anlegen eines Vektors der Länge `n`, wobei Speicher für `nnz` Nicht-Null-Elemente reserviert wird.
- `srvector(const rvector& v)`: Konstruktor zum Erzeugen eines dünn besetzten Vektors aus einem voll besetzten. Es werden nur die Nicht-Nullen aus `v` übernommen.

- `srvector(const int n, const int nnz, const intvector& index, const rvector& values)`: Erzeugt einen dünn besetzten Vektor aus schon vorhandenen Daten entsprechend der internen Datenstruktur (`index` und `values`) der Länge `n` mit `nnz` Nicht-Nullen (d.h. `index` und `values` müssen die Länge `nnz` haben).

- `srvector(const int n, const int nnz, const int* index, const real* values)`: Funktioniert analog zum vorherigen Konstruktor, verwendet aber Arrays statt C-XSC Vektoren.

Für dünn besetzte Matrizen stehen im Wesentlichen folgende Konstruktoren zur Auswahl (wiederum erläutert am Beispiel der Klasse `srmatrix`, also für reelle Matrizen):

- `srmatrix()`: Standardkonstruktor, legt eine Matrix der Dimension 0×0 an.

- `srmatrix(const int r, const int c)`: Legt eine leere Matrix mit `r` Zeilen und `c` Spalten an. Dabei wird Speicher für $2(r + c)$ Elemente reserviert.

- `srmatrix(const int r, const int c, const int e)`: Legt eine leere Matrix mit `r` Zeilen und `c` Spalten an. Dabei wird Speicher für `e` Elemente reserviert.

- `srmatrix(const int m, const int n, const int nnz, const intvector& rows, const intvector& cols, const rvector& values, const enum STORAGE_TYPE t = triplet)`: Legt eine Matrix der Dimension $m \times n$ mit `nnz` Nicht-Nullen an. Die Matrixeinträge werden mittels der Daten in den Arrays `rows`, `cols` und `values` belegt, wobei der Parameter `t` angibt, wie diese Arrays zu interpretieren sind. Mögliche Werte sind dabei `triplet` (dies ist auch der Standardwert), `compressed_row` und `compressed_column`.

- `srmatrix(const int m, const int n, const int nnz, const int* rows, const int* cols, const real* values, const enum STORAGE_TYPE t = triplet)`: Funktioniert analog zum vorherigen Konstruktor, nur dass statt den Datentypen `intvector` und `rvector` entsprechende Arrays benutzt werden.

- `srmatrix(const rmatrix& A)`: Erzeugt eine dünn besetzte Matrix aus einer voll besetzten Matrix. Dabei werden nur die Nicht-Null-Einträge übernommen.

- `srmatrix(const int m, const int n, const rmatrix& A)`:
  Konstruktor für Bandmatrizen. Erzeugt wird eine Bandmatrix der Dimension $m \times n$, deren Bänder durch die Spalten der Matrix `A` bestimmt werden. Die Spaltenindizes der Matrix `A` müssen dabei so gesetzt werden (Erläuterungen zum Setzen des Index folgen im Verlauf dieses Abschnitts), dass die Spalten, aus welchen Bänder unter der Hauptdiagonalen erzeugt werden sollen, negative Spaltenindizes haben, die Spalte für die Hauptdiagonale den Index 0, und Spalten für Bänder über der Hauptdiagonalen positive Indizes. Die Zeilenindizes von `A` sollten von 1 bis `m` laufen. Beim Erzeugen werden dann für das i-te Band unter der Hauptdiagonalen

die Elemente mit Zeilenindex zwischen $i+1$ und n der Spalte $-i$ von `A` verwendet, für das j-te Band über der Hauptdiagonalen die Elemente mit Zeilenindex zwischen i und $n-j$ der Spalte j von `A`.

Weiterhin sind sowohl für Vektoren als auch Matrizen entsprechende Copy- und Konvertierungskonstruktoren (z.B. zum Erzeugen eines `sivector` aus einem `srvector`) vorhanden, die hier nicht extra aufgeführt werden.

Die dünn besetzten Datentypen bieten im Wesentlichen die gleichen Operatoren wie die dicht besetzten Typen. Die vorhandenen Operatoren entsprechen also denen in Tabelle 2.4 und Tabelle 2.5 aus Abschnitt 2.3.

Weiterhin sind Aus- und Eingabeoperatoren definiert. Um zwischen dünn und dicht besetzter Ein-/Ausgabe unterscheiden zu können, werden folgende IO-Flags eingeführt:

- `FullInOut`: Ein- und Ausgabe als dicht besetzte Matrix bzw. dicht besetzter Vektor. Dies ist die Standardeinstellung.

- `SparseInOut`: Ein- und Ausgabe als dünn besetzte Matrix bzw. dünn besetzter Vektor (Ausgabeformat s.u.). Dieses Flag wirkt sich nur auf die dünn besetzten Datentypen aus.

- `MatrixMarketInOut`: Ein- und Ausgabe im Matrix-Market-Format. Dieses Flag wirkt sich nur auf Matrixdatentypen (sowohl dicht als auch dünn besetzt) aus. Das Matrix-Market Format wird dabei um die Datentypen `interval` und `cinterval` erweitert, damit prinzipiell jeder C-XSC Datentyp unterstützt werden kann. Das Format schreibt allerdings vor, dass die Werte dezimal gespeichert werden, d.h. es können Konversionsfehler auftreten. Bei der Intervall Ein- und Ausgabe wird daher entsprechend gerundet. Alternativ ist es bei ausschließlicher Verwendung von C-XSC auch möglich, mittels des IO-Flags `Hex` von C-XSC auf hexadezimale Ausgabe umzuschalten und so Konversionsfehler komplett zu umgehen. Hauptanwendungsgebiet dieser Funktionalität ist aber im Allgemeinen das Einlesen von (Punkt-)Testdateien aus dem Matrix-Market.

Das Matrix-Market-Format ist sehr weit verbreitet. Mit Hilfe der frei erhältlichen C-Bibliothek BeBOP Sparse Matrix Converter kann außerdem zwischen den verbreitetsten Dateiformaten zur Weitergabe dünn besetzter Dateien (Matrix Market, Harwell-Boeing, Matlab) konvertiert werden. Dadurch sollten nahezu alle gebräuchlichen Sammlungen von Testmatrizen mit C-XSC ohne größere Umstände nutzbar sein.

Listing 3.25 zeigt ein Beispiel für die Verwendung der IO-Flags. Bei der dünn besetzten Ausgabe wird zunächst die Dimension, dann in der nächsten Zeile die Zahl der Nicht-Null-Elemente und schließlich jeweils untereinander die Arrays der dünn besetzten Datenstrukturen ausgegeben. Eine detaillierte Beschreibung des Matrix-Market Formats wurde von Boisvert, Pozo und Remington formuliert [24]. Einige Beispiele für Matrix-Market Dateien, sowohl mit als auch ohne die C-XSC Erweiterungen für Intervalle, finden sich in Anhang A.

```
1  srmatrix A(5,5);
```

```
2  A = Id(A); //Einheitsmatrix
3
4  cout << A << endl; //dicht besetzte Ausgabe
5  cout << SparseInOut << A << endl; //duenn besetzte Ausgabe
6  cout << MatrixMarketInOut << A << endl; //Ausgabe im Matrix-Market Format
```

Listing 3.25: Beispiel zur Verwendung der IO-Flags

Die Ausgabe dieses Beispielprogramms ist:

```
1    1.000000    0.000000    0.000000    0.000000    0.000000
2    0.000000    1.000000    0.000000    0.000000    0.000000
3    0.000000    0.000000    1.000000    0.000000    0.000000
4    0.000000    0.000000    0.000000    1.000000    0.000000
5    0.000000    0.000000    0.000000    0.000000    1.000000
6
7  5 5
8  5
9  0 1 2 3 4 5
10 0 1 2 3 4
11   1.000000    1.000000    1.000000    1.000000    1.000000
12
13 %%MatrixMarket matrix coordinate real general
14 %Generated by C-XSC
15 5 5 5
16 1 1    1.000000
17 2 2    1.000000
18 3 3    1.000000
19 4 4    1.000000
20 5 5    1.000000
```

Wie bei den dicht besetzten Datentypen gibt es außerdem noch einen Operator für Permutationen, ähnlich wie bei Matlab. Mit Permutationsmatrizen P und Q, welche pro Zeile und Spalte jeweils nur einen Eintrag 1 und sonst nur Nulleinträge haben, können die Zeilen und Spalten einer Matrix A über PAQ permutiert werden. Die direkte Berechnung auf diese Weise ist aber ineffektiv. Mit dem Permutationsoperator und Matrizen `P` und `Q` vom Typ `intmatrix` können Permutationen von dünn besetzten Matrizen bzw. Vektoren effektiver ausgeführt werden, da die jeweiligen Vertauschungen direkt vorgenommen werden.

Weiterhin können statt der Permutationsmatrizen auch Permutationsvektoren p und q verwendet werden. Der Permutationsvektor p zu einer Permutationmatrix P besitzt die Einträge $p_i = j$ für alle $P_{ij} = 1$. Die Nulleinträge der Permutationsmatrix werden so also vermieden (dies ist vor allem deshalb von Bedeutung, da es in C-XSC zur Zeit keine dünn besetzte Version von `intmatrix` gibt). Der Permutationsoperator kann dann analog mit Hilfe solcher Permutationsvektoren vom Typ `intvector` verwendet werden. Listing 3.26 zeigt Beispielcode zur Verwendung des Operators.

```
1  srmatrix A,B;
2  srvector a,b;
3  intmatrix P,Q; //Permutationsmatrizen
4  intvector p,q; //Permutationsvektoren
5
```

```
6   //... Matrizen/Vektoren fuellen ...
7
8   //Permutation ueber direkte Berechnung
9   B = P*A*Q;
10  //Gleiche Permutation mit Operator (effektiver)
11  B = A(P,Q);
12  //Gleiche Permutation mit Operator und Permutationsvektoren
13  //(noch effektiver)
14  B = A(p,q);
15
16  //Jeweils wie oben, aber nur Zeilenpermutation
17  B = P*Q;
18  B = A(P);
19  B = A(p);
20
21  //Vektor permutieren, direkt
22  b = P*a;
23  //Mit Operator
24  b = A(P);
25  //Mit Operator und Permutationsvektoren
26  b = a(p);
```

Listing 3.26: Beispiel zur Verwendung des Permutations-Operators

Ausschneiden von Teilmatrizen und -vektoren

Weiterhin stellen die dünn besetzten Datentypen über die Operatoren `[]` und `()` auch die Möglichkeit zum Zugriff auf einzelne Elemente, zum Ausschneiden einzelner Spalten und Zeilen oder sogar kompletter Teilmatrizen, sowie zum Ausschneiden von Teilen aus einem Vektor bereit. Das Besondere hierbei ist, dass diese Ausschnitte sowohl lesenden als auch schreibenden Zugriff erlauben, d.h. eine Zuweisung zu einem solchen Ausschnitt verändert den entsprechenden Bereich in der Originalmatrix.

Zunächst wird der Zugriff auf einzelne Elemente einer dünn besetzten Matrix oder eines dünn besetzten Vektors betrachtet. Bei den dicht besetzten Datentypen geschieht dieser Zugriff über den Eckige-Klammern-Operator, also z.B. im Stil von `A[1][1]` für Matrizen und `x[1]` für Vektoren. Dies ist auch bei den dünn besetzten Datentypen möglich, allerdings muss folgendes beachtet werden: Da der Zugriff auf ein einzelnes Element auch schreibend erfolgen kann, muss der entsprechende `operator[]` eine *Referenz* auf das jeweilige Element aus der Originaldatenstruktur zurückgeben. Anders als bei den dicht besetzten Datentypen speichern die dünn besetzten Datentypen aber nicht jedes Element explizit ab. Für den Fall, dass einem Element, welches bisher nicht explizit gespeichert wurde, ein neuer Wert zugewiesen werden soll, muss der `operator[]` daher zunächst dieses Element explizit mit dem Wert 0 in die Datenstruktur einfügen, um eine Referenz darauf zurückgeben zu können. Bei einem nur lesenden Zugriff entsteht hier das Problem, dass die Datenstruktur unnötigerweise um ein Element vergrößert wird. Werden mehrere Elemente auf diese Weise angesprochen, z.B. innerhalb einer Schleife, kommt es so also zu einer unnötigen Vergrößerung des Speicherbedarfs und einer Verlangsamung von evtl. noch folgenden Berechnungen mit der betreffenden Datenstruktur. Weiterhin

ist das Einfügen eines zusätzlichen Elementes relativ aufwändig, da hierzu große Teile der Datenstruktur neu angeordnet werden müssen. Deshalb wird für die dünn besetzten Datenstrukturen zwischen lesendem und schreibendem Zugriff unterschieden.

Bei Verwendung nur eines Operators ist dies mit den Sprachfeatures von C++ nicht möglich. Zwar lässt sich zwischen dem Zugriff auf ein konstantes (also mit dem Schlüsselwort `const` deklariertes) Objekt, auf welches per Definition schon nur lesend zugegriffen werden kann, und auf ein nicht konstantes Objekt unterscheiden, allerdings ist es möglich (und in der Tat auch ein häufiger Anwendungsfall), dass auf ein nicht konstantes Objekt nur lesend zugegriffen werden soll. Aus diesem Grund wurde für die dünn besetzten Datentypen in C-XSC festgelegt, dass grundsätzlich der `operator[]` für schreibenden Zugriff verwendet wird, d.h. er liefert eine Referenz auf das betreffende Element zurück und verhält sich damit konsistent zum entsprechenden Operator der dicht besetzten Datentypen, und der `operator()` grundsätzlich für lesenden Zugriff, d.h. eine Kopie des betreffenden Elements wird zurückgeliefert (oder 0 falls das Element nicht explizit gespeichert wurde, wodurch die weiter oben angesprochenen Probleme also vermieden werden). Listing 3.27 zeigt ein entsprechendes Beispiel.

```
1  srmatrix A(10,10);
2
3  //... Diagonale von A wird gefuellt ...
4
5  cout << A[1][1] << endl; //Element existiert,
6                           //Referenzrueckgabe
7  cout << A[2][1] << endl; //Element existiert nicht,
8                           //wird angelegt, Referenzrueckgabe
9  cout << A(3,1) << endl;  //Element existiert nicht
10                          //und wird nicht angelegt (0 wird zurueckgegeben)
11 A[4][2] = 2.0;           //Element wird angelegt und gesetzt
12 A(5,2) = 3.0;            //Fehler! ()-operator nur fuer Lesezugriff
```

Listing 3.27: Beispiel zum Elementzugriff bei dünn besetzten Datentypen

Das Ausschneiden von Teilen aus dünn besetzten Vektoren und Matrizen benutzt ebenfalls die überladenen Operatoren `()` und `[]`, allerdings mit angepassten Parametern. Um den schreibenden Zugriff auf diese Teile zu ermöglichen sind dabei jeweils Hilfsklassen erforderlich, welche den Bezug zur Originalmatrix bzw. zum Originalvektor herstellen (bei den dicht besetzten Datentypen wird ebenfalls mit solchen Hilfsklassen gearbeitet). Zunächst wird das Ausschneiden eines Teilbereichs aus einem dünn besetzten Vektor betrachtet.

Hierfür werden die Hilfsklassen `srvector_slice`, `sivector_slice`, `scvector_slice` und `scivector_slice` benutzt. Da die Elemente der dünn besetzten Vektoren sortiert in den STL-Vektoren `p` und `x` gespeichert werden, können diese Hilfsklassen einfach Referenzen auf die Originaldaten speichern sowie den Anfangs- und den Endpunkt innerhalb der Originaldaten, welche vom betreffenden Ausschnitt abgedeckt werden. Für diese Hilfsklassen sind daher speziell angepasste Operatoren nötig, welche zwar auf den Originaldaten arbeiten, aber diesen Anfangs- und Endpunkt bei der Berechnung berücksichtigen.

Um einen solchen Ausschnitt zu erzeugen, wird der `operator()` verwendet. Dieser bekommt zwei Ganzzahl-Parameter übergeben, den Start- und den Endindex des Bereichs, der ausgeschnitten werden soll (die Indizes beziehen sich dabei auf die für diesen Vektor aktuell verwendete Indizierung). Der Operator legt dann ein Objekt der betreffenden Hilfsklassen an und liefert dieses Objekt als Rückgabewert zurück. Geschieht dies innerhalb einer Zuweisung, so wird also der Zuweisungsoperator der Hilfsklasse benutzt, welcher dann auch die Originaldaten verändert.

Bei den Datentypen für dünn besetzte Matrizen ist das Vorgehen komplizierter. Zunächst wird das Vorgehen beim Ausschneiden von Teilmatrizen betrachtet. Hierfür stehen ebenfalls spezielle Hilfsklassen namens `srmatrix_slice`, `simatrix_slice` sowie `scmatrix_slice` und `scimatrix_slice` zur Verfügung. Eine Teilmatrix kann mit dem `operator()` ausgeschnitten werden, welcher dann ein Objekt der jeweiligen Hilfsklasse zurück liefert. Listing 3.28 zeigt ein Beispiel für die Anwendung des Operators. Die dem Operator übergebenen Indizes sind wiederum von der aktuell verwendeten Indizierung der jeweiligen Matrix abhängig.

```
1 srmatrix A(10,10);
2
3 //... Matrix fuellen ...
4
5 //gib Teilmatrix zwischen Zeile 2 und 4 und Spalte 3 und 5 aus
6 cout << A(2,4,3,5) << endl;
7 //Belege obige Teilmatrix mit 1.0
8 A(2,4,3,5) = 1.0;
```

Listing 3.28: Beispiel zum Ausschneiden von Teilmatrizen

Im Folgenden wird der Aufbau der Hilfsklassen für Teilmatrizen erläutert. Die Hilfsklassen müssen die Originaldaten in irgendeiner Form referenzieren, um den Schreibzugriff auf die jeweilige Teilmatrix zu ermöglichen. Weiterhin ist das Finden der Elemente der Originalmatrix, welche zur gewünschten Teilmatrix gehören, vergleichsweise aufwändig, da die gesamte Datenstruktur durchlaufen werden muss (für jede Spalte der Teilmatrix muss festgestellt werden, welche Elemente in den angeforderten Zeilen liegen). Da die meisten Zugriffe auf Teilmatrizen lesend sind, wird daher zunächst eine Kopie der Elemente der Teilmatrix in Form einer dünn besetzten Matrix angelegt und als Membervariable `A` gespeichert. Diese Kopie wird für alle Operatoren verwendet, welche der Teilmatrix keine neuen Werte zuweisen. Listing 3.29 zeigt als Beispiel die Implementierung des `operator+` zur Addition eines `srmatrix_slice`- und eines `srmatrix`-Objektes.

```
1 inline srmatrix operator+(const srmatrix_slice& M1, const srmatrix& M2) {
2   return spsp_mm_add<srmatrix,srmatrix,srmatrix,real>(M1.A,M2);
3 }
```

Listing 3.29: Additionsoperator für `srmatrix_slice` und `srmatrix`

Um auch die Zuweisung zu ermöglichen, wird zusätzlich ein Zeiger `M` auf die Originalmatrix gespeichert. Alle zuweisenden Operatoren (also z.B. auch `operator+=`) greifen über diesen Zeiger auf die Originalmatrix zu und verändern sie. Die Zeilen- und Spaltenindizes der Teilmatrix werden implizit in der Kopie `A` der Teilmatrix gespeichert. Als

Beispiel für dieses Vorgehen zeigt Listing 3.30 die Funktion `slsp_mm_assign`, welche einer Teilmatrix eine dünn besetzte Matrix zuweist.

```
1  //TA=Typ der Teilmatrix (z.B. srmatrix_slice)
2  //TB=Typ der zugewiesen Matrix (z.B. srmatrix)
3  //TElement=Typ eines Elements (z.B. real)
4  template<class TA, class TB, class TElement>
5  inline TA& slsp_mm_assign(TA& A, const TB& C) {
6      //Indizes aus Kopie uebernehmen
7      int lb1=A.A.lb1, ub1=A.A.ub1, lb2=A.A.lb2, ub2=A.A.ub2;
8      A.A = C;
9      A.A.lb1=lb1; A.A.ub1=ub1; A.A.lb2=lb2; A.A.ub2=ub2;
10
11     //Anfang und Ende der Teilmatrix in interner Indizierung
12     int start1 = A.A.lb1 - A.M->lb1;
13     int end1 = A.A.ub1 - A.M->lb1;
14     int start2 = A.A.lb2 - A.M->lb2;
15     int end2 = A.A.ub2 - A.M->lb2;
16     int tmp1=0, tmp2=0;
17
18     for(int j=start2 ; j<=end2 ; j++) {
19       //1. Alte Eintraege zwischen start1 und end1 loeschen
20       std::vector<int>::iterator ind_it = A.M->ind.begin()+A.M->p[j];
21       TElement x_it  = A.M->x.begin()+A.M->p[j];
22
23       int size = A.M->p[j+1] - tmp1 + tmp2 - A.M->p[j];
24
25       for(int k=0 ; k<size ; k++) {
26         if(*ind_it>=start1 && *ind_it<=end1) {
27           ind_it = A.M->ind.erase(ind_it);
28           x_it = A.M->x.erase(x_it);
29           tmp1++;
30         } else {
31           x_it++;
32           ind_it++;
33         }
34       }
35
36       A.M->p[j+1] -= tmp1;
37
38       ind_it = A.M->ind.begin()+A.M->p[j];
39       x_it  = A.M->x.begin()+A.M->p[j];
40
41       //2. Neue Eintraege aus C zwischen start1 und end1 kopieren
42       for(int k=A.A.p[j-start2+1]-1 ; k>=A.A.p[j-start2] ; k--) {
43         if(A.A.ind[k]>=0 && A.A.ind[k]<=end1-start1) {
44           ind_it = A.M->ind.insert(ind_it, A.A.ind[k]+start1);
45           x_it = A.M->x.insert(x_it, A.A.x[k]);
46           tmp2++;
47         }
48       }
49
50       A.M->p[j+1] += tmp2;
```

```
51      }
52
53      for(unsigned int i=end2+2 ; i<A.M->p.size() ; i++) {
54        A.M->p[i] += tmp2 - tmp1;
55      }
56
57      return A;
58  }
```

Listing 3.30: Zuweisungsfunktion für Teilmatrizen

Weiterhin besteht bei den dünn besetzten Matrizen nicht nur die Möglichkeit, Teilmatrizen auszuschneiden, sondern es können auch komplette Zeilen- oder Spalten ausgeschnitten werden, welche in diesem Zusammenhang in C-XSC auch als Subvektoren bezeichnet werden. Hierfür werden wieder entsprechende Hilfsklassen benötigt, welche die Bezeichnungen `srmatrix_subv`, `simatrix_subv`, `scmatrix_subv` und `scimatrix_subv` tragen. Mit dem `operator[]` kann dann eine Zeile oder Spalte aus der Matrix ausgeschnitten werden, der Operator liefert als Rückgabewert ein Objekt der jeweiligen Hilfsklasse. Listing 3.31 verdeutlicht die Anwendung an einem Beispiel.

```
1  srmatrix A(10,10);
2
3  //... Matrix fuellen ...
4
5  cout << A[5] << endl;   //gib fuenfte Zeile aus
6  A[1] = 1.0;             //setze erste Zeile auf 1.0
7  A[Col(2)] = 2.0;        //setze zweite Spalte auf 2.0
```

Listing 3.31: Beispiel zum Ausschneiden von Subvektoren

Die Hilfsklassen für Subvektoren ermöglichen ebenfalls das Überschreiben der Originaldaten, d.h. auch hier muss auf die Originaldaten zugegriffen werden können. Da die Subvektoren als Spezialfall einer Teilmatrix angesehen werden können, werden sie intern auch in einer Membervariablen vom Typ `srmatrix_slice`, `simatrix_slice`, bzw. `scmatrix_slice` oder `scimatrix_slice` gespeichert. Zusätzlich wird nur noch festgehalten, ob es sich um eine Zeile oder Spalte handelt und welchen Index diese Zeile bzw. Spalte besitzt. Alle weiteren Berechnungen können so auf die bereits implementierten Funktionen der Teilmatrix-Hilfsklassen zurückgreifen.

Freie Indexwahl

Wie bereits mehrfach angedeutet, gibt es bei den dünn besetzten Datentypen, ebenso wie bei den dicht besetzten, die Möglichkeit, den Indexbereich frei zu wählen. Standardmäßig wird dabei von einer 1-basierten Indizierung ausgegangen, prinzipiell ist aber jede beliebige Indizierung (auch im negativen Bereich) möglich. Diese Möglichkeit kann z.B. die Implementierung bestimmter Algorithmen erleichtern und zu besser lesbarem Code führen.

Um die freie Indexwahl zu ermöglichen, besitzt jeder Datentyp Membervariablen vom Typ `int`, welche den Start- und Endindex des Vektors bzw. der Zeilen und Spalten bei

Matrizen angeben. Bei Benutzung dieser Datentypen werden diese Werte dann gegebenenfalls beim Zugriff auf die (intern immer Null-basiert gespeicherten Elemente) berücksichtigt, was zwar (sehr geringe) Laufzeitverluste nach sich zieht, welche aber durch den zusätzlich gewonnen Komfort für die Nutzung der Datentypen mehr als aufgewogen werden. Die Indizes können mit den Funktionen `Lb` und `Ub` ausgelesen und mit den Funktionen `SetLb` und `SetUb` gesetzt werden. Listing 3.32 zeigt ein kurzes Beispielprogramm zum Umgang mit der freien Indexwahl.

```
1  int n = 100; //Dimension
2
3  srmatrix A(n,n);
4  rvector x(n);
5
6  //Setze untere Indexschranke von x auf 0-basiert
7  //Obere Schranke wird automatisch auf n-1 gesetzt
8  SetLb(x,0);
9
10 //Indexschranke mit Lb und Ub auslesen
11 cout << "Index bounds of x: " << Lb(x) << "," << Ub(x) << endl;
12
13 //Schranke fuer oberen Zeilenindex von A auf n-1 setzen
14 //Untere Schranke wird automatisch auf 0 gesetzt
15 SetUb(A,ROW,n-1);
16
17 //Operatoren arbeiten auch mit unterschiedlichen Indexbereiechen korrekt
18 cout << A*x << endl;
19
20 //Bei Zugriffen in Schleifen muessen aktuelle Indexgrenzen beruecksichtigt
21 //werden
22 for(int i=Lb(x) ; i<=Ub(x) ; i++)
23   x[i]++;
```

Listing 3.32: Beispiel zur freien Indexwahl

Skalarprodukte mit wählbarer Genauigkeit

Auch die dünn besetzten Datentypen unterstützen die in Abschnitt 3.1 erläuterten Methoden zum Berechnen von Skalarprodukten und Skalarproduktausdrücken in K-facher *double*-Präzision. Zum einen wird die Präzision für Operatoren, welche implizit oder explizit Skalarprodukte berechnen, ebenfalls über die globale Variable `opdotprec` gesteuert. Zum anderen können Skalarproduktausdrücke mit Hilfe der `dotprecision`-Variablen auf die gleiche Weise wie bei den dicht besetzten Typen berechnet werden. Auch hier wird im Fall $K = 0$ der Akkumulator (maximale Genauigkeit), im Fall $K = 1$ reine Gleitkommarechnung (am schnellsten) und im Fall $K \geq 2$ der DotK-Algorithmus verwendet.

Allerdings muss die Berechnung für die dünn besetzten Datentypen intern anders aufgebaut werden. Bei den dicht besetzten Typen können dem jeweils verwendeten Algorithmus direkt zwei komplette Vektoren übergeben werden, für welche das Skalarprodukt berechnet werden soll. Nun aber müssen die am Skalarprodukt beteiligten (dünn besetzten) Vektoren durchlaufen und nach den Elementen durchsucht werden, welche

einen Beitrag zum Skalarprodukt liefern (welche also in beiden Vektoren nicht den Wert Null haben). Die Berechnung des Skalarproduktes muss entsprechend so implementiert werden, dass die einzelnen elementweisen Produkte des Gesamt-Skalarproduktes einzeln an die Berechnungsfunktion übergeben werden können. Zu diesem Zweck wird für jeden Grunddatentyp eine Hilfsklasse eingeführt. Zur Erklärung wird hier die Klasse `sparse_dot` für reelle Skalarprodukte betrachtet. Der Aufbau der Klasse ist in Listing 3.33 gegeben. Der Aufbau der anderen Klassen ist analog.

```
1  class sparse_dot {
2
3    private:
4
5      dotprecision* dot;
6      std::vector<real> cm;
7      std::vector<real> ca;
8      real val;
9      real corr;
10     real err;
11     int n;
12     int k;
13
14
15   public:
16
17     sparse_dot(unsigned int p);
18     sparse_dot(const sparse_dot& s);
19     ~sparse_dot();
20     void reset();
21     void add_dot(const real& x, const real& y);
22     void add_dot_err(const real& x, const real& y);
23     real result();
24     void result(dotprecision& res_dot);
25
26 };
```

Listing 3.33: Aufbau der Klasse `sparse_dot`

Die Klasse besitzt folgende Datenmember:

- `dotprecision* dot`: Zeigt auf eine `dotprecision` Variable, falls mit Präzision $K = 0$ gerechnet wird. Ist $K \neq 0$, so wird, um Speicher zu sparen, kein `dotprecision`-Objekt angelegt und der Zeiger zeigt auf `NULL`.

- `std::vector<real> cm,ca`: In diesen Vektoren werden die Ergebnisse der fehlerfreien Transformationen für den DotK Algorithmus zwischengespeichert.

- `real val`: Aktuelles Zwischenergebnis bei Benutzung von Gleitkommarechnung oder Zwischenergebnis ohne Korrekturwerte bei Verwendung des DotK-Algorithmus.

- `real corr`: Summe der Korrekturwerte bei Verwendung des DotK-Algorithmus.

- `real err`: Hilfsvariable zur Fehlerschrankenberechnung bei Verwendung des DotK-Algorithmus mit Fehlerschranke.
- `int n`: Tatsächliche Dimension des Skalarprodukts (Anzahl der einzelnen Produkte).
- `int k`: Verwendete Präzision K.

Neben den Konstruktoren und dem Destruktor werden die folgenden Memberfunktionen für die Berechnung bereitgestellt:

- `void reset()`: Setzt das `sparse_dot`-Objekt in den Ausgangszustand zurück, so dass ein neues Skalarprodukt berechnet werden kann.
- `void add_dot(const real& x, const real& y)`: Fügt das Produkt `x*y` zu diesem Skalarprodukt hinzu. Je nach gewählter Genauigkeit wird ein Teil des jeweiligen Algorithmus berechnet (s.u.) und in den entsprechenden Membervariablen zwischengespeichert. Auf die Berechnung von Fehlerschranken wird dabei verzichtet.
- `void add_dot_err(const real& x, const real& y)`: Analog zur obigen Funktion `add_dot`, aber mit Berechnung von Fehlerschranken.
- `real result()`: Berechnet das Endergebnis und gibt einen entsprechenden Rückgabewert zurück.
- `void result(dotprecision& res_dot)`: Berechnet das Endergebnis und speichert es in der übergebenen `dotprecision`-Variable.

Zur Verdeutlichung zeigt Listing 3.34 den Quelltext der Funktionen `add_dot` und `result`, welche keine Fehlerberechnungen vornehmen. Anhand dieser Listings ist zu erkennen, dass es sich im Endeffekt um eine Aufteilung der Berechnungsschritte der in Abschnitt 3.1 beschriebenen Funktion `addDot_op` handelt. Für andere Datentypen und bei den Versionen mit Berechnung einer Fehlerschranke wird in ähnlicher Weise analog zu den in Abschnitt 3.1 beschriebenen Funktionen für den jeweiligen Fall vorgegangen.

```
1  void sparse_dot::add_dot(const real& x, const real& y) {
2    if(k==0) {
3      accumulate(*dot, x, y);
4    } else if(k==1) {
5      val += x*y;
6    } else if(k==2) {
7      real a,b,c;
8      TwoProduct(x,y,a,b);
9      TwoSum(val,a,val,c);
10     corr += (b+c);
11   } else if(k>=3) {
12     real a,b;
13     TwoProduct(x,y,a,b);
14     cm.push_back(b);
```

```
15      TwoSum(val,a,val,b);
16      ca.push_back(b);
17    }
18  }
19
20  real sparse_dot::result() {
21    if(k==0) {
22      return rnd(*dot);
23    } else if(k==1) {
24      return val;
25    } else if(k==2) {
26      return val+corr;
27    } else if(k>=3) {
28      n = cm.size();
29      if(n == 0) return val;
30
31      for(int j=1 ; j<k-1 ; j++) {
32        for(int i=1 ; i<n ; i++)
33          TwoSum(cm[i],cm[i-1],cm[i],cm[i-1]);
34          TwoSum(ca[0],cm[n-1],ca[0],cm[n-1]);
35          for(int i=1 ; i<n ; i++)
36            TwoSum(ca[i],ca[i-1],ca[i],ca[i-1]);
37          TwoSum(val,ca[n-1],val,ca[n-1]);
38       }
39
40      corr = std::accumulate(cm.begin(),cm.end(),corr);
41      corr = std::accumulate(ca.begin(),ca.end(),corr);
42
43      val += corr;
44
45      return val;
46    }
47
48    return val;
49  }
```

Listing 3.34: Die Funktionen `add_dot` und `result`

Mit Hilfe der Klasse `sparse_dot` (bzw. der entsprechenden Klassen für die anderen Datentypen) können nun die Template-Berechnungsfunktionen der dünn besetzten Datentypen so formuliert werden, dass sie für jede gewählte Skalarprodukt-Genauigkeit die richtigen Berechnungen ausführen. Das Skalarprodukt zweier Vektoren `v1` und `v2` innerhalb eines Operators kann nun z.B. wie in Listing 3.35 implementiert werden.

```
1  TDot dot(opdotprec);
2
3  for(int i=0 ; i<v1.get_nnz() ; i++) {
4     for(int j=0 ; j<v2.get_nnz() && v2.p[j]<=v1.p[i] ; j++) {
5        if(v1.p[i] == v2.p[j])
6          dot.add_dot(v1.x[i], v2.x[j]);
7     }
8  }
9
```

```
10 return dot.result();
```

Listing 3.35: Beispielimplementierung für die Berechnung eines Skalarproduktes

Hierbei entspricht der Template-Typ `TDot` der jeweils benötigten Variante der Klasse `sparse_dot`. Dem entsprechenden Objekt wird die aktuelle Präzision für Operatoren aus der globalen Variablen `opdotprec` übergeben. Die einzelnen Produkte des folgenden Skalarprodukts können nun über die Funktion `add_dot` an dieses Objekt übergeben werden. Um schließlich das Ergebnis zu erhalten, muss nur noch die `result()`-Methode aufgerufen werden.

Weitere Funktionen

Im Folgenden werden einige weitere Funktionen der dünn besetzten Datentypen kurz beschrieben:

- `int get_nnz()` (Memberfunktion): Liefert die Anzahl der Nicht-Null-Einträge der betreffenden Matrix bzw. des betreffenden Vektors.
- `void dropzeros()` (Memberfunktion): Löscht alle explizit gespeicherten Nullen (eingeführt z.B. durch Auslöschung) aus der Datenstruktur.
- `srmatrix transp(const srmatrix&)` (einfache Funktion, analog für andere dünn besetzte Matrixtypen): Transponiert eine dünn besetzte Matrix.
- Standardfunktionen wie `Re` (Realteil), `Im` (Imaginärteil), `Sup` (Supremum), `Inf` (Infimum), analog zu den dicht besetzten Datentypen.

Zusätzlich bieten die dünn besetzten Datentypen noch einige Memberfunktionen, welche direkten Zugriff auf die zugrunde liegende Datenstruktur erlauben. Diese sind:

- `vector<int>& row_indices()` (Memberfunktion aller Vektordatentypen): Liefert eine Referenz auf den `vector<int> p` zurück.
- `vector<T>& values()` (Memberfunktion aller Vektordatentypen): Liefert eine Referenz auf den `vector<T> x` zurück (`T` ist der jeweilige Grunddatentyp).
- `vector<int>& row_indices()` (Memberfunktion aller Matrixdatentypen): Liefert eine Referenz auf den `vector<int> ind` zurück.
- `vector<int>& column_pointers()` (Memberfunktion aller Matrixdatentypen): Liefert eine Referenz auf den `vector<int> p` zurück.
- `vector<T>& values()` (Memberfunktion aller Matrixdatentypen): Liefert eine Referenz auf den `vector<T> x` zurück (`T` ist der jeweilige Grunddatentyp).

Dieser Zugriff auf die interne Datenstruktur widerspricht zwar grundsätzlich dem objektorientierten Paradgima der Datenkapselung, ist in diesem Fall aber als Option nötig, um dem Benutzer die Implementierung performanter Algorithmen (z.B. für die LU-Zerlegung

einer dünn besetzten Matrix) bzw. von Schnittstellen zu bestehender Software für dünn besetzte Matrizen zu ermöglichen. Der Anwender muss dafür Sorge tragen, dass die Datenstruktur konsistent bleibt, d.h. weiterhin dem in Abschnitt 3.3.1 beschriebenen Format entspricht.

Hinweise zum praktischen Umgang

Zum Abschluss dieses Abschnitts sollen noch kurz einige Hinweise zur richtigen Benutzung der dünn besetzten Datentypen und zu einigen Unterschieden zu dicht besetzten Datentypen gegeben werden. Um performante Programme zu erhalten, sollten diese Hinweise unbedingt beachtet werden.

- Der Zugriff auf einzelne Elemente der Matrix bzw. des Vektors über die entsprechenden Operatoren sollte nach Möglichkeit vermieden werden, insbesondere der wiederholte Zugriff innerhalb von Schleifen. Die dahinter stehenden Operationen sind deutlich aufwändiger als bei den dicht besetzten Typen und können das Laufzeitverhalten sehr negativ beeinflussen.

- Falls doch auf einzelne Elemente zugegriffen werden muss, sollte zwischen lesendem und schreibendem Zugriff unterschieden und der entsprechende Operator verwendet werden.

- Das Ausschneiden von Teilmatrizen und Subvektoren sollte ebenfalls aus Performancegründen möglichst selten eingesetzt werden.

- Falls der Zugriff auf Subvektoren einer dünn besetzten Matrix doch nötig sein sollte (z.B. bei Berechnung von Skalarproduktausdrücken mittels einer `dotprecision` Variable), so sollte möglichst auf Spalten statt auf Zeilen zugegriffen werden. Ist der Zugriff auf Zeilen nötig, so kann es evtl. vorteilhaft sein, die Matrix zunächst zu transponieren und dann auf die Spalten der transponierten Matrix zuzugreifen.

- Bei der Implementierung spezieller Algorithmen für dünn besetzte Datenstrukturen ist es zumeist von Vorteil, direkt auf die zugrunde liegende Datenstruktur (unter Umgehung der Datenkapselung) zuzugreifen (bzw. mit deren Hilfe Schnittstellen zu vorhandenen Softwarepaketen für dünn besetzte Matrizen zu erstellen).

3.3.3. Tests und Zeitmessungen

In diesem Abschnitt sollen nun einige Tests und Zeitmessungen mit den dünn besetzten Datentypen durchgeführt werden. Insbesondere werden Laufzeit und Ergebnisse der Grundoperationen mit den dicht besetzten Datentypen verglichen und es werden Vergleichsmessungen mit einigen anderen Softwarepaketen vorgenommen. Weiterhin werden die Auswirkungen der Benutzung von Teilmatrizen auf die Laufzeit näher betrachtet. Die Tests in diesem Abschnitt beschränken sich auf die dünn besetzten Matrix-Datentypen, da diese eine weitaus größere praktische Relevanz besitzen als dünn besetzte Vektoren.

Name	Dimension	nnz	Typ
af23560	23560×23560	484256	Reell
bcsstk10	1086×1086	22070	Reell
bcsstm05	153×153	153	Reell
conf5.4-00l8x8-0500	49152×49152	1916928	Komplex
dwg961a	961×961	3405	Komplex
mahindas	1258×1258	7682	Reell
mhd1280a	1280×1280	47906	Komplex
qc324	324×324	26730	Komplex
tub1000	1000×1000	3996	Reell
young1c	841×841	7337	Komplex

Tabelle 3.3.: Übersicht der verwendeten Testmatrizen

Die Aussagen der Tests in diesem Abschnitt lassen sich aber weitgehend auf die dünn besetzten Vektordatentypen übertragen.

Für die folgenden Tests werden einige Matrizen aus dem Matrix-Market verwendet. Diese Matrizen entstammen zumeist praktischen Anwendungen. In Tabelle 3.3 werden sie mit ihren Eigenschaften kurz aufgelistet. Für weitere Details zu den Testmatrizen sei auf Anhang B verwiesen. Für die Testmessungen mit (komplexen) Intervallmatrizen werden diese Testmatrizen um den Faktor 10^{-10} aufgebläht.

Das Testsystem für alle Messungen in diesem Abschnitt besteht wie zuvor aus zwei Intel Xeon E5520 Prozessoren mit 2.26 GHz und je 4 Kernen sowie 24 GB DDR3 Hauptspeicher. Bei den hier durchgeführten Tests wird allerdings (sofern nichts anderes angegeben ist) nur ein Prozessorkern verwendet, da die dünn besetzten Grundoperationen in C-XSC nicht parallelisiert sind. Alle Operationen wurden zehn Mal ausgeführt, der Durchschnitt aller Messungen wird als Testwert verwendet.

Zunächst werden die Zeiten für die Grundoperationen Addition (`operator+`), konvexe Hülle (`operator|`) und Schnitt (`operator&`, nur für Intervallmatrizen) gemessen und mit den dicht besetzten Typen verglichen. Tabelle 3.4 zeigt die Ergebnisse bei Verwendung von Punktmatrizen, Tabelle 3.5 die Ergebnisse bei Verwendung von Intervallmatrizen. Die Matrizen `af23560` und `conf5.4-00l8x8-0500` konnten aufgrund ihrer Größe nicht mit den dicht besetzten Datentypen benutzt werden, da nicht genug Speicher zur Verfügung stand.

Die Ergebnisse zeigen, dass die Grundoperationen bei Benutzung der dünn besetzten Typen in etwa um zwei Größenordnungen schneller sind, da sie nur einen Aufwand $\mathcal{O}(nnz)$ statt wie im dicht besetzten Fall $\mathcal{O}(n^2)$ besitzen. Die Berechnung der konvexen Hülle ist für Punktmatrizen etwas langsamer als die Addition und auch langsamer als für Intervalle, da C-XSC hier intern die Elemente der Operanden vom Typ `real` bzw.

Matrix	Besetztheit	operator+	operator|
af23560	Dünn	$5.6 \cdot 10^{-3}$	$1.0 \cdot 10^{-2}$
	Dicht	-	-
bcsstk10	Dünn	$2.4 \cdot 10^{-4}$	$3.6 \cdot 10^{-4}$
	Dicht	$6.7 \cdot 10^{-3}$	$1.7 \cdot 10^{-2}$
bcsstm05	Dünn	$3.6 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$
	Dicht	$5.3 \cdot 10^{-5}$	$6.2 \cdot 10^{-4}$
conf5.4-00l8x8-0500	Dünn	$4.9 \cdot 10^{-2}$	$1.1 \cdot 10^{-1}$
	Dicht	-	-
dwg961a	Dünn	$5.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$
	Dicht	$1.1 \cdot 10^{-2}$	$4.6 \cdot 10^{-2}$
mahindas	Dünn	$8.8 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$
	Dicht	$8.7 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$
mhd1280a	Dünn	$7.2 \cdot 10^{-4}$	$1.9 \cdot 10^{-3}$
	Dicht	$1.4 \cdot 10^{-2}$	$8.8 \cdot 10^{-2}$
qc324	Dünn	$3.9 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$
	Dicht	$5.1 \cdot 10^{-4}$	$4.8 \cdot 10^{-3}$
tub1000	Dünn	$4.5 \cdot 10^{-5}$	$8.0 \cdot 10^{-5}$
	Dicht	$4.2 \cdot 10^{-3}$	$1.2 \cdot 10^{-2}$
young1c	Dünn	$1.3 \cdot 10^{-4}$	$3.0 \cdot 10^{-4}$
	Dicht	$7.2 \cdot 10^{-3}$	$3.1 \cdot 10^{-2}$

Tabelle 3.4.: Zeitmessung für Grundoperationen mit Punktmatrizen

Matrix	**Besetztheit**	**operator+**	**operator|**	**operator&**
af23560	Dünn	$1.3 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$
	Dicht	-	-	-
bcsstk10	Dünn	$6.6 \cdot 10^{-4}$	$4.3 \cdot 10^{-4}$	$4.5 \cdot 10^{-4}$
	Dicht	$2.6 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$
bcsstm05	Dünn	$4.7 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$	$6.5 \cdot 10^{-6}$
	Dicht	$4.4 \cdot 10^{-4}$	$1.9 \cdot 10^{-4}$	$7.0 \cdot 10^{-4}$
conf5.4-00l8x8-0500	Dünn	$1.1 \cdot 10^{-1}$	$8.1 \cdot 10^{-2}$	$8.5 \cdot 10^{-2}$
	Dicht	-	-	-
dwg961a	Dünn	$2.1 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$8.6 \cdot 10^{-4}$
	Dicht	$5.1 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$	$1.9 \cdot 10^{-2}$
mahindas	Dünn	$1.8 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$
	Dicht	$3.4 \cdot 10^{-2}$	$2.0 \cdot 10^{-2}$	$1.8 \cdot 10^{-2}$
mhd1280a	Dünn	$1.9 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$
	Dicht	$1.1 \cdot 10^{-1}$	$6.3 \cdot 10^{-2}$	$6.1 \cdot 10^{-2}$
qc324	Dünn	$1.0 \cdot 10^{-3}$	$6.2 \cdot 10^{-4}$	$6.7 \cdot 10^{-4}$
	Dicht	$4.5 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	$2.1 \cdot 10^{-3}$
tub1000	Dünn	$9.2 \cdot 10^{-5}$	$8.5 \cdot 10^{-5}$	$8.1 \cdot 10^{-5}$
	Dicht	$2.1 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	$1.0 \cdot 10^{-2}$
young1c	Dünn	$3.2 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$	$1.7 \cdot 10^{-4}$
	Dicht	$3.4 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$

Tabelle 3.5.: Zeitmessung für Grundoperationen mit Intervallmatrizen

`complex` erst in den Typ `interval` bzw. `cinterval` umwandeln muss. Bei Verwendung von Intervallen sind in der Regel alle elementweise vorgehenden Operatoren in etwa auf dem gleichen Geschwindigkeitsniveau. Die numerischen Ergebnisse waren bei allen Tests identisch mit denen der dicht besetzten Datentypen und werden hier daher nicht extra aufgeführt.

Als nächstes wird die Berechnung von Matrix-Matrix-Produkten mit dem `operator*` betrachtet. Dazu wird jeweils das Produkt jeder Testmatrix mit sich selbst berechnet. Die Präzision für die Berechnung der dabei auftretenden Skalarprodukte wird jeweils einmal auf $K = 0$, $K = 1$, $K = 2$ und $K = 3$ gesetzt. Außerdem wird bei Verwendung der dicht besetzten Matrizen und $K = 1$ zusätzlich die Laufzeit bei Nutzung der BLAS-Unterstützung von C-XSC aus Abschnitt 3.2 ermittelt. Als BLAS Bibliothek wird dabei die Intel-MKL Version 10.2 verwendet. Die Tabellen 3.6 und 3.7 zeigen die Ergebnisse bei Verwendung von Punkt- bzw. Intervallmatrizen.

Der Unterschied zwischen den dicht und den dünn besetzten Datentypen ist hier deutlich größer, da im dicht besetzten Fall nun eine Operation mit Aufwand $\mathcal{O}(n^3)$ betrachtet wird. Teilweise tritt hier das Phänomen auf, dass bei Verwendung dicht besetzter Matrizen der Fall $K = 0$, also maximale Genauigkeit, sogar schneller ist, als der Fall $K = 2$. Dieser Effekt kommt dadurch zustande, dass der bei $K = 0$ verwendete Akkumulator einzelne Produkte, bei denen einer der beiden Faktoren gleich Null ist, nicht berechnet bzw. diesen Fall direkt zu Beginn abfragt und keine weiteren Berechnungen mehr vornimmt, was beim DotK Algorithmus nicht der Fall ist. Ist die als dicht besetzt gespeicherte Matrix also eigentlich dünn besetzt, wie in diesem Test, so wird ein Großteil der einzelnen Produkte im Fall $K = 0$ gar nicht berechnet, was hier je nach Matrix zu einer schnelleren Verarbeitung als bei Verwendung des DotK-Algorithmus führen kann.

Ein weiteres interessantes Ergebnis ist, dass auch bei Verwendung der hochoptimierten BLAS-Routinen mit 8 Prozessorkernen die Berechnung auf einem Kern mit den dünn besetzten Datentypen immer noch - durchweg für alle Testmatrizen - schneller ist. Dies verdeutlicht noch einmal den Nutzen spezieller Algorithmen und Datenstrukturen für dünn besetzte Matrizen. Die numerischen Ergebnisse obiger Tests waren wiederum identisch. Theoretisch kann es bei Verwendung des DotK-Algorithmus passieren, dass die berechneten Intervalleinschließungen bei Nutzung dünn besetzter Datentypen etwas besser sind. Dieser Effekt kommt dadurch zustande, dass bei der Berechnung der Fehlerschranke nicht die Dimension n des Skalarproduktes benutzt wird, sondern die tatsächliche Zahl der berechneten einzelnen Produkte (also nur die Anzahl der Produkte, welche nicht Null ergeben).

Als nächstes wird die Geschwindigkeit des Matrix-Matrix-Produktes der C-XSC Datentypen mit einigen anderen Softwarepaketen verglichen. Dazu werden zum einen Matlab bzw. Intlab, zum anderen die C-Bibliothek CSparse (bzw. CXSparse zur Unterstützung von komplexen Matrizen) von Davis [35] betrachtet. Tabelle 3.8 zeigt die Ergebnisse der Vergleichsmessung. CXSparse unterstützt keine Intervallmatrizen, weshalb die entsprechende Vergleichsmessung fehlt. Außerdem ist zu beachten, dass weder Matlab/Intlab noch CXSparse die in C-XSC vorhandene Berechnung von Skalarprodukten mit höherer Genauigkeit direkt unterstützt.

Die Ergebnisse zeigen, dass C-XSC und Matlab/Intlab bei reellen Punkt- und Intervall-

Matrix	**Besetztheit**	$K = 0$	$K = 1$	$K = 2$	$K = 3$
af23560	Dünn	1.4	0.1	0.3	0.5
	Dicht	-	-	-	-
	Dicht, BLAS	-	-	-	-
bcsstk10	Dünn	$5.8 \cdot 10^{-2}$	$4.0 \cdot 10^{-3}$	$9.1 \cdot 10^{-3}$	$1.7 \cdot 10^{-2}$
	Dicht	9.0	2.4	27.2	44.0
	Dicht, BLAS	-	$8.1 \cdot 10^{-2}$	-	-
bcsstm05	Dünn	$7.5 \cdot 10^{-5}$	$7.2 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$3.6 \cdot 10^{-5}$
	Dicht	$2.7 \cdot 10^{-2}$	$5.9 \cdot 10^{-3}$	$3.3 \cdot 10^{-2}$	$8.2 \cdot 10^{-2}$
	Dicht, BLAS	-	$8.3 \cdot 10^{-4}$	-	-
conf5.4-00l8x8-0500	Dünn	33.0	1.1	3.8	7.9
	Dicht	-	-	-	-
	Dicht, BLAS	-	-	-	-
dwg961a	Dünn	$8.1 \cdot 10^{-3}$	$2.9 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$	$2.4 \cdot 10^{-3}$
	Dicht	112.8	3.5	61.5	89.0
	Dicht, BLAS	-	$1.2 \cdot 10^{-1}$	-	-
mahindas	Dünn	$1.0 \cdot 10^{-2}$	$1.6 \cdot 10^{-3}$	$2.2 \cdot 10^{-3}$	$3.4 \cdot 10^{-3}$
	Dicht	13.9	3.7	47.7	73.6
	Dicht, BLAS	-	$6.5 \cdot 10^{-2}$	-	-
mhd1280a	Dünn	$6.6 \cdot 10^{-1}$	$2.1 \cdot 10^{-2}$	$9.7 \cdot 10^{-2}$	$2.0 \cdot 10^{-1}$
	Dicht	298.4	8.3	163.3	227.0
	Dicht, BLAS	-	$2.6 \cdot 10^{-1}$	-	-
qc324	Dünn	$8.0 \cdot 10^{-1}$	$1.9 \cdot 10^{-2}$	$9.6 \cdot 10^{-2}$	$2.0 \cdot 10^{-1}$
	Dicht	4.1	$1.1 \cdot 10^{-1}$	2.1	3.2
	Dicht, BLAS	-	$6.9 \cdot 10^{-3}$	-	-
tub1000	Dünn	$2.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-4}$	$6.6 \cdot 10^{-4}$	$1.2 \cdot 10^{-3}$
	Dicht	7.0	1.8	18.1	30.8
	Dicht, BLAS	-	$3.5 \cdot 10^{-2}$	-	-
young1c	Dünn	$1.4 \cdot 10^{-2}$	$6.9 \cdot 10^{-4}$	$3.1 \cdot 10^{-3}$	$7.4 \cdot 10^{-3}$
	Dicht	74.1	2.3	40.8	59.8
	Dicht, BLAS	-	$8.1 \cdot 10^{-2}$	-	-

Tabelle 3.6.: Zeitmessung für Matrix-Matrix-Produkte mit Punktmatrizen

Matrix	**Besetztheit**	$K = 0$	$K = 1$	$K = 2$	$K = 3$
af23560	Dünn	2.9	0.7	0.9	1.4
	Dicht	-	-	-	-
	Dicht, BLAS	-	-	-	-
bcsstk10	Dünn	$1.2 \cdot 10^{-1}$	$2.6 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$	$4.9 \cdot 10^{-2}$
	Dicht	116.1	49.1	57.4	74.9
	Dicht, BLAS	-	$2.1 \cdot 10^{-1}$	-	-
bcsstm05	Dünn	$1.2 \cdot 10^{-4}$	$6.4 \cdot 10^{-5}$	$4.0 \cdot 10^{-5}$	$9.2 \cdot 10^{-5}$
	Dicht	$2.4 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	$1.8 \cdot 10^{-1}$
	Dicht, BLAS	-	$1.9 \cdot 10^{-3}$	-	-
conf5.4-00l8x8-0500	Dünn	78.9	19.6	15.8	29.5
	Dicht	-	-	-	-
	Dicht, BLAS	-	-	-	-
dwg961a	Dünn	$1.7 \cdot 10^{-2}$	$6.1 \cdot 10^{-3}$	$5.2 \cdot 10^{-3}$	$9.4 \cdot 10^{-3}$
	Dicht	249.7	43.8	144.5	194.9
	Dicht, BLAS	-	$7.1 \cdot 10^{-1}$	-	-
mahindas	Dünn	$2.0 \cdot 10^{-2}$	$5.2 \cdot 10^{-3}$	$8.1 \cdot 10^{-3}$	$1.2 \cdot 10^{-2}$
	Dicht	177.1	77.0	94.4	121.9
	Dicht, BLAS	-	$3.3 \cdot 10^{-1}$	-	-
mhd1280a	Dünn	1.6	$4.3 \cdot 10^{-1}$	$3.7 \cdot 10^{-1}$	$6.7 \cdot 10^{-1}$
	Dicht	743.5	131.5	523.4	635.5
	Dicht, BLAS	-	1.6	-	-
qc324	Dünn	2.0	$4.1 \cdot 10^{-1}$	$3.3 \cdot 10^{-1}$	$6.4 \cdot 10^{-1}$
	Dicht	10.2	1.4	5.2	6.9
	Dicht, BLAS	-	$4.5 \cdot 10^{-2}$	-	-
tub1000	Dünn	$5.8 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	$2.0 \cdot 10^{-3}$	$3.7 \cdot 10^{-3}$
	Dicht	89.1	37.3	44.6	56.7
	Dicht, BLAS	-	$1.8 \cdot 10^{-1}$	-	-
young1c	Dünn	$3.4 \cdot 10^{-2}$	$1.5 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	$2.4 \cdot 10^{-2}$
	Dicht	152.9	28.6	94.9	129.0
	Dicht, BLAS	-	$8.6 \cdot 10^{-1}$	-	-

Tabelle 3.7.: Zeitmessung für Matrix-Matrix-Produkte mit Intervallmatrizen

Matrix	Typ	C-XSC	Matlab/Intlab	CXSparse
af23560	Punkt	0.1	0.1	$6.5 \cdot 10^{-2}$
	Intervall	0.7	0.7	-
bcsstk10	Punkt	$4.0 \cdot 10^{-3}$	$3.2 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$
	Intervall	$2.6 \cdot 10^{-2}$	$1.5 \cdot 10^{-2}$	-
bcsstm05	Punkt	$7.2 \cdot 10^{-6}$	$7.7 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$
	Intervall	$6.4 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	-
conf5.4-00l8x8-0500	Punkt	1.1	1.1	0.7
	Intervall	19.6	11.1	-
dwg961a	Punkt	$2.9 \cdot 10^{-4}$	$2.5 \cdot 10^{-4}$	$2.2 \cdot 10^{-4}$
	Intervall	$6.1 \cdot 10^{-3}$	$3.1 \cdot 10^{-3}$	-
mahindas	Punkt	$1.6 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$3.7 \cdot 10^{-4}$
	Intervall	$5.2 \cdot 10^{-3}$	$5.8 \cdot 10^{-3}$	-
mhd1280a	Punkt	$2.1 \cdot 10^{-2}$	$1.8 \cdot 10^{-2}$	$1.8 \cdot 10^{-2}$
	Intervall	$4.3 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	-
qc324	Punkt	$1.9 \cdot 10^{-2}$	$1.5 \cdot 10^{-2}$	$1.8 \cdot 10^{-2}$
	Intervall	$4.1 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	-
tub1000	Punkt	$2.3 \cdot 10^{-4}$	$1.7 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$
	Intervall	$1.9 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	-
young1c	Punkt	$6.9 \cdot 10^{-4}$	$2.6 \cdot 10^{-4}$	$6.5 \cdot 10^{-4}$
	Intervall	$1.5 \cdot 10^{-2}$	$2.3 \cdot 10^{-3}$	-

Tabelle 3.8.: Vergleichsmessung für Matrix-Matrix-Produkte mit Intervallmatrizen

Typ	Mit Teilmatrizen	Ohne Teilmatrizen	Verlangsamungsfaktor
`rmatrix`	3.44	3.40	1.01
`srmatrix`	0.28	0.004	70.0

Tabelle 3.9.: Vergleichsmessung für die Nutzung von Teilmatrizen

matrizen auf einem Niveau liegen, ebenso bei komplexen Punktmatrizen. Bei komplexen Intervallen hat Intlab recht deutliche Vorteile, da sich hier erneut die in Intlab verwendete Kreisscheibenarithmetik als Vorteil erweist. Matlab bzw. Intlab muss allerdings den eigentlichen Berechnungscode interpretieren, wodurch ein gewisser Mehraufwand entsteht. Aufgrund des für diesen Test sehr simplen Codes dürfte dieser Mehraufwand aber hier kaum ins Gewicht fallen (die eigentliche Berechnung des Produktes ist auch in Matlab nativer Code). CXSparse ist insbesondere im Reellen dank seiner leichtgewichtigen Natur zumeist am schnellsten. Im Komplexen hingegen bewegt es sich in etwa auf dem gleichen Niveau wie C-XSC und Intlab. Insgesamt sind die dünn besetzten Datentypen in C-XSC von der Performance her also durchaus konkurrenzfähig. Weiterhin bietet C-XSC als einziges Softwarepaket die Möglichkeit, die Genauigkeit von Skalarproduktberechnungen frei zu wählen oder bei Bedarf auch ganze Skalarproduktausdrücke hochgenau zu berechnen.

Zum Abschluss dieses Kapitels soll noch ein Blick auf den Einfluss einer starken Nutzung von Teilmatrizen aus dünn besetzten Matrizen auf die Laufzeit erfolgen. Anders als bei den dicht besetzten Matrizen ist dies durch die zu Grunde liegende Datenstruktur eine, besonders bei schreibendem Zugriff, recht aufwändige Operation, da ein großer Teil der Datenstruktur durchlaufen wird bzw. abgeändert werden muss. Um den Effekt zu testen, wird die in Listing 3.36 angegebene Template-Funktion für die Matrix-Datentypen `rmatrix` und `srmatrix` ausgeprägt und aufgerufen. Die Matrix `A` speichert jeweils die Einheitsmatrix. Die Funktion wird jeweils einmal mit Nutzung von Teilmatrizen und einer Matrix der Dimension 1000×1000 und einmal ohne Nutzung von Teilmatrizen und einer Matrix der Dimension 100×100 aufgerufen. In allen Fällen wird also eine 100×100 Einheitsmatrix 1000 mal mit sich selbst multipliziert. Tabelle 3.9 zeigt die Ergebnisse.

```
1  template<typename T>
2  void f(bool slices, T& A) {
3    for(int i=0; i<=1000; i++) {
4      if(slices) {
5        A(1,100,1,100) *= A(1,100,1,100);
6      } else {
7        A *= A;
8      }
9    }
10 }
```

Listing 3.36: Funktion zum Testen der Auswirkungen der Nutzung von Teilmatrizen auf die Laufzeit

Bei Benutzung dicht besetzter Matrizen wirkt sich der Aufwand für die Teilmatrizen so gut wie gar nicht aus. Bei den dünn besetzten Matrizen hingegen steigt der Aufwand,

allein durch die Verwendung von Teilmatrizen, um den Faktor 70 an. Dies verdeutlicht noch einmal, dass bei Verwendung der dünn besetzten Datentypen Teilmatrizen möglichst selten verwendet werden sollten.

3.4. Schnittstelle zu MPI

Bei der Programmierung für Parallelrechner mit verteiltem Speicher, also in der Regel von in einem Netzwerk zusammengeschlossenen Einzelrechnern, ist es notwendig, Daten zwischen den einzelnen Knoten zu verschicken. Genauere Erläuterungen finden sich in Abschnitt 2.2.2.

Die am weitesten verbreitete Schnittstelle hierfür ist das *Message Passing Interface* (MPI), welches eine Reihe von Kommunikationsfunktionen, insbesondere zum Senden und Empfangen von Daten, bereitstellt. Diese Funktionen sind allerdings nur für Standard C-Datentypen, also z.B. `int`, `double` etc. ausgelegt, für die es entsprechende Gegenstücke im MPI-Standard gibt (`MPI_INT`, `MPI_DOUBLE` etc.). Um C-XSC-Datentypen mittels MPI verschicken zu können, ist daher eine passende Schnittstelle nötig. MPI bietet entsprechende Möglichkeiten, um auch für zusammengesetzte Datentypen wie die C-XSC-Klassen die benötigten Kommunikationsfunktionen implementieren zu können. Diese Schnittstelle wird im Folgenden beschrieben. Sie basiert auf einer vorherigen Implementierung durch Markus Grimmer [49, 50], welche zur Unterstützung der in diesem Kapitel erläuterten neuen C-XSC Möglichkeiten (dünn besetzte Datentypen, angepasste `dotprecision`-Typen für Skalarprodukte mit wählbarer Genauigkeit) erweitert und angepasst wurde.

Allgemein gibt es bei MPI die Möglichkeit, eigene Datentypen zu definieren, die auch aus mehreren (entweder Standard- oder bereits vorher selbst definierten) Datentypen zusammengesetzt sein können. Ein Nachteil hierbei ist, dass die Datentypen eine feste Größe haben müssen, d.h. für die Matrix- und Vektordatentypen, deren Größe sich zur Laufzeit ändern kann, ist diese Methode nicht geeignet. Die vier C-XSC Grunddatentypen können allerdings auf diese Weise festgelegt werden. Listing 3.37 zeigt den entsprechenden Quellcode aus der C-XSC MPI-Schnittstelle.

```
1  bool MPI_CXSC_TYPES_DEFINED=false;
2  MPI_Datatype MPI_CXSC_REAL;
3  MPI_Datatype MPI_CXSC_COMPLEX;
4  MPI_Datatype MPI_CXSC_INTERVAL;
5  MPI_Datatype MPI_CXSC_CINTERVAL;
6
7
8  int MPI_Define_CXSC_Types() {
9    int err;
10   if ((err=MPI_Type_contiguous(1,MPI_DOUBLE,&MPI_CXSC_REAL))
11                                                    !=MPI_SUCCESS)
12     return err;
13   if ((err=MPI_Type_commit(&MPI_CXSC_REAL))!=MPI_SUCCESS)
14     return err;
15   if ((err=MPI_Type_contiguous(2,MPI_CXSC_REAL,&MPI_CXSC_COMPLEX))
```

```
16                                                              !=MPI_SUCCESS)
17      return err;
18    if ((err=MPI_Type_commit(&MPI_CXSC_COMPLEX))!=MPI_SUCCESS)
19      return err;
20    if ((err=MPI_Type_contiguous(2,MPI_CXSC_REAL,&MPI_CXSC_INTERVAL))
21                                                              !=MPI_SUCCESS)
22      return err;
23    if ((err=MPI_Type_commit(&MPI_CXSC_INTERVAL))!=MPI_SUCCESS)
24      return err;
25    if ((err=MPI_Type_contiguous(2,MPI_CXSC_INTERVAL,&MPI_CXSC_CINTERVAL))
26                                                              !=MPI_SUCCESS)
27      return err;
28    if ((err=MPI_Type_commit(&MPI_CXSC_CINTERVAL))!=MPI_SUCCESS)
29      return err;
30    MPI_CXSC_TYPES_DEFINED=true;
31    return err;
32  }
```

Listing 3.37: Definition der C-XSC Grunddatentypen für MPI

Mit Hilfe der MPI-Funktion `MPI_Type_contiguous` werden hier die entsprechenden Grunddatentypen definiert:

- `MPI_CXSC_REAL`: Der Datentyp `real`, wird als eine `double`-Variable definiert.
- `MPI_CXSC_INTERVAL`: Der Datentyp `interval`, wird als zwei `double`-Variablen (eine für das Infimum und eine für das Supremum) definiert .
- `MPI_CXSC_COMPLEX`: Der Datentyp `complex`, wird als zwei `double`-Variablen (eine für den Real- und eine für den Imaginärteil) definiert.
- `MPI_CXSC_CINTERVAL`: Der Datentyp `cinterval`, wird als zwei `interval`-Variablen (eine für den Real- und eine für den Imaginärteil) definiert.

Mittels der Funktion `MPI_Type_commit` wird der neu definierte Datentyp angemeldet und kann danach verwendet werden. Die globale Variable `MPI_CXSC_TYPES_DEFINED` hält fest, ob die Grunddatentypen bereits definiert wurden. Zu Beginn jeder Kommunikationsroutine aus der C-XSC MPI-Schnittstelle wird diese Variable abgefragt und bei Bedarf die Funktion `MPI_Define_CXSC_Types` aufgerufen.

Die Kommunikationsfunktionen für alle anderen Datentypen, insbesondere solche zur Speicherung von Vektoren und Matrizen, werden über den *Pack* und *Unpack* Mechanismus von MPI realisiert. Hierbei werden Daten vom Sender in einem Kommunikationspuffer in einer bestimmten Reihenfolge zwischengespeichert (gepackt), dann von MPI als ein großer Datenblock versendet und schließlich vom Empfänger in der Reihenfolge der Speicherung aus dem Puffer ausgelesen und in einer Variablen des betreffenden Typs gespeichert. Auf diese Art können also im Prinzip Daten beliebiger Größe und beliebigen Typs versendet werden.

In der C-XSC MPI-Schnittstelle gibt es daher Implementierungen der Funktionen `MPI_Pack` und `MPI_Unpack` für alle Datentypen aus der Kernbibliothek von C-XSC, also

auch für die Matrix- und Vektordatentypen sowie für die `dotprecision`-Datentypen. Ebenso gibt es eine entsprechende Implementierung für die Klasse `vector` aus der Standard Template Library (STL), die ein nützliches Werkzeug bei der Verwaltung der Daten in parallelen Programmen sein kann. Weiterhin werden diese Routinen auch gebraucht, um die entsprechenden Attribute der dünn besetzten Datentypen (siehe Abschnitt 3.3) mit `MPI_Pack` verwenden zu können.

Wichtig ist dabei zunächst, den Kommunikationspuffer mit einer ausreichenden Größe zu initialisieren. Dazu steht eine Funktion `init_CXSC_MPI` zur Verfügung, welche den Puffer mit einer Standardgröße initialisiert. Optional kann der Funktion auch ein `int`-Parameter für die Größe des Puffers übergeben werden. Vor der Benutzung der MPI-Schnittstelle muss diese Funktion einmal aufgerufen werden, damit der Puffer zur Verfügung steht. Wird er nicht mehr benötigt, so kann er über die Funktion `finish_CXSC_MPI` wieder freigegeben werden.

Als Beispiel für die Implementierung der Funktionen für das Packen und Entpacken der C-XSC Datentypen werden die entsprechenden Versionen für den Datentyp `rmatrix` betrachtet. Listing 3.38 zeigt zunächst die `MPI_Pack` Funktion.

```
1  int MPI_Pack (rmatrix& rm, void* buff, int bufflen, int* pos,
2                                                    MPI_Comm MC) {
3    int err;
4
5    int lb1=Lb(rm,1);
6    int lb2=Lb(rm,2);
7    int ub1=Ub(rm,1);
8    int ub2=Ub(rm,2);
9
10   if (!MPI_CXSC_TYPES_DEFINED)
11      if ((err=MPI_Define_CXSC_Types())!=MPI_SUCCESS)
12         return err;
13
14   if ((err=MPI_Pack(&lb1,1,MPI_INT,buff,bufflen,pos,MC))!=MPI_SUCCESS)
15     return err;
16   if ((err=MPI_Pack(&ub1,1,MPI_INT,buff,bufflen,pos,MC))!=MPI_SUCCESS)
17     return err;
18   if ((err=MPI_Pack(&lb2,1,MPI_INT,buff,bufflen,pos,MC))!=MPI_SUCCESS)
19     return err;
20   if ((err=MPI_Pack(&ub2,1,MPI_INT,buff,bufflen,pos,MC))!=MPI_SUCCESS)
21     return err;
22   err=MPI_Pack(&rm[lb1][lb2],(ub1-lb1+1)*(ub2-lb2+1),MPI_CXSC_REAL,buff,
23            bufflen,pos,MC);
24
25   return err;
26 }
```

Listing 3.38: Implementierung von `MPI_Pack` für den Datentyp `rmatrix`

Als erstes werden hier die Attribute `lb1`, `lb2`, `ub1` und `ub2` zur Speicherung der Indizes in den Kommunikationspuffer geschrieben. Danach folgen die Elemente der Matrix, für welche der zuvor definierte MPI-Datentyp `MPI_CXSC_REAL` verwendet wird. Listing 3.39 zeigt nun die Funktion zum Entpacken der Daten.

```
1  int MPI_Unpack (void* buff, int bufflen, int* pos, rmatrix& rm,
2                                                    MPI_Comm MC) {
3    int err;
4    int mlen1, mlen2, lb1, ub1, lb2, ub2;
5
6    if (!MPI_CXSC_TYPES_DEFINED)
7       if ((err=MPI_Define_CXSC_Types())!=MPI_SUCCESS)
8          return err;
9
10   if ((err=MPI_Unpack(buff, bufflen, pos, &lb1, 1, MPI_INT, MC))
11         !=MPI_SUCCESS)
12     return err;
13   if ((err=MPI_Unpack(buff, bufflen, pos, &ub1, 1, MPI_INT, MC))
14         !=MPI_SUCCESS)
15     return err;
16   if ((err=MPI_Unpack(buff, bufflen, pos, &lb2, 1, MPI_INT, MC))
17         !=MPI_SUCCESS)
18     return err;
19   if ((err=MPI_Unpack(buff, bufflen, pos, &ub2, 1, MPI_INT, MC))
20         !=MPI_SUCCESS)
21     return err;
22   mlen1=ub1-lb1+1;
23   mlen2=ub2-lb2+1;
24   Resize(rm,mlen1,mlen2);
25   SetLb(rm,1,lb1);
26   SetLb(rm,2,lb2);
27
28   err=MPI_Unpack(buff, bufflen, pos, &rm[lb1][lb2],
29               mlen1*mlen2, MPI_CXSC_REAL, MC);
30   return err;
31 }
```

Listing 3.39: Implementierung von `MPI_Unpack` für den Datentyp `rmatrix`

Beim Entpacken ist darauf zu achten, dass die Daten in der gleichen Reihenfolge aus dem Puffer ausgelesen werden, in der sie zuvor hineingeschrieben wurden. Es müssen also zunächst die Indizierungsvariablen ausgelesen und die Matrix entsprechend dimensioniert werden. Erst danach werden die Daten zu den Elementen der Matrix ausgelesen und direkt an die zugehörige Speicherstelle geschrieben. Zusätzlich gibt es für alle Matrix- und Vektorversionen von `MPI_Pack` und `MPI_Unpack` auch Varianten, mit denen nur ein bestimmter Ausschnitt aus der Matrix bzw. dem Vektor gepackt und entpackt wird, was z.B. beim Versenden von Teilmatrizen, einem häufigen Anwendungsfall, nützlich ist.

Die Funktionen zum Packen und Entpacken für die anderen Datentypen funktionieren analog zu obigen Beispielen. Die einzelnen Attribute der jeweiligen Klasse werden beim Packen nacheinander in den Kommunikationspuffer geschrieben und beim Entpacken in der gleichen Reihenfolge wieder ausgelesen und in einem passenden C-XSC Objekt gespeichert. Mit Hilfe der beschriebenen `MPI_Pack`- und `MPI_Unpack`-Funktionen für alle C-XSC Datentypen können nun Implementierungen der Kommunikationsfunktionen von MPI erstellt werden.

Diese Funktionen haben unabhängig vom betreffenden Datentyp stehts den gleichen

grundlegenden Ablauf:

- Beim Senden: Aufruf von `MPI_Pack` für den betreffenden Datentyp.
- Verwendung der entsprechenden Kommunikationsfunktion für den Kommunikationspuffer (der Puffer liegt als `int`-Array vor, es wird also eine entsprechende Anzahl von `MPI_INT` Elementen versendet). Dabei wird nicht der komplette Kommunikationspuffer versendet, sondern nur der tatsächlich belegte Anteil.
- Beim Empfangen: Aufruf von `MPI_Unpack` für den betreffenden Datentyp und Speicherung der Daten in einem passenden Objekt.

Als Beispiel für die Implementierung werden die beiden wohl am häufigsten verwendeten MPI-Funktionen betrachtet, `MPI_Send` und `MPI_Recv`. Listing 3.40 zeigt die entsprechenden Implementierungen.

```
1  template<class T>
2  int MPI_Send(T& Tobj, int i1, int i2, MPI_Comm MC) {
3    int pos=0;
4    int err;
5
6    if ((err=MPI_Pack(Tobj, commbuffer, MPI_CXSC_BUFFERLEN, &pos, MC))
7                                                          !=MPI_SUCCESS)
8      return err;
9
10   err=MPI_Send((void*)commbuffer, pos, MPI_PACKED, i1, i2, MC);
11
12   return err;
13 }
14
15 template<class T>
16 int MPI_Recv(T& Tobj, int i1, int i2, MPI_Comm MC, MPI_Status* MS) {
17   int pos=0;
18   int err;
19
20   if ((err=MPI_Recv((void*)commbuffer, MPI_CXSC_BUFFERLEN, MPI_PACKED,
21                     i1, i2, MC, MS))!= MPI_SUCCESS)
22     return err;
23
24   err=MPI_Unpack(commbuffer, MPI_CXSC_BUFFERLEN, &pos, Tobj, MC);
25
26   return err;
27 }
```

Listing 3.40: Template-Implementierung von `MPI_Send` und `MPI_Recv` für C-XSC Datentypen

Die Funktion `MPI_Send` für C-XSC-Datentypen bekommt also das zu sendende Objekt `Tobj`, den Empfänger `i1`, einen Nachrichten-Tag `i2` sowie den betreffenden MPI-Kommunikator `mc` (MPI Kommunikatoren sind eine Untermenge der zur Verfügung stehenden Knoten, welche für diese Kommunikation benutzt werden). Die Funktion

`MPI_Recv` erhält ein Objekt `Tobj`, welches die empfangenen Daten aufnimmt, den Empfänger `i1`, einen Nachrichtentag `i2`, den betreffenden MPI-Kommunikator `mc` sowie einen Zeiger auf eine Statusvariable `MS` (in dieser wird der Status der Übertragung zurückgegeben) übergeben. Beide gehen dann wie oben beschrieben vor.

Diese Funktionen können so für alle C-XSC Datentypen benutzt werden. Für die vier Grunddatentypen `real`, `interval`, `complex` und `cinterval` gibt es allerdings effizientere Spezialisierungen dieser Template-Funktionen, da hier der Pack/Unpack-Mechanismus nicht verwendet werden muss. Stattdessen können diese Datentypen, da für sie einfache MPI-Datentypen definiert werden können, direkt die Standard-Kommunikationsfunktionen nutzen.

Zum Abschluss dieses Abschnitts zeigt Listing 3.41 ein kleines Beispiel für die Benutzung der MPI-Schnittstelle von C-XSC. Hierbei wird eine Intervallmatrix (Typ `imatrix`) komplett von Prozess 0 an Prozess 1 versendet und dann ausgegeben. Zu sehen ist neben der Initialisierung von MPI selbst auch die Initialisierung und die abschließende Freigabe des Kommunikationspuffers.

```
1  #include <iostream>
2  #include <imatrix.hpp>
3  #include "cxsc_mpicomm.hpp" //Header fuer die MPI-Schnittstelle
4
5  using namespace std;
6  using namespace cxsc;
7
8  int main(int argc, char *argv[]) {
9
10   int mypid;  //ID des Prozesses
11   MPI_Status status;
12
13   // MPI Initialisierung
14   MPI_Init(&argc, &argv);
15
16   //Eigene ID bestimmen
17   MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
18
19   //Kommunikationspuffer initialisieren
20   //Groesse kann auch als Parameter uebergeben werden,
21   //ein zu kleiner Kommunikationspuffer fuehrt zum Abbruch
22   //des Programms
23   init_CXSC_MPI();
24
25   imatrix A(5,5);
26   A = 0.0;
27
28   if (mypid==0) {
29
30      //Matrix mit Werten fuellen
31      for(int i=1 ; i<=5 ; i++)
32        A[i][i] = interval(i,i+1);
33
34      cout << "Matrix gespeichert von Prozess 0:" << endl;
```

```
35        cout << A << endl;
36
37        //Matrix an Prozess 1 senden
38        MPI_Send(A, 1, 0, MPI_COMM_WORLD);
39
40    } else if(mypid==1) {
41
42        //Matrix A von Prozess 0 empfangen
43        MPI_Recv(A, 0, 0, MPI_COMM_WORLD, &status);
44
45        cout << "Matrix empfangen von Prozess 1:" << endl;
46        cout << A << endl;
47    }
48
49    MPI_Finalize();
50
51    //Kommunikationspuffer loeschen
52    finish_CXSC_MPI();
53
54    return 0;
55 }
```

Listing 3.41: Beispiel zur Benutzung der MPI-Schnittstelle von C-XSC

Insgesamt steht also mit der MPI-Schnittstelle von C-XSC eine komfortable Möglichkeit bereit, C-XSC-Programme in einem Distributed-Memory Kontext zu erstellen. Gegenüber der ursprünglichen Implementierung stehen nun auch Kommunikationsfunktionen für die neuen dünn besetzten Datentypen zur Verfügung. Auch werden weitere Ergänzungen der C-XSC Bibliothek wie die Skalarprodukte in K-facher Arbeitsgenauigkeit berücksichtigt. Durch das Hinzufügen von Funktionen zum Versenden von Teilvektoren und -matrizen wurde die Schnittstelle außerdem etwas komfortabler und flexibler.

3.5. Vorbereitungen für Multithreading-Anwendungen

Die C-XSC Bibliothek ist bereits mehr als 20 Jahre alt. Entsprechend waren Mehrkern- bzw. Mehrprozessor-Systeme und somit die Entwicklung von mittels Multithreading für Shared-Memory-Systeme parallelisierten Programmen während der Entwicklung von C-XSC lange Zeit kein entscheidendes Thema. Heute ist die Parallelisierung von Programmen durch Einführung von Nebenläufigkeiten entscheidend, um die gesamte Rechenleistung eines Systems effektiv ausnutzen zu können. Aus diesem Grund ist eine Überarbeitung der C-XSC-Bibliothek im Hinblick auf Threadsicherheit von großer Bedeutung.

Das größte Problem stellt dabei die starke Verwendung von globalen Variablen innerhalb der C-XSC Bibliothek dar. Da sich bei Multithreading-Anwendungen alle Threads den gleichen Speicher teilen, müssen schreibende Zugriffe auf Daten im Speicher synchronisiert werden (siehe Abschnitt 2.2.1). Problematisch waren bzw. sind insbesondere folgende Bereiche:

1. Globale `dotprecision` Variablen zur Verwendung in Operatoren,

2. Globale `dotprecision`-Variablen bei Staggered-Datentypen,
3. Globale Konfigurationsvariablen bei der automatischen Differentiation,
4. Globale Ein- und Ausgabeflags,
5. Globale Konfigurationsvariablen für Skalarprodukt- und Staggered-Präzision.

Diese Punkte werden im Folgenden kurz erläutert und es wird aufgezeigt, wie die jeweiligen Probleme behoben wurden oder, falls eine Behebung des Problems nicht ohne weiteres möglich ist, wie sie vom Programmierer umgangen werden können.

Die ersten beiden Punkte betreffen ein ganz ähnliches Problem. Sowohl bei Berechnungen innerhalb von Matrix/Vektor-Operatoren, welche ein oder mehrere Skalarprodukte berechnen, als auch bei einigen Berechnungen innerhalb der in C-XSC enthaltenen Staggered-Datentypen wie `l_real`, `l_interval` etc. werden Berechnungen mit Hilfe der `dotprecision`-Datentypen durchgeführt. Bei den Matrix/Vektor-Operatoren werden z.B. die Skalarprodukte über ein `dotprecision`-Objekt berechnet.

Da der von den `dotprecision`-Objekten verwaltete lange Akkumulator im Vergleich zu einfachen Datentypen wie `double` bzw. `real` sehr groß ist (529 Byte gegenüber 8 Byte), wurden in der ursprünglichen Implementierung globale `dotprecision`-Variablen angelegt, welche dann bei allen Berechnungen verwendet wurden. Dies diente dazu, das Anlegen lokaler `dotprecision`-Variablen bei jedem Aufruf der betroffenen Funktionen zu vermeiden, was sich auf den Rechnern der damaligen Zeit noch spürbar auswirken konnte. Problematisch wird es nun, wenn parallel in mehreren Threads z.B. unterschiedliche Matrix-Vektor-Produkte berechnet werden sollen, die für ihre Berechnung aber alle auf die gleiche globale `dotprecision`-Variable zugreifen. Dies führt zu undefiniertem Verhalten, welches sich in der Praxis oft in Speicherzugriffsfehlern oder schlimmstenfalls in falschen Ergebnissen äußert.

Auf heutigen Systemen ist der durch das Anlegen lokaler `dotprecision`-Variablen verursachte Mehraufwand vernachlässigbar. Daher werden zur Behebung dieses Problems alle globalen `dotprecision`-Variablen entfernt und durch temporäre, lokale Variablen ersetzt.

Punkt drei betrifft ein Problem mit globalen Konfigurationsvariablen, welche innerhalb der Module zur automatischen Differentiation und zur Hessearithmetik in der C-XSC Toolbox verwendet werden. Die automatische Differentiation, bzw. für mehrdimensionale Probleme die Hessearithmetik, werden insbesondere in globalen Optimierungsalgorithmen verwendet, welche sich in der Regel sehr gut parallelisieren lassen. Daher ist in diesem Bereich die Threadsicherheit von großer Bedeutung.

In beiden Modulen wird der jeweils zu verwendende Differenzierungsgrad in einer globalen Variablen zwischengespeichert, welche dann von den jeweiligen Berechnungsfunktionen ausgelesen wird. Im Modul zur automatischen Differentiation ist dies die Variable `DerivOrder`. Innerhalb des Moduls stehen die drei Funktionen `fEval`, `dfEval` und `ddfEval` zur Verfügung, welche jeweils den Funktionswert, den Wert der ersten Ableitung und den Funktionswert sowie die Werte der ersten und zweiten Ableitung und den Funktionswert an der jeweils übergebenen Stelle berechnen. Jede dieser Funktionen

setzt dabei den Wert von `DerivOrder` entsprechend um. Werden diese Funktionen parallel in mehreren Threads ausgeführt, so überschreiben sie sich gegenseitig den aktuell benötigten Grad, wodurch es insbesondere bei Aufruf von `dfEval` und `ddfEval` dazu kommen kann, dass die Funktionswerte der ersten bzw. zweiten Ableitung gar nicht berechnet werden.

Dieses Problem wurde mittels der sogenannten *Thread Local Storage* (TLS) gelöst, welche alle aktuellen Compiler beherrschen (eine Ausnahme bildet Apples Version des GNU-Compilers unter Mac OS X). Thread Local Storage sorgt dafür, dass jeder Thread automatisch mit einer lokalen Kopie der betreffenden globalen Variable arbeitet. So können die oben beschriebenen Konflikte nicht auftreten. Bei der Deklaration der Variablen muss dazu üblicherweise das Schlüsselwort `__thread` dem Datentyp vorangestellt werden (bei Verwendung des Microsoft Visual C++ Compilers muss das Schlüsselwort `__declspec(thread)` verwendet werden). Im neuen C++-Standard C++11 [66] ist eine entsprechende Möglichkeit zur Deklarierung einer Variablen als Thread Local verpflichtend vorgegeben, so dass in naher Zukunft auch alle Compiler diese Funktionalität unterstützen sollten.

Dieses Vorgehen kann bei Punkt fünf leider nicht verwendet werden, obwohl das Problem ähnlich gelagert ist. Für die zu verwendende Präzision bei Skalarproduktberechnungen über die in Abschnitt 3.1.3 erläuterte globale Variable `opdotprec` ebenso wie für die Variable zur Steuerung der Präzision der Staggered-Datentypen (`stagprec`) kann zwar prinzipiell auch TLS genutzt werden, allerdings entsteht so ein entscheidender Nachteil: Da jeder Thread seine eigene Kopie der globalen Variable verwendet und diese Kopien schon bei Programmstart erzeugt werden (und nicht etwa erst bei Eintritt in eine parallele Region des Programms), haben Änderungen außerhalb paralleler Regionen nur Einfluss auf den Wert der lokalen Kopie für den Masterthread. Listing 3.42 zeigt ein entsprechendes Beispielprogramm unter Verwendung von OpenMP.

```
1  #include <stdio>
2  #include <rmatrix.hpp>
3  #include <openmp.h>
4
5  using namespace std;
6  using namespace cxsc;
7
8  int main(int argc, char *argv[]) {
9    opdotprec = 2;
10
11   #pragma omp parallel
12   {
13     int id = omp_get_thread_num();
14     printf("%d: %d\n", id, opdotprec);
15   }
16
17   return 0;
18 }
```

Listing 3.42: Beispiel zu den Problemen mit globalen Konfigurationsvariablen bei Verwendung von Thread Local Storage

Hier wird die Genauigkeit für Skalarprodukte zunächst scheinbar global auf zwei gesetzt. Bei Verwendung von Thread Local Storage für die globale Variable `opdotprec` würde man so aber nur den Wert der Kopie von Thread 0 ändern. Die Ausgabe dieses Programms ist also bei Verwendung von vier Threads (die Reihenfolge der Zeilen hängt vom Scheduling des Betriebssystems ab):

```
0: 2
1: 0
2: 0
3: 0
```

Die lokalen Kopien der anderen Threads werden also nicht geändert und bleiben in diesem Fall auf den Standardwert 0 gesetzt. Da dies in den meisten Fällen nicht dem gewünschten Verhalten entsprechen dürfte, wird hier standardmäßig auf die Benutzung von Thread Local Storage verzichtet. Der Programmierer muss also selber darauf achten, die beiden globalen Konfigurationsvariablen `opdotprec` und `stagprec` korrekt zu verwenden. Optional kann aber durch Setzen der Präprozessorvariable `CXSC_USE_TLC` Thread Local Storage auch für diesen Fall explizit aktiviert werden, wobei dann aber auf die oben angesprochenen Probleme geachtet werden muss.

Schließlich bleibt noch Punkt vier, die globalen Ein- und Ausgabeflags. C-XSC bietet Ein- und Ausgabeoperatoren gemäß dem Stream-Konzept von C++. Diese können, ähnlich wie die Ein- und Ausgaben aus der C++-Standard-Bibliothek, über Flags konfiguriert werden (z.B. Zahl der Nachkommastellen, hexadezimale Ausgabe, etc.). Diese Flags werden allerdings in globalen Variablen gespeichert, wodurch es bei Multithreading-Anwendungen zu Konflikten kommen kann.

Die Verwendung von Streams führt allgemein zu Problemen bei parallelen Anwendungen. Da es in der Natur von Streams liegt, dass Nachrichten nicht auf einmal, sondern in einzelnen Abschnitten nach und nach in den Ausgabestrom geschrieben werden, kann es bei gleichzeitigem Zugriff auf den selben Strom aus unterschiedlichen Threads heraus zu Ausgaben kommen, in denen die Daten aus den verschiedenen Threads vermischt sind. Aus diesen Gründen wird in der parallelen Programmierung allgemein eher auf die C-Funktionen zur Ein- und Ausgabe, wie z.B. `printf` zurückgegriffen. Daher wurde auch in C-XSC nichts an den globalen IO-Flags geändert. Bei der Erstellung von Programmen mit mehreren Threads sollte man sich dieser Problematik allerdings bewusst sein und die Verwendung von Streams möglichst vermeiden.

Für die Ausgabe der C-XSC Datentypen sollte daher innerhalb von Threads die Funktion `printf` verwendet werden. Da bei der Ausgabe von Intervallen auf eine korrekte Rundung geachtet werden muss, sollten Ausgaben dabei im Hexadezimalsystem erfolgen. Dazu steht die Funktion `realToHex` bereit, welche ein Objekt vom Datentyp `real` in einen String mit entsprechender hexadezimaler Darstellung umwandelt. Listing 3.43 demonstriert die Verwendung dieser Funktion und zeigt dabei, wie die vier C-XSC Grunddatentypen `real`, `interval`, `complex` und `cinterval` mittels `printf` ausgegeben werden können.

```
1 #include <cstdio>
```

```
2  #include <cinterval.hpp>
3  #include <omp.h>
4
5  using namespace cxsc;
6
7  int main() {
8    real r = 2.0;
9    interval i(2.0,4.4);
10   complex c(1.0,2.0);
11   cinterval ci(i,-i);
12
13   //Parallele Ausgabe
14   #pragma omp parallel
15   {
16     int id = omp_get_thread_num();
17     //real output
18     printf("%d: %s \n", id, realToHex(r).c_str());
19     //interval output
20     printf("%d: [%s,%s] \n", id, realToHex(Inf(i)).c_str(),
21                                 realToHex(Sup(i)).c_str());
22     //complex output
23     printf("%d: (%s,%s) \n", id, realToHex(Re(c)).c_str(),
24                                 realToHex(Im(c)).c_str());
25     //cinterval output
26     printf("%d: ([%s,%s],[%s,%s]) \n", id, realToHex(InfRe(ci)).c_str(),
27                                           realToHex(SupRe(ci)).c_str(),
28                                           realToHex(InfIm(ci)).c_str(),
29                                           realToHex(SupIm(ci)).c_str());
30   }
31
32   return 0;
33 }
```

Listing 3.43: Korrekte Ausgabe der C-XSC-Grunddatentypen mittels `printf`

Mit den in diesem Abschnitt beschriebenen Änderungen sind die wesentlichen Probleme mit C-XSC bei der Erstellung von Multithreading-Anweisungen behoben. Es ist nun also möglich, unter Berücksichtigung der hier besprochenen Fallstricke, threadsichere und performante parallele C-XSC-Anwendungen für Shared-Memory-Systeme, z.B. unter Verwendung von OpenMP, zu erstellen. Beispiele dafür finden sich in den in den folgenden Kapiteln beschriebenen Lösern für lineare Gleichungssysteme und im nächsten Abschnitt.

3.5.1. OpenMP Parallelisierung einiger Operatoren

Durch die im letzten Abschnitt beschriebenen Änderungen ist es nun möglich, Funktionen aus der C-XSC Kernbibliothek zu parallelisieren. Hierfür bieten sich vor allem einige der Operatoren für dicht besetzte Matrizen und Vektoren an, da bei diesen zum einen durch die Mehrdimensionalität ein erhöhter Rechenaufwand entsteht, zum anderen eine Parallelisierung normalerweise ohne großen Aufwand über eine Aufteilung von

for-Schleifen auf die verfügbaren Threads implementierbar ist.

Im Rahmen dieser Arbeit werden die Operatoren für Matrix-Vektor sowie Matrix-Matrix Produkte für dicht besetzte Vektoren und Matrizen mittels OpenMP parallelisiert. Diese Operationen profitieren, insbesondere falls die in Abschnitt 3.1 beschriebenen Skalarprodukte mit höherer Genauigkeit verwendet werden, besonders stark von einer Parallelisierung. Bei Verwendung der in Abschnitt 3.2 beschriebenen BLAS Unterstützung wird keine explizite Parallelisierung verwendet, da moderne BLAS Bibliotheken in der Regel von sich aus bereits parallelisiert sind.

Die Parallelisierung dieser Operationen mit OpenMP kann in der Regel über die Parallelisierung der äußeren for-Schleife der Berechnung mittels des entsprechenden OpenMP Pragmas erfolgen. C-XSC verwendet in den Operatoren der dicht besetzten Datentypen Template-Funktionen, welche die eigentliche Berechnung durchführen. Diese werden in der Datei `matrix.inl` definiert, welche zusammen mit dem jeweiligen Header eingebunden wird. Dadurch kann die Entscheidung, ob diese OpenMP Unterstützung genutzt werden soll, durch den Benutzer erst bei der Kompilierung eines C-XSC-Programms erfolgen, es muss also nicht bereits bei der Installation der C-XSC Bibliothek eine Entscheidung getroffen werden. Wie dies funktioniert und wann das Ein- bzw. Ausschalten der Parallelisierung sinnvoll ist, wird im Folgenden noch genauer erläutert.

Zunächst soll als Beispiel für das Vorgehen bei der Parallelisierung die Template-Funktion zur Berechnung des Produktes zweier reeller Punktmatrizen betrachtet werden. Die entsprechende Funktion `_mmmult` wird in Listing 3.44 angegeben.

```
1  template <class M1, class M2, class E>
2  TINLINE E _mmmult(const M1 &m1, const M2 &m2) {
3    E r(m1.lb1,m1.ub1,m2.lb2,m2.ub2);
4
5    if(opdotprec == 1) {
6      #ifdef CXSC_USE_BLAS
7        blasmatmul(m1,m2,r);
8        return r;
9      #else
10       r = 0.0;
11       #if defined(_OPENMP) && defined(CXSC_USE_OPENMP)
12       #pragma omp parallel for
13       #endif
14       for(int i=0 ; i<m1.ysize ; i++)
15         for(int k=0 ; k<m1.xsize ; k++)
16           for(int j=0 ; j<m2.xsize ; j++)
17             r.dat[i*m2.xsize+j] +=
18                     m1.dat[i*m1.xsize+k] * m2.dat[k*m2.xsize+j];
19       return r;
20     #endif
21   } else {
22     dotprecision dot(0.0);
23     dot.set_k(opdotprec);
24
25     #if defined(_OPENMP) && defined(CXSC_USE_OPENMP)
26     #pragma omp parallel for firstprivate(dot)
27     #endif
```

```
28        for(int i=0;i<m1.ysize;i++) {
29          for(int j=0;j<m2.xsize;j++) {
30            dot=0.0;
31            accumulate_approx(dot,m1[i+Lb(m1,ROW)],m2[Col(j+Lb(m2,COL))]);
32            r.dat[i*m2.xsize+j]=rnd(dot);
33          }
34        }
35
36        return r;
37      }
38    }
```

Listing 3.44: Parallelisierung der Berechnungsfunktion für das Produkt zweier reeller Punktmatrizen

Wie zu sehen ist, wird bei Aktivierung der BLAS-Unterstützung und einfacher Genauigkeit die entsprechende Funktion aus der BLAS-Schnittstelle aufgerufen, es erfolgt also keine direkte Parallelisierung. Für einfache Genauigkeit ohne BLAS-Unterstützung wird das Produkt über drei for-Schleifen berechnet, von denen die äußere mittels OpenMP über das entsprechende Pragma direkt parallelisiert werden kann.

Für alle anderen Fälle erfolgt die Berechnung über ein `dotprecision`-Objekt. Auch hier wird die äußere for-Schleife direkt mittels OpenMP parallelisiert. Zusätzlich zu beachten ist hier nur, dass jeder Thread mit einer lokalen Kopie der `dotprecision`-Variable arbeitet, welche mit dem Objekt aus dem Master-Thread initialisiert wird. Dies ist nötig, damit jeder Thread die gleiche Präzision für die Skalarprodukt-Berechnungen verwendet, welche zuvor mittels der Funktion `set_k` gesetzt wurde. Daher muss die `dotprecision`-Variable `dot` hier als `firstprivate` deklariert werden.

Weiterhin ist zu sehen, dass die OpenMP-Unterstützung nur aktiviert wird, falls OpenMP vom Compiler unterstützt wird bzw. die entsprechende Option aktiviert wurde und falls zusätzlich die Präprozessorvariable `CXSC_USE_OPENMP` gesetzt wurde. Auf diese Weise kann die OpenMP-Unterstützung für die parallelisierten C-XSC Operatoren bei Kompilierung des eigenen Programms durch den Benutzer explizit aktiviert oder nicht aktiviert werden.

Die Aktivierung der Parallelisierung kann von Nachteil sein, falls sehr viele sehr kleine Produkte berechnet werden müssen. Bei einem Matrix-Vektor oder Matrix-Matrix-Produkt sehr kleiner Dimension kann der Verwaltungsaufwand für das Multithreading den Geschwindigkeitsgewinn durch die Parallelisierung aufheben oder sogar übersteigen. In solchen Fällen ist es meist besser, die Parallelisierung auf einer höheren Ebene anzusetzen. Wird also z.B. in einem Programm eine for-Schleife ausgeführt, in welcher sehr viele sehr kleine (und voneinander unabhängige) Matrix-Matrix-Produkte berechnet werden, so macht es meist mehr Sinn, diese for-Schleife selbst anstatt die einzelnen Produkte zu parallelisieren.

Zum Abschluss dieses Abschnitts soll nun noch kurz ein Laufzeittest mit dem parallelisierten Operator für das Produkt zweier reeller Punktmatrizen der Dimension $n = 1000$ durchgeführt werden, welcher den Gewinn durch die Parallelisierung verdeutlichen soll, insbesondere bei Verwendung von Skalarprodukten höherer Genauigkeit. Zum Vergleich wird dabei die Referenzimplementierung der XBLAS Bibliothek [90] in Version 1.0.248

Methode	$P = 1$	$P = 2$	$P = 4$	$P = 8$	**Relativer Fehler**
`XBLAS`	25.5	–	–	–	2.33
`C-XSC`, $K = 1$	5.2	2.6	1.3	0.7	$9.05 \cdot 10^{15}$
`C-XSC`, $K = 2$	15.9	8.1	4.1	2.1	24.2
`C-XSC`, $K = 3$	30.8	15.4	7.6	4.0	0.00
`C-XSC`, $K = 0$	105.1	53.1	26.4	13.7	0.00

Tabelle 3.10.: Testergebnisse für reelles Matrix-Matrix-Produkt der Dimension $n = 1000$ mit OpenMP Unterstützung

herangezogen, welche eine entsprechende Operation in (ungefähr) doppelter Genauigkeit anbietet. Die beiden Matrizen für das Produkt werden dabei so angelegt, dass jedes einzelne Skalarprodukt eine Kondition von etwa10^{30} aufweist. Tabelle 3.10 zeigt die Ergebnisse.

Die Tabelle zeigt die Laufzeiten für das mit C-XSC berechnete Matrix-Matrix-Produkt für die Präzision $K = 0, 1, 2, 3$ sowie mit $P = 1, 2, 4, 8$ Threads. Die Tests erfolgen auch hier wieder auf einem System mit 2 Intel Xeon E5520 CPUs mit jeweils 4 mit 2.27 GHz getakteten Kernen sowie 24 GB Hauptspeicher (DDR3). Da XBLAS kein Multithreading aufweist, werden hier nur die Ergebnisse für $P = 1$ angegeben. Weiterhin wird der durchschnittliche relative Fehler der Ergebnismatrix aufgeführt. Da das Ergebnis hier unabhängig von der Anzahl der Threads P ist, wird der relative Fehler nur einmal für jede Berechnungsmethode angegeben.

Zunächst ist zu sehen, dass XBLAS und C-XSC für $K = 2$ in der Tat ein Ergebnis von etwa doppelter Genauigkeit liefern (bei Kondition 10^{30} ist ein relativer Fehler nahe 1 zu erwarten). Allerdings ist C-XSC auch bei Verwendung von nur einem Thread deutlich schneller als XBLAS, da der DotK-Algorithmus weniger Operationen als die von XBLAS verwendete Methode benötigt [106]. Wird die Präzision erhöht ($K = 3$ oder $K = 0$) so ist das Ergebnis hier maximal genau und der relative Fehler fällt auf 0.

Die Parallelisierung zeigt für alle Genauigkeiten nahezu idealen Speed-Up, d.h. bei Einsatz von z.B. 8 Threads und 8 zur Verfügung stehenden Kernen ist in der Regel eine Beschleunigung um den Faktor 8 zu erwarten. Für diesen Testfall bedeutet dies, dass ein Ergebnis von ähnlicher Qualität wie XBLAS (also berechnet mit $K = 2$) bei Aktivierung der OpenMP-Parallelisierung und Nutzung von 8 Kernen mehr als zehnmal schneller berechnet werden kann (2.1 gegenüber 25.5 Sekunden). C-XSC stellt also eine sehr schnelle Methode zur Berechnung von Skalarproduktausdrücken in höherer Genauigkeit zur Verfügung.

4. Verifizierende Löser für dicht besetzte lineare Gleichungssysteme

In diesem Kapitel werden Löser für dicht besetzte lineare (Intervall-)Gleichungssysteme beschrieben, welche viele der in Kapitel 3 erläuterten Erweiterungen der C-XSC Bibliothek verwenden, um möglichst schnell möglichst genaue Intervalleinschließungen der Lösungsmenge solcher Systeme zu berechnen. Für die hier beschriebenen Löser besonders von Bedeutung sind dabei die wählbare Genauigkeit für Skalarproduktberechnungen und die Unterstützung von BLAS- und LAPACK-Routinen.

Als erstes wird in diesem Kapitel auf verifizierende Löser für allgemeine dicht besetzte Gleichungssysteme eingegangen. Dabei werden zunächst einige theoretische Überlegungen angestellt, welche auch für die meisten der anderen Löser in dieser Arbeit relevant sind, und der verwendete Lösungsalgorithmus wird genauer erläutert. Danach folgt eine Beschreibung der Implementierung, wobei besonders auf die Verwendung der neuen C-XSC Erweiterungen und die Parallelisierung eingegangen wird. Neben der über OpenMP parallelisierten Version für Shared-Memory-Systeme wird außerdem eine mittels MPI parallelisierte Version für Distributed-Memory-Systeme beschrieben, welche das verifizierte Lösen sehr großer dicht besetzter Gleichungssysteme ermöglicht. Abschließend werden diverse Tests und Zeitmessungen vorgenommen, wobei auch Vergleichsmessungen mit anderen, vergleichbaren Lösern, u.a. aus Intlab [130], durchgeführt werden.

Als weiterer großer Abschnitt in diesem Kapitel folgt eine Erläuterung von Lösern für dicht besetzte, parameterabhängige Gleichungssysteme. Analog zu den allgemeinen dicht besetzten Lösern wird dabei zunächst auf die theoretischen Aspekte eingegangen. Es folgt eine Beschreibung der Implementierung, bei der wiederum auf die Verwendung der in Kapitel 3 beschriebenen C-XSC Erweiterungen und Maßnahmen zur Parallelisierung eingegangen wird. Danach folgt die Erläuterung eines alternativen Lösungsansatzes, sowohl in theoretischer Hinsicht als auch in Form eines speziellen Lösers, welcher für bestimmte Klassen von Gleichungssystemen bessere Ergebnisse liefern kann. Im abschließenden Abschnitt werden auch hier wiederum detaillierte Tests und Zeitmessungen vorgenommen. Dabei werden auch Vergleiche der verschiedenen Löser angestellt.

Im gesamten Kapitel werden außerdem, wo sinnvoll, einige praktische Anwendungen der Löser angesprochen bzw. für die Tests verwendet. Dies gilt insbesondere für die parameterabhängigen Gleichungssysteme, für welche sich gerade im Ingenieursbereich diverse praktische Anwendungen finden, wie z.B. bei der mathematischen Modellierung von elektrischen Schaltungen oder dem Lösen strukturmechanischer Problemstellungen mittels der Finiten Elemente Methode.

4.1. Allgemeine dicht besetzte Gleichungssysteme

In diesem Abschnitt wird die verifizierte Lösung allgemeiner dicht besetzter (Intervall-) Gleichungssysteme behandelt, d.h. es werden Gleichungssysteme der Form

$$Ax = b \tag{4.1}$$

betrachtet, mit einer Systemmatrix $A \in \mathbb{K}^{n\times n}$, der rechten Seite $b \in \mathbb{K}^n$ und dem Lösungsvektor $x \in \mathbb{L}^n$, mit $\mathbb{K} \in \{\mathbb{R}, \mathbb{IR}, \mathbb{C}, \mathbb{IC}\}$ und $\mathbb{L} \in \{\mathbb{IR}, \mathbb{IC}\}$. Der berechnete Lösungsvektor soll eine (möglichst enge) Einschließung der tatsächlichen Lösungsmenge des Gleichungssystems sein (zur Gestalt der exakten Lösungsmenge siehe Abschnitt 2.1.3).

An die Struktur der Systemmatrix A werden dabei keinerlei nähere Anforderungen gestellt, d.h. das Problem soll allgemein für jede nicht singuläre (und nicht zu schlecht konditionierte) Systemmatrix gelöst werden können. Die Berechnung einer verifizierten Lösung bedeutet dabei auch eine Verifizierung der Regularität der Systemmatrix A (bzw. bei Intervallsystemen aller in A enthaltenen Punktmatrizen). Dies wird in Abschnitt 4.1.1 genauer erläutert und ein entsprechender Algorithmus wird besprochen.

Abschnitt 4.1.2 behandelt die Implementierung eines auf diesem Algorithmus aufbauenden Lösers mittels C-XSC. Eine spezielle, mittels MPI parallelisierte Version für Cluster wird in Abschnitt 4.1.3 vorgestellt. Tests, Zeitmessungen und Vergleiche mit anderen Softwarepaketen zur verifizierten Lösung allgemeiner dicht besetzter linearer (Intervall-) Gleichungssysteme folgen in Abschnitt 4.1.4.

Die in diesem Abschnitt vorgestellten Löser sind Weiterentwicklungen der vom Autor im Zuge seiner Master Thesis [146] erstellten Löser, welche in vielerlei Hinsicht erweitert und verbessert wurden, u.a. auch im Hinblick auf die in Kapitel 3 beschriebenen C-XSC Erweiterungen.

4.1.1. Theoretische Grundlagen

Für die näherungsweise Lösung linearer Gleichungssysteme stehen eine Vielzahl numerischer Verfahren zur Verfügung, darunter iterative Methoden wie das konjugierte Gradienten Verfahren sowie direkte Verfahren wie z.B. die LU-Zerlegung. Um nun lineare Punkt- und Intervallgleichungssysteme verifiziert zu lösen, ist eine mögliche Vorgehensweise, diese Verfahren direkt auf die Intervallarithmetik zu übertragen, indem sämtliche vorkommenden Grundoperationen durch entsprechende Intervalloperationen ersetzt werden. Wie in Abschnitt 2.1 am Beispiel des Intervallgaußverfahrens bereits erläutert, ist eine solche direkte Übertragung bestehender Verfahren aber in der Regel nicht zielführend, da es in den meisten Fällen entweder zu großen Überschätzungen oder gar zu einem kompletten Zusammenbruch der jeweiligen Methode kommt (sowohl beim Intervallgaußverfahren als auch beim konjugierte Gradienten Verfahren mittels Intervalloperationen tritt z.B. in der Regel sehr schnell eine Division durch ein Intervall, welches Null enthält, auf).

Aus diesem Grund sind speziell angepasste Algorithmen für die verifizierte Lösung linearer (Intervall-) Gleichungssysteme nötig. Für allgemeine dicht besetzte Systeme hat sich dabei eine Methode nach Rump [123], basierend auf dem Krawczyk-Operator [85],

bewährt. Diese wird auch in den hier beschriebenen Lösern verwendet und wird daher im Folgenden genauer erläutert. Die Erläuterungen folgen dabei im Wesentlichen Rump [132].

Zunächst wird nur das Punkt-Problem für Gleichung (4.1) mit vorgegebenen $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ und dem Lösungsvektor $x \in \mathbb{R}^n$ betrachtet. Das Lösen dieser Gleichung entspricht dem Problem, eine Nullstelle der Funktion

$$\mathrm{f}(x) := Ax - b \tag{4.2}$$

zu finden. Mittels des Newtonverfahrens lässt sich eine entsprechende Iterationsvorschrift

$$x^{(k+1)} := x^{(k)} - A^{-1}(Ax^{(k)} - b) \tag{4.3}$$

formulieren. Die exakte Inverse A^{-1} der Systemmatrix ist im Allgemeinen nicht bekannt. Stattdessen wird mit einer Näherungsinversen $R \approx A^{-1}$ gearbeitet, welche z.B. gleitkommamäßig mit der entsprechenden LAPACK-Routine berechnet werden kann. Weiterhin soll nun eine Einschließung $\boldsymbol{x} \in \mathbb{IR}^n$ des exakten Lösungsvektors betrachtet werden, d.h. es gilt $x \in \boldsymbol{x}$. Mit diesen Änderungen und einigen einfachen Umstellungen von (4.3) erhält man die Iterationsvorschrift

$$\boldsymbol{x}^{(k+1)} := Rb + (I - RA)\boldsymbol{x}^{(k)}. \tag{4.4}$$

Diese Iterationsvorschrift ist als Krawcyk-Operator [85] bekannt. Tritt nun während der Iteration der Fall auf, dass

$$\boldsymbol{x}^{(k+1)} \subseteq \mathrm{int}(\boldsymbol{x}^{(k)}) \tag{4.5}$$

gilt, so folgt aus Satz 2.6 (Brouwerscher Fixpunktsatz), das (4.2) wenigstens einen Fixpunkt $x^* \in \boldsymbol{x}^{(k+1)}$ besitzt, welcher eine Lösung des betrachteten linearen Gleichungssystems ist. Allerdings folgt aus dem Brouwerschen Fixpunktsatz nicht die Eindeutigkeit der Lösung. Wie folgender Satz (nach Rump [123], basierend auf Krawczyk [85] und Moore [98]) zeigt, lassen sich, falls Bedingung (4.5) erfüllt ist, allerdings noch wesentlich stärkere Schlußfolgerungen ziehen.

Satz 4.1. *Es seien $A, R \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ und $\boldsymbol{x} \in \mathbb{IR}^n$ gegeben. Gilt dann*

$$Rb + (I - RA)\boldsymbol{x} \subseteq \mathrm{int}(\boldsymbol{x}), \tag{4.6}$$

so sind die Matrizen A und R regulär und es gilt $A^{-1}b \in Rb + (I - RA)\boldsymbol{x}$.

Beweis: Mit $z := Rb$ und $C := (I - RA)$ lässt sich (4.6) umschreiben zu

$$z + C\boldsymbol{x} = C \cdot \mathrm{mid}\,(\boldsymbol{x}) + [z - |C| \cdot \mathrm{rad}\,(\boldsymbol{x}), z + |C| \cdot \mathrm{rad}\,(\boldsymbol{x})] \subseteq \mathrm{int}(\boldsymbol{x}).$$

Daraus folgt, dass

$$|C| \cdot \mathrm{rad}\,(\boldsymbol{x}) < \mathrm{rad}\,(\boldsymbol{x}). \tag{4.7}$$

Per definitionem ist $\mathrm{rad}\,(\boldsymbol{x})$ ein nicht negativer reeller Vektor und $|C|$ besitzt nur nichtnegative Einträge. Nach Perron und Frobenius folgt daher, dass $\mathrm{rad}\,(\boldsymbol{x})$ der Eigenvektor

zum betragsgrößten Eigenwert von $|C|$ ist, womit nach (4.7) für den Spektralradius von C folgt:

$$\rho(C) = \rho(I - RA) \leq \rho(|C|) < 1.$$

Aus $\rho(I - RA) < 1$ wiederum folgt, dass A und R nicht singulär sind, denn sonst wäre das Produkt RA singulär und hätte mindestens einen Eigenwert vom Betrag 0, so dass $I - RA$ mindestens einen Eigenwert vom Betrag 1 hätte. Weiterhin folgt, dass die Iteration $x^{(k+1)} = z + Cx^{(k)}$ für jeden Startvektor $x^{(0)} \in \mathbb{R}^n$ gegen

$$x = (I - C)^{(-1)}z = (I - I + RA)^{(-1)}Rb = (RA)^{(-1)}Rb = A^{(-1)}b$$

konvergiert. Ist $x^{(0)} \in \boldsymbol{x}$, so folgt aus der Annahme, dass auch $x^{(k)} \in \boldsymbol{x}$ für alle $k \geq 0$ gilt, womit $A^{-1}b \in Rb + (I - RA)\boldsymbol{x}$ folgt. □

Rump hat in seiner Dissertation [123] auf Basis von Satz 4.1 einen verifizierenden Algorithmus für lineare Gleichungssysteme entworfen, wobei er noch einige Erweiterungen vornahm. Neben der bereits angedeuteten Idee, eine Iterationsvorschrift entsprechend (4.4) zu verwenden und dann jeweils die Annahme von Satz 4.1 zu prüfen, gehören die Verwendung einer sogenannten Epsilon-Aufblähung und die Berechnung der Einschließung des Defektes einer Näherungslösung statt der direkten Berechnung einer Lösungseinschließung zu diesen Erweiterungen.

Die Epsilon-Aufblähung verbreitert in jedem Iterationsschritt das Intervall leicht, da es sonst zu Sonderfällen kommen kann, in denen $x^{(k+1)}$ nie im Inneren von $x^{(k)}$ eingeschlossen wird. Hierzu wird die Iterierte in jedem Schritt etwas verbreitert. Die Iteration wird also folgendermaßen abgewandelt, wobei blow die Funktion zur Berechnung der Epsilon-Aufblähung sei:

$$\boldsymbol{y} := \text{blow}(\boldsymbol{x}^{(k)}, \epsilon) \tag{4.8}$$

$$\boldsymbol{x}^{(k+1)} := Rb + (I - RA)\boldsymbol{y} \tag{4.9}$$

In einer Implementierung berechnet folgende Funktion für ein Gleitkommaintervall $\boldsymbol{a} \in \mathbb{IF}$ und ein beliebiges, kleines $\epsilon \in \mathbb{F}$ die Epsilon-Aufblähung:

$$\text{blow}(\boldsymbol{a}, \epsilon) := \begin{cases} (1 + \epsilon)\boldsymbol{a} - \epsilon\boldsymbol{a} & \text{, falls } \text{diam}([\boldsymbol{a}]) > 0 \\ [\text{pred}(\underline{a}), \text{succ}(\overline{a})] & \text{, falls } \text{diam}([\boldsymbol{a}]) = 0 \end{cases} \tag{4.10}$$

Die Funktionen pred und succ liefern dabei die nächstkleinere bzw. nächstgrößere Gleitkommazahl.

Die bedeutendste Modifizierung von Satz 4.1 durch Rump war es, statt der eigentlichen Lösung den Defekt einer vorher berechneten Näherungslösung $\tilde{x}$ verifiziert einzuschließen, was in der Regel zu deutlich besseren Ergebnissen (also engeren Einschließungen des tatsächlichen Ergebnisses) führt. Dies bedeutet, dass die Iterationsvorschrift abgeändert wird zu

$$\boldsymbol{x}^{(k+1)} = R(b - A\tilde{x}) + (I - RA)\boldsymbol{x}^{(k)}. \tag{4.11}$$

Wird dann für ein k die Bedingung $\hat{\boldsymbol{x}} := \boldsymbol{x}^{(k+1)} \subseteq \text{int}(\boldsymbol{x}^{(k)})$ erfüllt, so ist bewiesen dass $A^{-1}b \in \tilde{x} + \hat{\boldsymbol{x}}$. Die Näherungslösung lässt sich einfach über $\tilde{x} = Rb$ berechnen.

Die Qualität der am Ende berechneten Einschließung lässt sich steigern, in dem diese Näherungslösung durch eine mit höherer Genauigkeit berechnete Defektiteration noch verbessert wird.

Für Intervallsysteme $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ mit $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$, $\boldsymbol{b}, \boldsymbol{x} \in \mathbb{IR}^n$ kann das bisher beschriebene Vorgehen ebenfalls genutzt werden. Dazu wird $R = (\operatorname{mid}(\boldsymbol{A}))^{-1}$ und $\tilde{x} = R \cdot \operatorname{mid}(\boldsymbol{b})$ gesetzt, die Iterationsvorschrift lautet dann also

$$\boldsymbol{x}^{(k+1)} = R(\boldsymbol{b} - \boldsymbol{A}\tilde{x}) + (I - R\boldsymbol{A})\boldsymbol{x}^{(k)}, \tag{4.12}$$

wobei auch hier die Epsilon-Aufblähung während der Iteration genutzt wird.

Satz 4.2. *Für ein lineares Intervallgleichungssystem $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ mit $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$, $\boldsymbol{b} \in \mathbb{IR}^n$ führe die Iteration (4.12) zu einem Ergebnis (also einer Einschließung des Fehlers) $\boldsymbol{y} \in \mathbb{R}^n$. Dann sind alle Matrizen $A \in \boldsymbol{A}$ regulär und es gilt*

$$\Sigma(\boldsymbol{A}, \boldsymbol{b}) := \{x \in \mathbb{R}^n \mid Ax = b, A \in \boldsymbol{A}, b \in \boldsymbol{b}\} \subseteq \tilde{x} + \boldsymbol{y}.$$

Dies folgt direkt durch Anwendung von Satz 4.1 auf beliebige, feste $A \in \boldsymbol{A}, b \in \boldsymbol{b}$ und aus der Inklusionsmonotonie der Intervallrechnung. Es lässt sich zeigen [102], dass das beschriebene Verfahren für Intervallsysteme nur dann erfolgreich ist, wenn $\boldsymbol{A}$ stark regulär ist. Dies ist genau dann der Fall, wenn $\rho(|I - R\boldsymbol{A}|) < 1$ gilt [102].

Alle bisherigen Ausführungen lassen sich direkt ins Komplexe übertragen, so dass für das Eingangs dieses Kapitels formulierte Problem nun ein entsprechender Lösungsalgorithmus (Algorithmus 4.1) formuliert werden kann. Weiterhin ist anzumerken, dass der Algorithmus auch anwendbar ist, falls die rechte Seite $\boldsymbol{b}$ eine Matrix ist. Somit lässt sich der Algorithmus also z.B. nutzen, um eine verifizierte Einschließung der Inversen der Matrix A zu berechnen. Handelt es sich bei der Systemmatrix um eine Intervallmatrix $\boldsymbol{A}$, so kann Algorithmus 4.1 eine Einschließung für alle Inversen zu jeder Punktmatrix $A \in \boldsymbol{A}$ berechnen. Berechnet der Algorithmus in diesem Fall erfolgreich ein Lösung, so ist weiterhin bewiesen, dass alle Punktmatrizen $A \in \boldsymbol{A}$ regulär sind (hierzu genügt es schon, nur eine rechte Seite zu betrachten).

Die Qualität des berechneten Ergebnisses hängt von der Güte der Näherungslösung und der Näherungsinversen ab. Allgemein ist diese Methode bei Verwendung des *double*-Formats bis zu einer Kondition von etwa 10^{16} anwendbar. Für Punktsysteme entspricht die erreichte Genauigkeit in etwa $16 - \log(\operatorname{cond}(A))$ Ziffern. Durch eine Berechnung des Residuums mit höherer Präzision kann die Genauigkeit allerdings deutlich gesteigert werden. Der Aufwand wird dominiert von den beiden $\mathcal{O}(n^3)$ Operationen im Algorithmus, also der Bestimmung der Näherungsinversen und der Berechnung von $\boldsymbol{C}$. Beide Berechnungen können durch die Verwendung von BLAS- und LAPACK-Routinen sehr effektiv durchgeführt werden. Die Berechnung von $\boldsymbol{z}$ und die Defektiteration sollten mit höherer Genauigkeit durchgeführt werden. In C-XSC stehen hierzu inzwischen, gemäß Abschnitt 3.1, diverse Möglichkeiten zur Verfügung. Weitere Details zur Implementierung werden in Abschnitt 4.1.2 gegeben.

Der Algorithmus lässt sich so erweitern, dass er auch für schlechter konditionierte Systeme zu einer Lösung führt. Hierzu wird eine Inverse doppelter Länge verwendet. Die

Grundidee ist, dass die Matrix RA in der Regel deutlich besser konditioniert ist als die Matrix A. Da offensichtlich

$$A^{-1} = (RA)^{-1}R$$

gilt, kann also die Näherungsinverse in diesem Fall als SR berechnet werden, mit $S := (RA)^{-1}$. Dabei sollte insbesondere das Produkt SR in höherer Genauigkeit berechnet und das Ergebnis in Form einer Summe $R_1 + R_2$ abgespeichert werden, wobei R_1 und R_2 $n \times n$ Matrizen mit Einträgen vom Typ `double` sind. Auch hierfür sind die in Abschnitt 3.1 eingeführten Skalarproduktalgorithmen gut geeignet.

Eingabe : Systemmatrix A und rechte Seite b
Ausgabe : Intervalleinschließung der Lösung von $Ax = b$
Setze $Am := A$ für Punkt- bzw. $Am := \mathrm{mid}\,(A)$ für Intervallmatrizen
Setze $bm := b$ für Punkt- bzw. $bm := \mathrm{mid}\,(b)$ für Intervall-Rechte-Seite

Bestimme eine Näherungsinverse R von Am
Berechne Näherungslösung $\widetilde{x} := R \cdot bm$

`// Defektiteration zur Verbesserung von` $\widetilde{x}$
repeat
  `// Berechnung wird mit höherer Genauigkeit durchgeführt`
  $\widetilde{x} := \widetilde{x} + R(bm - Am\widetilde{x})$
until $\widetilde{x}$ *genau genug oder maximale Iterationszahl erreicht*

`// Berechne Einschließungen des Residuums und der Iterationsmatrix`
$Z := R \diamond (b - A\widetilde{x})$
$C := \diamond(I - RA)$

`// Intervalliteration`
$Y := Z$
repeat
  $Y_A := \mathrm{blow}(Y, \epsilon)$
  $Y := Z + C \cdot Y_A$
until $Y \subset int(Y_A)$ *oder maximale Iterationen erreicht*

`// Prüfung des Ergebnisses`
if $Y \subset int(Y_A)$ **then**
  Eindeutige Lösung existiert in $x \in \widetilde{x} + Y$
else
  Algorithmus fehlgeschlagen, A ist singulär oder schlecht konditioniert

Algorithmus 4.1: Algorithmus zur verifizierten Lösung linearer Gleichungssysteme

Wird diese neue Näherungsinverse $R := R_1 + R_2$ in Algorithmus 4.1 verwendet, so lassen sich auch Systeme mit Konditionszahlen größer als 10^{16} lösen. Durch die not-

wendige Verwendung von genaueren Skalarprodukalgorithmen und die Benutzung einer Inversen doppelter Länge steigt der Aufwand bei dieser Methode allerdings stark an. In der Praxis, falls die Kondition des Systems nicht vorab bekannt ist, sollte daher in der Regel zunächst Algorithmus 4.1 normal durchgeführt und nur bei dessen Scheitern (optional) die genauere Methode verwendet werden. Dabei ist anzumerken, dass, falls der einfache Algorithmus 4.1 scheitert, diese Erweiterung bei Intervallsystemen mit echten Intervalleinträgen in der Systemmatrix (keine Punktintervalle) normalerweise ebenfalls nicht zu einer Lösung führen wird, da die Systemmatrix neben der schlecht konditionierten Mittelpunktsmatrix dann in der Regel weitere, noch schlechter konditionierte oder sogar singuläre Punktmatrizen mit einschließt [129].

Bei der Lösung von Intervallgleichungssystemen kann die nach Algorithmus 4.1 berechnete Einschließung durch Überschätzungseffekte und eine große Konditionszahl des Problems recht breit ausfallen. Um die Qualität der bestimmten Einschließung abschätzen zu können, kann eine sogenannte Inneneinschließung berechnet werden (nach Neumaier [101], siehe auch Rump [125]).

Satz 4.3. *Gegeben sei ein lineares Intervallgleichungssystem mit Systemmatrix* $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ *und rechter Seite* $\boldsymbol{b} \in \mathbb{IR}^n$*, sowie eine Näherungslösung* $\tilde{x} \in \mathbb{R}^n$ *und ein* $\boldsymbol{x} \in \mathbb{IR}^n$*. Weiterhin sei*

$$\boldsymbol{z} := R(\boldsymbol{b} - \boldsymbol{A}\tilde{x})$$

und

$$\boldsymbol{D} := (I - R\boldsymbol{A})\boldsymbol{x}.$$

Gilt nun

$$\boldsymbol{z} + \boldsymbol{D} \subset \operatorname{int}(\boldsymbol{x}),$$

so sind R *und alle* $A \in \boldsymbol{A}$ *regulär und es gilt mit* $\boldsymbol{H} := \square(\Sigma(\boldsymbol{A}, \boldsymbol{b}))$

$$\tilde{x} + \underline{\boldsymbol{z}} + \underline{\boldsymbol{D}} \leq \underline{\boldsymbol{H}} \leq \tilde{x} + \underline{\boldsymbol{z}} + \overline{\boldsymbol{D}}$$

sowie

$$\tilde{x} + \overline{\boldsymbol{z}} + \underline{\boldsymbol{D}} \leq \overline{\boldsymbol{H}} \leq \tilde{x} + \overline{\boldsymbol{z}} + \overline{\boldsymbol{D}}.$$

Für einen Beweis siehe Rump [132]. Eine Inneneinschließung $\boldsymbol{y} \in \mathbb{IR}^n$ kann berechnet werden als

$$\boldsymbol{y} := [\tilde{x} + \underline{\boldsymbol{z}} + \overline{\boldsymbol{D}}, \tilde{x} + \overline{\boldsymbol{z}} + \underline{\boldsymbol{D}}].$$

Bei der Implementierung ist zu beachten, dass eine Oberschranke von $\underline{z}$ und eine Unterschranke von $\overline{z}$ benötigt wird. Weiterhin erhält man offensichtlich nur eine sinnvolle Inneneinschließung, falls $\operatorname{diam}(\boldsymbol{D}) \leq \operatorname{diam}(\boldsymbol{z})$.

Zum Abschluss dieses Abschnitts soll nun noch auf die verifizierte Lösung unter- und überbestimmter linearer (Intervall-) Gleichungssysteme eingegangen werden, d.h. (4.1) wird nun für A mit Dimension $m \times n$, x mit Dimension n und b mit Dimension m betrachtet. Es wird im Folgenden angenommen, dass A stets vollen Rang, also Rang n im überbestimmten Fall und Rang m im unterbestimmten Fall, besitzt.

Im überbestimmten Fall, also $m > n$, besitzt (4.1) in der Regel keine Lösung im üblichen Sinn. Stattdessen wird dasjenige x gesucht, welches das Quadrat der euklidischen

Norm des Residuums, also $||Ax - b||_2^2$, minimiert (Methode der kleinsten Fehlerquadrate). Da A vollen Rang besitzt, ist x in diesem Fall eindeutig bestimmt als die Lösung der Normalengleichung

$$A^H Ax = A^H b, \tag{4.13}$$

wobei A^H die Hermitesche von A bezeichnet. Dieses System direkt zu lösen hätte zum einen den Nachteil, dass $\text{cond}(A^H A) = \text{cond}(A)^2$, d.h. das System wäre schon für moderate Konditionszahlen evtl. nicht mehr lösbar, und zum anderen das Problem, dass $A^H A$ auf dem Rechner in der Regel nicht exakt bestimmt werden kann (es müsste also auch bei Punktsystemen mit Intervallmatrizen gerechnet werden).

Stattdessen wird mit $y := Ax - b$ auf ein $(m+n) \times (m+n)$ System übergegangen, welches sich mit Algorithmus 4.1 lösen lässt. Dazu wird (4.13) umgeschrieben zu

$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}. \tag{4.14}$$

Der Teil x der von Algorithmus 4.1 für dieses System berechneten Lösung ist eine Einschließung der gesuchten Lösung der Normalengleichung (4.13).

Für unterbestimmte Systeme, also den Fall $m < n$, kann ganz ähnlich verfahren werden. Ein solches System hat in der Regel unendlich viele Lösungen, daher wird der Lösungsvektor mit minimaler euklidischer Norm gesucht. Dieser kann mittels $y := A^H x$ aus der Lösung x des Systems $AA^H x = b$ bestimmt werden. Daraus ergibt sich wiederum ein $(m+n) \times (m+n)$ Ersatzsystem

$$\begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix}, \tag{4.15}$$

welches mit Algorithmus 4.1 gelöst werden kann. Der Teil y der berechneten Lösung ist dann der gesuchte Vektor.

Weitere Details zur verifizierten Lösung unter- und überbestimmter Systeme finden sich bei Rump [124]. Für allgemeine Erläuterungen zur Theorie solcher Systeme siehe z.B. Stoer [139].

4.1.2. Implementierung

Bei der Implementierung der in Abschnitt 4.1.1 beschriebenen Algorithmen steht insbesondere die Geschwindigkeit und im Zuge dessen die Ausnutzung moderner Mehrkernsysteme im Vordergrund. Bestehende Implementierungen für C-XSC, wie der Löser aus der C-XSC Toolbox [53] sowie der von Carlos Hölbig implementierte Löser [60], sind nicht nur in ihrem Funktionsumfang eingeschränkt (z.B. werden von beiden keine komplexen linearen Gleichungssysteme unterstützt) sondern sind auch, vor allem durch starke Nutzung des langen Akkumulators, äußerst langsam, gerade im Vergleich mit Intlab [130].

Um nun eine bessere Geschwindigkeit zu erreichen, werden in der Implementierung einige der in Kapitel 3 vorgestellten Erweiterungen von C-XSC verwendet. Zum einen

werden die neuen Skalarproduktalgorithmen eingesetzt, um an den Stellen, wo Skalarprodukte mit höherer Genauigkeit benötigt werden, durch die wählbare Präzision mehr Flexibilität zu bieten. Standardmäßig wird hierbei mit $K = 2$ gerechnet, also doppelter *double*-Präzision. Für den ersten Teil von Algorithmus 4.1 ist diese Genauigkeit in der Praxis ausreichend, denn falls eine höhere Genauigkeit nötig sein sollte, würde dies dafür sprechen, dass das Problem so schlecht konditioniert ist, dass der Algorithmus ohnehin nicht zum Erfolg führen würde. Allerdings kann eine höhere Genauigkeit gerade bei Punktsystemen, die gerade noch lösbar sind (für den ersten Teil des Algorithmus bedeutet dies also, dass die Kondition nahe bei 10^{16} liegt), die Genauigkeit des Ergebnisses um einige Stellen verbessern.

Weiterhin werden die neuen Möglichkeiten zur Verwendung von hochoptimierten BLAS und LAPACK Routinen genutzt. Diese werden für die beiden aufwändigsten Teile des Algorithmus, also die Berechnung von $\boldsymbol{C}$ sowie für die Berechnung der Näherungsinversen, eingesetzt. In beiden Fällen wird also einfache Gleitkommarechnung verwendet, und eine Einschließung bei der Berechnung von $\boldsymbol{C}$ wird über die Umstellung des Rundungsmodus bestimmt, gemäß dem in Abschnitt 3.2 beschriebenen Vorgehen.

Schließlich sind für die Implementierung des Lösers auch die in Abschnitt 3.5 beschriebenen Änderungen an der C-XSC Bibliothek zur Threadsicherheit von Bedeutung. Mittels OpenMP werden dabei die Teile der Implementierung, welche nicht mittels (normalerweise ohnehin schon mehrere Threads nutzender) BLAS- bzw. LAPACK-Routinen berechnet werden, für Mehrkern- bzw. Mehrprozessor-Systeme parallelisiert.

Bevor eine genauere Erläuterung einzelner Teile der Implementierung erfolgt, sollen die von dem hier implementierten Löser bereitgestellten Funktionalitäten kurz aufgezählt werden:

- Es werden alle C-XSC Grunddatentypen unterstützt, d.h. es können sowohl reelle als auch komplexe Punkt- oder Intervallsysteme gelöst werden.

- Die rechte Seite kann eine Matrix sein, d.h. das System kann durch einen Aufruf des Lösers für mehrere verschiedene rechte Seiten gelöst werden, wobei die aufwändigen Berechnungen zur Bestimmung der Näherungsinversen und der Matrix $\boldsymbol{C}$ unabhängig von der Zahl der rechten Seiten nur einmal durchgeführt werden. Unter anderem kann so auch eine Einschließung der Inversen der Systemmatrix berechnet werden. Zu diesem Zweck ist auch ein gesonderter Modus, der so genannte Matrix-Modus, wählbar, welcher bei sehr vielen rechten Seiten deutliche Geschwindigkeitsvorteile bietet. Näheres dazu folgt später in diesem Abschnitt.

- Optional kann eine Inneneinschließung gemäß Satz 4.3 berechnet werden, um die Qualität der berechneten Außeneinschließung abschätzen zu können.

- Über- und unterbestimmte Systeme werden, wie in Abschnitt 4.1.1 beschrieben, unterstützt.

- Es werden sowohl Algorithmus 4.1 in seiner ursprünglichen Form (in der Folge auch als Teil eins bezeichnet), als auch in Form der Nutzung einer Näherungsinversen

doppelter Länge (Teil zwei) unterstützt. Standardmäßig wird dabei zunächst Teil eins des Algorithmus verwendet und anschließend, falls keine verifizierte Einschließung der Lösung berechnet werden konnte, Teil zwei. Der Benutzer kann aber einstellen, ob stattdessen ausschließlich Teil eins oder ausschließlich Teil zwei des Algorithmus verwendet werden soll.

- Durch die Verwendung von BLAS und LAPACK für die in Punkto Laufzeit dominierenden Teile des Algorithmus ist die Implementierung (bei der Nutzung passender, optimierter BLAS- und LAPACK-Bibliotheken) deutlich schneller als vorherige C-XSC Implementierungen. Weiterhin werden Mehrkernsysteme automatisch durch moderne BLAS- und LAPACK-Bibliotheken ausgenutzt.

- Alle anderen Teile des Lösers werden mittels OpenMP ebenfalls parallelisiert. Da die meisten bedeutenden C++ Compiler mittlerweile OpenMP unterstützten, steht so für den Nutzer (optional) eine komplett parallelisierte Version zur Verfügung, welche die Vorteile moderner Mehrkernsysteme ausnutzen kann.

- Der Benutzer kann die Genauigkeit der Skalarproduktberechnungen für den gesamten Algorithmus wählen (außer bei der Berechnung von $\boldsymbol{C}$ und R, welche stets mittels normaler Gleitkommarechnung bestimmt werden). Bei passender Wahl der Genauigkeit können dabei sehr enge Einschließungen erreicht werden, welche mit den von den bisherigen C-XSC Lösern erzielten Resultaten vergleichbar sind und, je nach betrachtetem Gleichungssystem, deutlich besser ausfallen als die in Intlab erreichten Ergebnisse. Näheres dazu in Abschnitt 4.1.4.

Der neue Löser wurde direkt in die C-XSC Toolbox integriert, welche mit der Bibliothek mitgeliefert wird. Damit der Benutzer zur Kompilierzeit eines eigenen Programms die Möglichkeit hat, sowohl die Verwendung von BLAS und LAPACK (durch aktivieren der bereits erwähnten Schalter `-DCXSC_USE_BLAS` bzw. `-DCXSC_USE_LAPACK`) als auch von OpenMP (durch aktivieren der entsprechenden Option des verwendeten Compilers) zu aktivieren oder zu deaktivieren, liegt der gesamte Quellcode in einer Headerdatei (bzw. in einer `.inl` Datei, welche durch den Header inkludiert wird) vor. Bei einer Vorkompilierung mit dem Rest der C-XSC Bibliothek müssten die gewünschten Optionen sonst schon bei der Installation der C-XSC Bibliothek vorgewählt werden. Da die Zeit zur Kompilierung des Lösers auf modernen Systemen annehmbar ist, erscheint die zusätzliche Wartezeit gegenüber der gewonnen Flexibilität hinnehmbar.

Wie bereits in Abschnitt 4.1.1 bemerkt, ist Algorithmus 4.1 bis auf minimale Abweichungen für alle Grunddatentypen anwendbar. Es liegt also nahe, für die Implementierung Templates zu verwenden. Für einige kleinere Hilfsfunktionen ist dies problemlos möglich, bei der Erstellung einer Templateversion der Hauptroutine des Lösers muss allerdings einiges beachtet werden. Zunächst reicht es nicht, nur Templateparameter für die einzelnen Funktionsparameter anzulegen, da an vielen Stellen Datentypen an die Grunddatentypen des zu lösenden Systems angepasst werden müssen. Aus diesem Grund werden folgende Templateparameter für die Hauptfunktion (`LinSolveMain`) des Lösers verwendet:

- `TSysMat`: Typ der Systemmatrix $\boldsymbol{A}$, also einer der vier Matrixtypen.
- `TRhs`: Typ der rechten Seite $\boldsymbol{b}$, also ebenfalls einer der vier Matrixtypen (um mehrere rechte Seiten zu unterstützen).
- `TSolution`: Typ der berechneten Lösung $\boldsymbol{x}$, also entweder eine reelle oder komplexe Intervallmatrix.
- `TInverse`: Typ der Inversen $\boldsymbol{A}$, also entweder eine reelle oder komplexe Punktmatrix.
- `TC`: Typ der Matrix $\boldsymbol{C}$, also entweder eine reelle oder komplexe Intervallmatrix.
- `TMidMat`: Typ der zu berechnenden Mittelpunktsmatrizen, also entweder eine reelle oder komplexe Punktmatrix. Bei Punktsystemen entspricht `TMidMat` dem Typ `TSysMat`.
- `TMidVec`: Typ der zu berechnenden Mittelpunktsvektoren, analog zu den Mittelpunktsmatrizen.
- `TDot`: Typ der Variablen zur Berechnung von Punkt-Skalarprodukten, also entweder `dotprecision` oder `cdotprecision`.
- `TIDot`: Typ der Variablen zur Berechnung von Intervall-Skalarprodukten, also entweder `idotprecision` oder `cidotprecision`.
- `TVerivVec`: Typ des während der Intervalliteration zur abschließenden Verifizierung verwendeten reellen oder komplexen Intervallvektors.
- `TInterval`: Typ des einfachen Intervalldatentyps, also entweder `interval` oder `cinterval`.

Ein wesentlicher Unterschied bei der Lösung von Intervallsystemen gegenüber Punktsystemen ist, dass für Intervallsysteme die Mittelpunkte der Systemmatrix und der rechten Seite berechnet werden müssen. Eine Templateversion des Lösers muss also so implementiert sein, dass auf der einen Seite für Intervallsysteme die nötige Berechnung durchgeführt wird, auf der anderen Seite aber für Punktsysteme weder unnötige Berechnungen durchgeführt werden, noch Speicherplatz für eine nicht benötigte Mittelpunktsmatrix verschwendet wird.

Aus diesem Grund wird in der Implementierung mit Referenzen auf die Mittelpunktsmatrix gearbeitet, die durch eine überladene Funktion `midpoint` belegt werden, welche für Intervallsysteme eine Referenz auf die berechnete Mittelpunktsmatrix liefern, für Punktsysteme aber einfach eine Referenz auf die Matrix selbst. Im Quelltext stellt sich dies folgendermaßen dar:

```
TMidMat& Am = midpoint(A);
TMidMat& bm = midpoint(b);
```

Die Definition der verschiedenen Versionen der Funktion `midpoint` ist in Listing 4.1 gegeben.

```
1  inline rmatrix& midpoint(rmatrix &A) {
2    return A;
3  }
4
5  inline rmatrix& midpoint(imatrix &A) {
6    int m = Ub(A,ROW) - Lb(A,ROW) + 1;
7    int n = Ub(A,COL) - Lb(A,COL) + 1;
8
9    rmatrix* Am = new rmatrix(m,n);
10
11   #ifdef _OPENMP
12   #pragma omp parallel for default(shared)
13   #endif
14   for(int i=1 ; i<=m ; i++) {
15     for(int j=1 ; j<=n ; j++) {
16       (*Am)[i][j] = (Sup(A[i][j]) + Inf(A[i][j])) / 2.0;
17     }
18   }
19
20   return *Am;
21 }
22
23 inline cmatrix& midpoint(cmatrix &A) {
24   return A;
25 }
26
27 inline cmatrix& midpoint(cimatrix &A) {
28   int m = Ub(A,ROW) - Lb(A,ROW) + 1;
29   int n = Ub(A,COL) - Lb(A,COL) + 1;
30
31   cmatrix* Am = new cmatrix(m,n);
32
33   #ifdef _OPENMP
34   #pragma omp parallel for default(shared)
35   #endif
36   for(int i=1 ; i<=m ; i++) {
37     for(int j=1 ; j<=n ; j++) {
38       (*Am)[i][j] = complex( (SupRe(A[i][j]) + InfRe(A[i][j])) / 2.0,
39                              (SupIm(A[i][j]) + InfIm(A[i][j])) / 2.0);
40     }
41   }
42
43   return *Am;
44 }
```

Listing 4.1: Verschiedene Versionen der Funktion `midpoint` zur Berechnung der Mittelpunktsmatrix

Falls eine Mittelpunktsmatrix berechnet werden muss, so wird also dynamisch Speicher für diese Matrix angelegt und eine Referenz auf den neu angelegten Speicher zurückge-

geben, mit welcher dann in der Hauptfunktion gearbeitet wird. Bei Punktsystemen wird einfach direkt eine Referenz auf die übergebene Matrix zurückgeliefert. Die for-Schleife zur Berechnung der Mittelpunktsmatrix ist mittels OpenMP parallelisiert (falls OpenMP aktiviert wurde).

Falls eine Mittelpunktsmatrix angelegt wurde, so muss der reservierte Speicher nach Abschluss des Lösers wieder freigegeben werden. Da es sich um eine Templatefunktion handelt, der Quellcode also auch für Punktsysteme ausgeführt wird, muss dabei überprüft werden, ob überhaupt neuer Speicher für eine Mittelpunktsmatrix angelegt wurde. Dies geschieht mit folgendem Code:

```
if((TMidMat*)&A != &Am) delete &Am;
if((TMidMat*)&b != &bm) delete &bm;
```

Es wird also überprüft, ob die Referenzen der Mittelpunktsmatrizen die Original Systemmatrix bzw. die rechte Seite referenzieren. In diesem Fall handelt es sich offensichtlich um Punktmatrizen, d.h. es wurde kein Speicher für eine Mittelpunktsmatrix reserviert. Andernfalls wurde eine neue Mittelpunktsmatrix angelegt, d.h. der Speicher der von `Am` und `bm` referenzierten Variablen muss manuell wieder freigegeben werden.

Insgesamt sieht der Kopf der Hauptfunktion `LinSolveMain` aus wie in Listing 4.2 gezeigt.

```
1 template <typename TSysMat, typename TRhs, typename TSolution,
2           typename TInverse, typename TC, typename TMidMat,
3           typename TMidVec, typename TDot, typename TIDot,
4           typename TVerivVec, typename TInterval>
5 inline void LinSolveMain ( TSysMat &A, TRhs &b, TSolution& xx,
6                            int& Err, struct lssconfig cfg,
7                            bool inner, TSolution& ienc)
```

Listing 4.2: Kopf der Funktion `LinSolveMain`

Die Parameter haben dabei folgende Bedeutung:

- `A`: Die quadratische Systemmatrix. Über- und unterbestimmte Systeme werden vor Aufruf dieser Funktion entsprechend des in Abschnitt 4.1.1 erläuterten Vorgehens in ein quadratisches System umgeformt.
- `b`: Die rechte(n) Seite(n). Wird der Löser mit einem Vektor als rechter Seite aufgerufen, so wird dieser vor Aufruf der Hauptroutine automatisch in eine Matrix mit einer Spalte umgewandelt.
- `xx`: Zur Speicherung der berechneten Lösungseinschließung.
- `Err`: Speichert einen Fehlercode, welcher über die Funktion `LinSolveErrMsg` ausgewertet werden kann.
- `cfg`: Eine `struct`, welche diverse Konfigurationsoptionen für den Löser speichert. Diese wird im Folgenden noch näher erläutert.
- `inner`: Gibt an, ob eine Inneneinschließung berechnet werden soll.

- `ienc`: Falls eine Inneneinschließung berechnet werden soll, wird sie in dieser Variablen gespeichert.

Die Funktion `LinSolveMain` wird allerdings *nicht* vom Benutzer direkt aufgerufen. Stattdessen gibt es einfachere Einstiegsfunktionen für alle Systeme, welche nicht Template-basiert sind. Die verschiedenen vom Löser angebotenen Optionen werden über die in Listing 4.3 angegebene `struct` konfiguriert.

```
1  static const int
2    LSS_ONLY_PART_ONE = 0,
3    LSS_ONLY_PART_TWO = 1,
4    LSS_BOTH_PARTS    = 2;
5
6  struct lssconfig {
7    int   K;                //Dot product precision
8    bool  msg;              //Status message output?
9    int   lssparts;         //Solver stages to use
10   int   threads;          //Number of threads for OpenMP
11   int   maxIterResCorr;   //maximum number of iterations during residual
12                           //correction (not available for K=1)
13   int   maxIterVer;       //maximum number of iterations during the
14                           //verification step
15   bool  refinement;       //Perform an iterative refinement?
16   int   maxIterRef;       //maximum number of iterations during the
17                           //refinement step
18   real  epsVer;           //Epsilon for the verification step
19   real  epsRef;           //Epsilon for the refinement step
20   bool  matrixMode;       //Use only operators for computations. Faster
21                           //for multiple right hand sides, but might
22                           //lose some accuracy
23
24   lssconfig() : K(2),msg(false),lssparts(LSS_BOTH_PARTS),threads(-1),
25                 maxIterResCorr(10), maxIterVer(5), refinement(false),
26                 maxIterRef(5), epsVer(0.1), epsRef(1e-5),
27                 matrixMode(false) {}
28 };
```

Listing 4.3: `struct` zur Konfiguration des Lösers

Die Attribute der `struct` haben folgende Bedeutung:

- `K`: Die Präzision für die Skalarproduktberechnungen bei der Residuumsberechnung (in der Defektiteration und bei der Berechnung von $\boldsymbol{z}$).

- `msg`: Gibt an, ob Statusmeldungen des Lösers auf `cout` ausgegeben werden sollen oder nicht.

- `lssparts`: Legt fest, ob nur Teil eins (Wert `LSS_ONLY_PART_ONE`), nur Teil zwei (Wert `LSS_ONLY_PART_TWO`) oder beide Teile des Lösers (Wert `LSS_BOTH_PARTS`) verwendet werden sollen.

- `threads`: Anzahl an gewünschten Threads bei der Benutzung von OpenMP. Bei einem Wert ≤ 0 wird die aktuelle globale Vorgabe von OpenMP, welche z.B. über die Umgebungsvariable `OMP_NUM_THREADS` gesetzt werden kann, übernommen.
- `maxIterResCorr`: Maximale Anzahl an Iterationsschritten für die Defektiteration zur Verbesserung der berechneten Näherungslösung.
- `maxIterVer`: Maximale Anzahl an Iterationsschritten für die Verifikation.
- `refinement`: Gibt an, ob eine Nachiteration der Lösung durchgeführt werden soll. Hierbei wird im Wesentlichen die Iteration im Verifikationsschritt nach Finden einer Einschließung weitergeführt (Details folgen in Abschnitt 4.2).
- `maxIterRef`: Maximale Anzahl an Iterationsschritten für die Nachiteration.
- `epsVer`: Wert von Epsilon für die Epsilon-Aufblähung während der Verifikation.
- `epsRef`: Wert von Epsilon für das Abbruchkriterium der Nachiteration (der Durchmesser der Lösungseinschließung muss sich in jedem Schritt mindestens um einen Faktor Epsilon verkleinern, sonst wird die Iteration abgebrochen).
- `matrixMode`: Aktiviert den bereits angesprochenen Matrix-Modus, welcher für mehrere rechte Seiten und Präzision $K = 1$ deutlich schneller ist. Eine genauere Erläuterung folgt später in diesem Abschnitt.

Weiterhin verfügt die `struct` über einen Standardkonstruktor, welcher die Attribute auf die gebräuchlichsten Werte setzt. Dabei wird die Präzision $K = 2$, ohne Ausgabe von Statusmeldungen, mit beiden Teilen des Algorithmus, der globalen Vorgabe an Threads für OpenMP, 10 Iterationsschritten für die Defektiteration, 5 Iterationsschritten für die Verifikation und die iterative Verbesserung (welche standardmäßig ausgeschaltet ist), sowie einem Wert von 0.1 für das Epsilon der Epsilon-Aufblähung und einem Wert von 10^{-5} für das Epsilon der iterativen Verbesserung genutzt. Der Matrix-Modus ist standardmäßig deaktiviert. Gibt der Benutzer nicht explizit eine Konfiguration an, so wird die oben genannte verwendet. Zur Verdeutlichung zeigt Listing 4.4 als Beispiel die Köpfe einiger Einstiegsfunktionen des Lösers, welche vom Benutzer aufgerufen werden können.

```
1  //Ohne Inneneinschliessungen:
2
3  //! Entry function of real linear sytems solver
4  static inline void  lss(cxsc::rmatrix& A, cxsc::rmatrix& B,
5                          cxsc::imatrix& X, int& Err,
6                          struct lssconfig cfg = lssconfig());
7  //! Entry function of real linear sytems solver
8  static inline void  lss(cxsc::rmatrix& A, cxsc::rvector& b,
9                          cxsc::ivector& X, int& Err,
10                         struct lssconfig = lssconfig());
11 //! Entry function of real linear sytems solver
12 static inline void  lss(cxsc::rmatrix& A, cxsc::imatrix& B,
```

```
13                          cxsc::imatrix& X, int& Err,
14                          struct lssconfig = lssconfig());
15 //! Entry function of real linear sytems solver
16 static inline void  lss(cxsc::rmatrix& A, cxsc::ivector& b,
17                          cxsc::ivector& X, int& Err,
18                          struct lssconfig = lssconfig());
19
20 //Mit Inneneinschliessungen:
21
22 //! Entry function of real linear sytems solver
23 static inline void  lss(cxsc::rmatrix& A, cxsc::rmatrix& B,
24                          cxsc::imatrix& X, cxsc::imatrix& Y, int& Err,
25                          struct lssconfig cfg = lssconfig());
26 //! Entry function of real linear sytems solver
27 static inline void  lss(cxsc::rmatrix& A, cxsc::rvector& b,
28                          cxsc::ivector& X, cxsc::ivector& Y, int& Err,
29                          struct lssconfig = lssconfig());
30 //! Entry function of real linear sytems solver
31 static inline void  lss(cxsc::rmatrix& A, cxsc::imatrix& B,
32                          cxsc::imatrix& X, cxsc::imatrix& Y, int& Err,
33                          struct lssconfig = lssconfig());
34 //! Entry function of real linear sytems solver
35 static inline void  lss(cxsc::rmatrix& A, cxsc::ivector& b,
36                          cxsc::ivector& X, cxsc::ivector& Y, int& Err,
37                          struct lssconfig = lssconfig());
```

Listing 4.4: Beispiel für den Kopf einer Einstiegsfunktion des Lösers

Dies sind die Einstiegsfunktionen für den Löser für reelle Punktsysteme. Es werden jeweils verschiedene Versionen für matrix- oder vektorwertige rechte Seiten angeboten. Weiterhin gibt es auch Versionen, in denen die rechte Seite ein Intervallvektor bzw. eine Intervallmatrix ist. Die Standardkonfiguration wird als Defaultwert übergeben. Will der Benutzer keine eigene Konfiguration erstellen, so reicht es also aus, als Parameter die Systemmatrix, die rechte Seite, einen Vektor zur Speicherung der Lösung sowie die Variable für den Fehlercode zu übergeben. Alle weiteren Optionen werden dann automatisch auf die oben angegebenen Standardwerte gesetzt. Um eine Inneneinschließung zu berechnen, muss eine der Versionen mit einem zusätzlichen Parameter `Y` vom gleichen Typ wie die Lösung `X` aufgerufen werden. Sollten sich bei der Berechnung der Inneneinschließung Elemente mit leeren Mengen ergeben, so wird die entsprechende Komponente von `Y` auf `SignalingNaN` gesetzt. Neben diesen Einstiegsfunktionen gibt es analoge Funktionen für reelle Intervallsysteme (`ilss`), komplexe Punktsysteme (`clss`) sowie komplexe Intervallsysteme (`cilss`). Ein Beispielaufruf des Lösers folgt weiter unten in diesem Abschnitt.

Die Einstiegsfunktionen rufen zunächst eine Template-Funktion `SolverStart` auf. Diese prüft, ob es sich um ein quadratisches oder ein über- bzw unterbestimmtes System handelt. Ist das System nicht quadratisch, so wird es automatisch, wie in 4.1.1 beschrieben, in ein (größeres) quadratisches System umgewandelt. Erst dann wird die Hauptfunktion `LinSolveMain` des Lösers gestartet. Nach Ablauf der Hauptfunktion wird automatisch der passende Teil der Lösung des größeren Systems als Lösung des Originalsystems zurückgeliefert.

Die Fehlervariable speichert einen Fehlercode. Dabei sind folgende Fälle möglich:

- `NoError`: Es ist kein Fehler aufgetreten, eine verifizierte Lösung konnte berechnet werden.
- `NotSquare`: Das übergebene System war nicht quadratisch. Dieser Fehlerfall sollte nicht auftreten, da über- und unterbestimmte Systeme automatisch umgeformt werden. Er ist aus Kompatibilitätsgründen zu den Fehlermeldungen des originalen Toolbox-Lösers aber weiterhin vorhanden.
- `DimensionErr`: Die Dimensionen der Systemmatrix und der rechten Seite passen nicht zueinander.
- `InvFailed`: Die Berechnung der Näherungsinversen ist fehlgeschlagen. Das System ist entweder singulär oder schlecht konditioniert.
- `VerivFailed`: Die Verifizierung ist fehlgeschlagen. Das System ist entweder singulär oder schlecht konditioniert.

Über die Funktion `LinSolveErrMsg` lässt sich eine dem Fehlercode zugeordnete Fehlermeldung abfragen. In einer zukünftigen Version wäre eine Umsetzung des Fehlerhandlings in Form von Exceptions wünschenswert. Dies wurde hier noch nicht umgesetzt, da das Exception-Handling in der bevorstehenden C-XSC Version 3.0 komplett überarbeitet und vereinheitlicht werden soll, so dass erst dann eine sinnvolle Integrierung in die Fehlerbehandlung der Gesamtbibliothek möglich ist. Auch alle anderen Toolbox-Algorithmen verwenden zur Zeit noch Fehlercodes anstatt Exceptions.

Listing 4.5 zeigt nun ein kurzes Beipielprogramm für einen Aufruf des Lösers. Zunächst wird der Löser dabei mit den Standardoptionen aufgerufen, danach folgt ein Aufruf mit einer eigenen Konfiguration, wozu eine eigene Instanz der `struct lssconfig` angelegt wird. Weiterhin wird beim zweiten Durchlauf eine Inneneinschließung berechnet.

```
1  #include <fastlss.hpp>
2  #include <iostream>
3
4  using namespace cxsc;
5  using namespace std;
6
7  int main() {
8      int n = 1000;
9      rmatrix A(n,n);
10     rvector b(n);
11     ivector x(n), y(n);
12     int err;
13
14     for(int i=1 ; i<=n ; i++)
15         for(int j=1 ; j<=n ; j++)
16             A[i][j] = (i>j) ? i : j;
17     b = 1.0;
18
19     lss(A,b,x,err);
```

```
20      cout << LinSolveErrMsg(err) << endl;
21
22      struct lssconfig cfg;
23      cfg.K        = 3;
24      cfg.msg      = true;
25      cfg.lssparts = LSS_ONLY_PART_ONE;
26      cfg.threads  = 2;
27
28      lss(A,b,x,y,err,cfg);
29      cout << LinSolveErrMsg(err) << endl;
30
31      return 0;
32  }
```

Listing 4.5: Beispiel für den Aufruf des Lösers

Im Folgenden soll nun etwas genauer auf den Einsatz der in Kapitel 3 beschriebenen Erweiterungen von C-XSC eingegangen werden. Die Benutzung von BLAS und LAPACK lässt sich durch die in Abschnitt 3.2 beschriebenen Änderungen von C-XSC relativ einfach implementieren. Für die Berechnung der Näherungsinversen wird einfach die dort beschriebene Funktion zur Matrixinvertierung verwendet. Wird beim Kompilieren die Option `-DCXSC_USE_LAPACK` gesetzt, wird dabei automatisch die LAPACK Version genutzt. Andernfalls wird die einfache C++ Implementierung genutzt, welche bei Aktivierung von OpenMP auch mehrere Threads unterstützt.

Die BLAS Routinen werden bei Setzen des Schalters `-DCXSC_USE_BLAS` während der Kompilierung automatisch für alle Operatoraufrufe mit Präzision $K = 1$ verwendet. Wird BLAS nicht verwendet, so erfolgt die Berechnung mittels entsprechender `accumulate`-Aufrufe mit Präzision $K = 1$.

Um den Einsatz der Skalarproduktalgorithmen und die Nutzung von OpenMP näher zu erläutern, wird im Folgenden als Beispiel die Implementierung der Defektiteration, welche in Listing 4.6 angegeben wird, betrachtet.

```
1  x = R*bm[Col(s)]; k = 0;
2  do {
3       k++;
4       if(cfg.msg) std::cout << "␣" << k << ".␣iteration␣step" << std::endl;
5       y = x;
6       // Compute: d = #*(b-A*y)
7       #ifdef _OPENMP
8       #pragma omp parallel for firstprivate(dot) default(shared)
9       #endif
10      for (i = 1; i <= n; i++) {
11        dot = bm[i][s];
12        accumulate_approx(dot,-Am[i],y);
13        d[i] = rnd(dot);
14      }
15
16      // Compute: x = #*(y+R*d)
17      #ifdef _OPENMP
18      #pragma omp parallel for firstprivate(dot) default(shared)
19      #endif
```

```
20      for (i = 1; i <= n; i++) {
21        dot = y[i];
22        accumulate_approx(dot,R[i],d);
23        x[i] = rnd(dot);
24      }
25
26      CheckForZeros(y,x);
27  } while (!Accurate(y,x) && (k < cfg.maxIterResCorr));
```

Listing 4.6: Die Defektiteration als Beispiel für die Verwendung der veränderten `dotprecision`-Klassen und von OpenMP

Für die Skalarproduktberechnungen wird wie in alten Versionen des Lösers auch eine `dotprecision` Variable (`dot`) und entsprechende `accumulate`-Aufrufe verwendet. Durch die neuen Möglichkeiten zur Skalarproduktberechnung wird die Präzision K für die `dotprecision`-Variablen zu Beginn auf die vom Benutzer eingestellte Genauigkeit (standardmäßig also auf 2) gesetzt. Wie in Abschnitt 3.1 beschrieben werden alle Skalarprodukte, welche mit dieser `dotprecision`-Variable berechnet werden, dann in dieser Genauigkeit berechnet, das Ergebnis wird innerhalb eines langen Akkumulators in K-facher Länge gespeichert. Einfache Additionen werden immer maximal genau mittels des Akkumulators durchgeführt, wodurch die K-fache Länge der Skalarproduktergebnisse ausgenutzt wird und zu genaueren Resultaten führt. Die hier verwendete Funktion `accumulate_approx` bestimmt das Skalarprodukt, ohne gleichzeitig Fehlerschranken zu berechnen. Dies ist immer dann sinnvoll, wenn das zu berechnende Resultat, wie in diesem Beispiel, keine Einschließung sein und ein wenig Rechenzeit gespart werden soll.

Im Fall $K = 1$ führt die hier angegebene Art zur Berechnung des Residuums durch Auslöschungseffekte in der Regel nicht zu einer wesentlichen Verbesserung der Ergebnisgenauigkeit. In diesem Fall wird daher eine andere Methode nach Rump [127] verwendet. Diese basiert auf einer Methode von Dekker [38], eine Gleitkommazahl fehlerfrei in die Summe zweier Gleitkommazahlen, einen höherwertigen Kopf (*head*) und einen Schwanz (*tail*), zu zerlegen. Dieser Algorithmus wird auch bei den fehlerfreien Transformationen für den DotK-Algorithmus verwendet. Das Residuum $b - Ax$ kann dann wie in Listing 4.7 zu sehen berechnet werden.

```
1  template<typename TB>
2  inline rmatrix FastResidual(const rmatrix& B, const rmatrix& A,
3                              const rmatrix& X) {
4    static real factor = 68719476737;    //2^36 + 1
5    rmatrix T = factor * A;
6    rmatrix T2 = T - A;
7    T = T - T2;                          //erste 17 Bit von A
8    T2 = A - T;                          //hintere 35 Bit von A
9
10   rmatrix Y = factor * X;
11   rmatrix Y2 = Y - X;
12   Y = Y - Y2;                          //erste 17 Bit von X
13   Y2 = X - Y ;                         //hintere 35 Bit von X
14
15   return (B–T*Y)-(T*Y2+T2*X);
```

```
16 }
```

Listing 4.7: Schnelle Berechnung des Residuums nach Rump/Dekker

Die Systemmatrix A und die Näherungslösung X werden also fehlerfrei aufgeteilt in die Summe von jeweils zwei Matrizen, wobei die erste Matrix die ersten 17 Bit der Mantisse der Originalmatrix und die zweite Matrix die hinteren 35 Bit speichert. Durch diese Aufteilung kann das Produkt `T*Y` in Listing 4.7 sogar exakt berechnet werden, falls A und X keinen zu großen Exponentenbereich abdecken. Im Allgemeinen kann durch die angegebene Berechnungsart das Residuum in deutlich besserer Qualität als mit einfacher Gleitkommarechnung bestimmt werden.

Bei der Berechnung von $\boldsymbol{z}$ müssen entsprechende Anpassungen vorgenommen werden, um durch intervallmäßige Auswertung eine Einschließung des Residuums zu erhalten. Die Berechnung ist, insbesondere für mehrere rechte Seiten, vergleichsweise günstig. Allerdings ist die Ergebnisqualität der Residuumsberechnung in höherer Genauigkeit, so wie oben beschrieben, in den meisten Fällen signifikant besser. Aus diesem Grund ist auch die Einstellung $K = 2$ standardmäßig aktiviert, da der Mehraufwand für Systeme mit nur einer rechten Seite kaum ins Gewicht fällt. Bei vielen rechten Seiten sollte aber $K = 1$ und damit die schnelle Residuumsberechnung benutzt werden, da der dadurch in Kombination mit dem später beschriebenen Matrix-Modus erreichte Laufzeitgewinn sehr deutlich ausfällt.

Zur Integration von OpenMP können im Wesentlichen die in der Implementierung auftretenden for-Schleifen mittels des `parallel for` Konstrukts von OpenMP parallelisiert werden. Dabei ist darauf zu achten, dass die verwendeten `dotprecision`-Variablen mittels `firstprivate` als privat für die jeweilige Schleife deklariert werden. Dadurch werden die Kopien dieser Variablen für jeden Thread mit dem Wert der Variablen im Master-Thread initialisiert. Dies ist hier nötig, um die Übernahme der im `dotprecision`-Objekt gespeicherten Einstellung für die Präzision in allen Threads sicherzustellen. Ansonsten würde jeder Thread seine Kopie einfach mittels des Standardkonstruktors anlegen, wodurch jeweils die langsame Standardeinstellung $K = 0$ (maximale Genauigkeit mittels des langen Akkumulators) genutzt werden würde.

Die Effektivität der so erreichten Parallelisierung sowie Details zur Qualität der berechneten Ergebnisse bei Verwendung des DotK Algorithmus und der BLAS und LAPACK-Routinen im Vergleich zu den älteren C-XSC Lösern werden in Abschnitt 4.1.4 anhand verschiedener Tests genauer analysiert.

Zum Abschluss dieses Abschnitts soll nun noch einmal näher auf den Matrix-Modus eingegangen werden. Standardmäßig wird bei Systemen mit mehreren rechten Seiten einmalig die Näherungsinverse R sowie die Iterationsmatrix $\boldsymbol{C}$ berechnet und anschließend eine for-Schleife über alle rechten Seiten durchgeführt, welche die übrigen Berechnungen durchführt.

Bei sehr vielen rechten Seiten, also z.B. n rechten Seiten wie bei der Berechnung einer verifizierten Einschließung der Inversen der Systemmatrix, ist dieses Vorgehen sehr langsam. Zum einen wirkt sich dann die Verwendung einer höheren Präzision für die Residuumsiteration und die Berechnung von $\boldsymbol{z}$ deutlich stärker auf die Laufzeit aus, da diese Schritte ja nun für jede rechte Seite ausgeführt werden müssen. Zum anderen ist

aber selbst im Fall $K = 1$ das Vorgehen sehr langsam, da dann durch die Aufteilung der Berechnung auf die einzelnen rechten Seiten keine BLAS-Level-3-Operationen (also Matrix-Matrix-Operationen) genutzt werden können, welche deutlich effektiver sind.

Wird der Matrix-Modus aktiviert, so werden alle Berechnungen über die jeweiligen Operatoren durchgeführt. Im Fall $K = 1$ mit Verwendung der oben beschriebenen schnellen Residuumsberechnung können so viele Operationen mit den hochoptimierten BLAS-Level-3-Routinen durchgeführt werden. Auch für höhere Genauigkeiten ergibt sich zumindest ein leichter, in Kombination mit der in Abschnitt 3.5.1 beschriebenen Parallelisierung der Operatoren sogar ein deutlicher Geschwindigkeitsvorteil. Dadurch, dass der Akkumulator nicht für Zwischenberechnungen (z.B. bei der Berechnung der Differenz $b - Ax$) verwendet wird, kann die Genauigkeit des Ergebnisses aber leicht zurückgehen.

Insgesamt sollte der Matrix-Modus also insbesondere bei Systemen mit sehr vielen rechten Seiten aktiviert werden, in der Regel auch mit der Genauigkeit $K = 1$. Die dadurch gewonnene Geschwindigkeit dürfte in den meisten praktischen Fällen die unwesentlich niedrigere Genauigkeit ausgleichen. Details zu den Unterschieden in Geschwindigkeit und Ergebnisqualität folgen in Abschnitt 4.1.4.

4.1.3. Parallelisierung für Cluster

Die verifizierte Lösung großer dicht besetzter linearer Gleichungssysteme erfordert sowohl viel Speicherplatz als auch ein großes Maß an Rechenzeit. Der Rechenaufwand wird im Wesentlichen von der Berechnung der Näherungsinversen R und der Intervallmatrix C bestimmt (bei nur einer bzw. wenigen rechten Seiten). Für ein reelles Intervall-Gleichungssystem entsprechen diese Kosten in etwa vier reellen Matrix-Matrix-Multiplikationen, also $\mathcal{O}(4n^3)$.

Der Speicherplatzbedarf wird dominiert von den während der Berechnung benötigten Matrizen, also der Systemmatrix A, evtl. der Mittelpunktsmatrix $\operatorname{mid}(\boldsymbol{A})$, der Näherungsinversen R sowie der Matrix $\boldsymbol{C}$. Für die Lösung eines reellen Intervall-Gleichungssystems sind somit zwei reelle Punktmatrizen und zwei reelle Intervallmatrizen, welche den doppelten Speicherbedarf einer Punktmatrix aufweisen, zu speichern. Der Gesamtspeicherbedarf wird also dominiert von der Speicherung von sechs $n \times n$ Punktmatrizen, deren Elemente Gleitkommazahlen vom Typ `double` sind. Insgesamt belegen diese Matrizen also $6 \cdot n^2 \cdot 8$ Byte.

Für komplexe Systeme erhöht sich der Speicheraufwand um den Faktor zwei, der Rechenaufwand in etwa um einen Faktor vier. Weiterhin wird allgemein bei Verwendung von Teil zwei des Algorithmus deutlich mehr Rechenzeit benötigt, da hier die meisten Berechnungen mit größerer Präzision ausgeführt werden müssen (für genauere Vergleiche siehe Abschnitt 4.1.4). Aus diesem Grund stoßen viele aktuelle Systeme bei Dimensionen von $n = 10000$ oder wenig mehr bereits an ihre Grenzen. Um auch sehr große dicht besetzte Gleichungssysteme verifiziert lösen zu können, liegt daher eine eigene Implementierung für Systeme mit verteiltem Speicher (siehe Abschnitt 2.2.2) nahe.

Für einen möglichst optimalen Speicherbedarf und eine effiziente Parallelisierung ist dabei die Verteilung der Daten auf die einzelnen Knoten von großer Bedeutung. Die Daten sollten möglichst gleichmäßig auf alle Knoten verteilt werden, um den Speicherbedarf

pro Knoten gering zu halten, aber dabei auch so verteilt werden, dass die (zeitaufwändige) Kommunikation zwischen den Knoten möglichst minimiert wird, wobei jeder Knoten trotzdem so wenig Zeit wie möglich mit dem Warten auf benötigte Daten verbringen sollte.

Um auch auf Parallelrechnern mit verteiltem Speicher hochoptimierte Routinen zur Matrixinvertierung und zur Matrix-Matrix-Multiplikation nutzen zu können, wird in der Implementierung ScaLAPACK verwendet. Die hier vorgegebene zweidimensionale, blockzyklische Verteilung (siehe Abschnitt 2.4.3) weist genau die gewünschten Eigenschaften auf. Zur Kommunikation außerhalb der ScaLAPACK Routinen wird MPI verwendet. Wie in Abschnitt 3.4 beschrieben, steht ein entsprechendes Interface für alle C-XSC Datentypen sowie für den `vector`-Datentyp aus der STL zur Verfügung.

Die für ScaLAPACK benötigte logische Einteilung der Prozesse in ein Gitter wird so vorgenommen, dass in jedem Fall alle zur Verfügung stehenden Prozesse genutzt werden, das Prozessgitter dabei aber möglichst quadratisch ist, so dass Spalten- und Zeilenanzahl möglichst gleich oder nahezu gleich sind. Eine solche gleichmäßige Einteilung ist vorteilhaft für den Kommunikationsaufwand und damit die Laufzeit. Listing 4.8 zeigt die Funktion, welche für `procs` Prozesse die zu verwendende Zeilen- und Spaltenanzahl `p1` bzw. `p2` findet.

```
1  void determine_grid(int procs, int &p1, int &p2) {
2    if(procs == 2  ||  procs == 3) {
3      p1 = 1; p2 = procs;
4      return;
5    }
6
7    p1 = (int)floor(sqrt((double)procs));
8    p2 = p1;
9    int p1_tmp, p2_tmp = p2;
10   int maxp = 0;
11
12   for(p1_tmp = p1 ; p1_tmp>=1 ; p1_tmp--) {
13     while (p1_tmp*(p2_tmp+1)<=procs)
14       p2_tmp++;
15
16     if(p1_tmp * p2_tmp > maxp) {
17        p1 = p1_tmp;
18        p2 = p2_tmp;
19        maxp = p1 * p2;
20     }
21
22     if(p1 * p2 == procs)
23       break;
24     }
25 }
```

Listing 4.8: Funktion zur Bestimmung der Dimensionen des Prozessgitters

Anschließend werden die Daten entsprechend dem in Abschnitt 2.4.3 beschriebenen Schema verteilt. Die Aufteilung wird dabei nur für Matrizen verwendet (also A, R und $\boldsymbol{C}$). Sämtliche auftretenden Vektoren werden zur Verminderung des Kommunikationsauf-

wands von allen Prozessen komplett gespeichert. Sollte es mehrere rechte Seiten geben, so werden diese (ebenso wie die dazugehörigen Lösungen) zyklisch auf alle Prozesse verteilt. Eine Ausnahme bildet der bereits in Abschnitt 4.1.2 beschriebene Matrix-Modus, welcher auch für die parallele Version des Lösers zur Verfüng steht. Ist dieser aktiviert, so wird von sehr vielen rechten Seiten ausgegangen. Diese werden daher (ebenso wie die Lösung) als Matrix in zweidimensionaler, blockzyklischer Verteilung gespeichert.

Zur anfänglichen Verteilung der Systemmatrix A gibt es nun zwei Möglichkeiten:

1. Die Matrix liegt zu Beginn komplett in Prozess 0 vor und wird von diesem entsprechend der zweidimensionalen blockzyklischen Verteilung verteilt und anschließend gelöscht.

2. Der Löser wird mit einem Funktionspointer, welcher auf eine Funktion der Form (hier für eine reelle Punktmatrix)

   ```
   get_A(real& v, int i, int j);
   ```

   zeigt, aufgerufen. Diese Funktion speichert den Wert des Matrixelementes in Zeile `i` und Spalte `j` in der als Referenz übergebenen Variablen `v`. Die Funktion kann das zugehörige Element auf beliebige Weise bestimmen, z.B. über eine Formel (falls bekannt) oder einer Textdatei enthnehmen.

Die erste Methode eignet sich also nur für solche Matrizen, die in den Hauptspeicher eines Knotens passen. Die zweite Methode hingegen erfordert zu keinem Zeitpunkt, dass eine Matrix komplett in einem Knoten gespeichert wird und ermöglicht somit das Lösen wesentlich größerer Systeme. Ein ähnliches Vorgehen wird für die rechten Seiten verwendet.

Wie bei der in Abschnitt 4.1.2 beschriebenen Implementierung kommt auch hier Algorithmus 4.1 zum Einsatz. Allerdings wird jeder einzelne Schritt des Algorithmus nun parallel von allen Knoten ausgeführt. Dies bedeutet insbesondere, dass nicht nach dem Master-Slave Prinzip verfahren wird, sondern dass alle Knoten gleichberechtigt agieren. Die Berechnung der Näherungsinversen R sowie der Matrix $\boldsymbol{C}$ erfolgt dabei, analog zur Nutzung von BLAS und LAPACK beim seriellen Löser, mit ScaLAPACK bzw. PBLAS. Für die restlichen Berechnungen, also z.B. die Defektiteration, muss die Kommunikation hingegen manuell unter Berücksichtigung der gewählten Datenverteilung implementiert werden, da hier die genaueren Berechnungsalgorithmen für Skalarproduktausdrücke von C-XSC verwendet werden. Eine Ausnahme bildet auch hier wieder der Matrix-Modus, bei welchem alle Berechnungen mit entsprechenden ScaLAPACK bzw. PBLAS Routinen durchgeführt werden. Die eigentlichen Berechnungen entsprechen dabei dem Vorgehen beim seriellen Löser, d.h. auch hier wird die in Listing 4.7 angegebene Methode zur schnellen Residuumsberechnung genutzt, nur dass jetzt sämtliche Operationen entsprechend parallelisiert sind.

Durch die Aufteilung der einzelnen Skalarproduktberechnungen, welche im normalen Modus notwendig wird, ist zu beachten, dass bei der Kommunikation zwischen den beteiligten Knoten die Zwischenergebnisse in einem sinnvollen Format gesendet werden,

bei welchem keine Informationen verloren gehen. Da die Genauigkeit und somit auch die Länge der Einzelberechnungen hier, genau wie im seriellen Fall, wählbar ist, wird hierfür der lange Akkumulator verwendet. Unabhängig von der gewählten Genauigkeit können so die Zwischenergebnisse exakt weitergegeben werden, so dass es zu keinen Rundungsfehlern aufgrund der Kommunikation kommt. Der dadurch nötige zusätzliche Kommunikationsaufwand ist insgesamt vernachlässigbar, da diese Methode nur bei der Defektiteration, der Berechnung von $\boldsymbol{z}$, sowie der abschließenden Verifikation verwendet wird, welche (bei wenigen rechten Seiten) ohnehin nur geringen Einfluss auf den Gesamtaufwand haben.

Für die Berechnungen wird für jede Zeile und jede Spalte des Prozessgitters ein sogenannter MPI-Kommunikator angelegt. Dieser kann genutzt werden, um Kommunikationsoperationen mit MPI auf Teilmengen aller verfügbaren Knoten einzuschränken. So ist es also z.B. möglich, für jede Zeile und jede Spalte im Prozessgitter getrennt einen Broadcast durchzuführen, der die jeweiligen Daten ausschließlich an die der betreffenden Zeile bzw. Spalte zugeordneten Knoten sendet.

Vektoren werden grundsätzlich von allen Knoten komplett gespeichert, um Kommunikationsaufwand zu vermeiden. Bei der Berechnung eines Matrix-Vektor-Produktes muss also nur die Verteilung der Matrix gemäß dem ScaLAPACK Schema berücksichtigt werden. Mit den bisherigen Überlegungen ergibt sich ein allgemeiner Algorithmus (Algorithmus 4.2) für die Berechnung der im Löser auftretenden parallelen Matrix-Vektor-Produkte mit beliebiger Präzision.

Eingabe : Verteilte $n \times n$ Matrix A, Vektor b der Dimension n
Ausgabe : Das Produkt $x = Ab$
Lege Vektoren *mydot* und *dot* der Länge n von `dotprecision`-Objekten an
for *i=1:n* **do**
  for *j=1:n* **do**
    if A_{ij} *wird in diesem Knoten gespeichert* **then**
      Addiere $A_{ij} \cdot b_j$ zu $mydot_i$

Broadcast von *mydot* innerhalb der eigenen Zeile im Prozessgitter
Addiere alle *mydot* Vektoren in *dot* auf
Berechne alle x_i für die Zeilen i, an denen dieser Prozess beteiligt ist

Broadcast von x innerhalb der eigenen Spalte im Prozessgitter
Füge einzelne Teile von x zum Endergebnis zusammen

Algorithmus 4.2: Algorithmus für mit MPI parallelisierte Matrix-Vektor-Produkte

Nach Ablauf dieses Algorithmus liegt der Ergebnisvektor in allen Knoten komplett vor und kann für weitere Berechnungen verwendet werden. Durch die Speicherung der Zwischenergebnisse in Akkumulatoren ergeben sich dabei keine Rundungsfehler durch die Kommunikation, das Ergebnis wird weiterhin verlässlich eingeschlossen.

Da in Teil zwei des Algorithmus auch die Matrix-Matrix-Produkte in höherer Genauigkeit berechnet werden müssen, ist hier ein anderes Vorgehen nötig, da nun auch diese Produkte direkt und nicht mehr mit Hilfe von PBLAS berechnet werden. Da ScaLAPACK aber weiterhin für die Berechnung der Näherungsinversen benötigt wird, wird nun ein Spezialfall der zweidimensionalen blockzyklischen Verteilung verwendet, bei dem ein Prozessgitter mit $nc = 1$ und $nr = P$ (P ist die Anzahl der Prozesse) gewählt wird. Dadurch werden von jedem Prozess immer komplette Zeilen der Matrix gespeichert.

Für ein Matrix-Matrix-Produkt $A \cdot B$ in höherer Genauigkeit wird nun wie folgt vorgegangen: Die Matrix B wird in vertikalen $n \times nb$ (nb ist die ScaLAPACK Blockgröße, siehe Abschnitt 2.4.3) Blöcken betrachtet, von welchen jeder Prozess zunächst nur einen Teil speichert. Um komplette Skalarprodukte berechnen zu können, werden die eigenen Blöcke von jedem Prozess per Broadcast an alle anderen verschickt, so dass jeder Prozess die kompletten Skalarprodukte für die von ihm zu speichernden Zeilen der Ergebnismatrix selbst berechnen kann. Listing 4.9 zeigt den entsprechenden Quellcode. Die Arrays `ind1` und `ind2` enthalten dabei die Start- und Endindizes der von den verschiedenen Prozessen gespeicherten Matrixblöcke.

```
1  dotprecision dot(K,np);
2  rmatrix tmp;
3
4  for(int p=0 ; p<procs ; p++) {
5    if(ind1[p] == -1) continue;
6    tmp = rmatrix(1,n,ind1[p], ind2[p]);
7    for(int i=0 ; i<procs ; i++) {
8      if(ind1[i] == -1) continue;
9      if(mypid == i)
10       tmp(ind1[i],ind2[i],ind1[p],ind2[p]) = B(ind1[i],ind2[i],
11                                               ind1[p],ind2[p]);
12     MPI_Bcast(tmp, ind1[i],ind2[i],ind1[p],ind2[p], i, MPI_COMM_WORLD);
13   }
14
15   for(int i=ind1[mypid] ; i<=ind2[mypid] ; i++) {
16     for(int j=Lb(tmp,2) ; j<=Ub(tmp,2) ; j++) {
17       dot = (real)0.0;
18       accumulate(dot,A[Row(i)], tmp[Col(j)]);
19       R[i][j] = rnd(dot);
20     }
21   }
22 }
```

Listing 4.9: Parallele Multiplikation $R = A * B$ mit reellen Matrizen

Durch dieses Vorgehen wird zwar die Berechnung der Näherungsinversen etwas langsamer, ebenso fällt etwas Kommunikationsaufwand bei der Verteilung der Spalten von B sowie der anfänglichen Umverteilung an. Allerdings ist es so nicht nötig, Zwischenergebnisse exakt in Form von Akkumulatoren zu verschicken, was bei einem Matrix-Matrix-Produkt wesentlich größeren Aufwand verursachen würde als bei den Matrix-Vektor-Produkten aus Teil eins des Lösers.

Auch in der MPI-Version wird der Löser intern (also für den Benutzer nicht sichtbar)

mit Hilfe von Templates implementiert. Das Vorgehen entspricht dabei weitgehend der bereits in Abschnitt 4.1.2 beschriebenen Template-Version des seriellen Lösers.

Zum Aufruf des Lösers stehen wieder diverse Versionen für die vier Grunddatentypen zur Verfügung (`plss`, `pilss`, `pclss`, `pcilss`). Für jede dieser Versionen gibt es wiederum zwei verschiedene Aufrufmöglichkeiten: Einmal mit einer zunächst komplett in Prozess 0 gespeicherten Systemmatrix und rechter Seite und einmal mit einem Funktionszeiger, welcher die einzelnen Elemente der Systemmatrix und der rechten Seite direkt zurück liefert. Im Folgenden werden diese beiden Möglichkeiten jeweils beispielhaft für ein reelles Punktsystem betrachtet. Die Versionen für die anderen Datentypen verhalten sich analog.

Der Aufruf mit einer zunächst in Prozess 0 gespeicherten Systemmatrix hat folgendes Format:

```
void plss(cxsc::rmatrix& A, cxsc::rmatrix& B, cxsc::imatrix& X, int m,
          int n, int procs, int mypid, int& errc, std::ofstream& out,
          struct plssconfig cfg);
```

Die Parameter haben dabei folgende Bedeutung:

- **A**: Die Systemmatrix A, welche zunächst komplett von Prozess 0 gespeichert und von diesem an die anderen Prozesse verteilt wird.
- **B**: Die rechte(n) Seite(n) des Gleichungssystems. Bei mehreren rechten Seiten werden diese im Normalfall spaltenzyklisch auf die Prozesse verteilt. Bei Aktivierung des Matrix-Modus hingegen wird auch `B` in zweidimensionaler blockzyklischer Verteilung verteilt.
- **X**: Die verifizierte Einschließung der Lösung. Die Verteilung entspricht der rechten Seite `B`. In dieser Version des Lösers steht die komplette Lösung `X` am Ende in Prozess 0 zur Verfügung.
- **m**: Die Anzahl der Zeilen der Systemmatrix.
- **n**: Die Anzahl der Spalten der Systemmatrix.
- **procs**: Die Gesamtzahl der zur Berechnung verwendeten Prozesse.
- **mypid**: Die eigene Prozess-ID des jeweils aufrufenden Prozesses.
- **errc**: Ein Fehlercode analog zum seriellen Löser.
- **out**: Ein Dateistrom, auf welchen Statusmeldungen und das Ergebnis geschrieben werden.
- **cfg**: Eine `struct`, über welche weitere Konfigurationsmöglichkeiten gegeben werden.

Die `struct` zur Konfiguration ist ähnlich aufgebaut wie beim seriellen Löser und stellt die gleichen Attribute zur Konfiguration bereit. Einziger Unterschied ist das zusätzliche Attribut `nb`, welches die Blockgröße für ScaLAPACK angibt. Standardmäßig wird hier der Wert 256 verwendet. Weiterhin ist zu beachten, dass die gewählte Thread-Anzahl nur Einfluss auf die in ScaLAPACK bzw. PBLAS verwendeten Threads nimmt, und dies auch nur, falls eine mit OpenMP parallelisierte BLAS-Bibliothek verwendet wird.

Der Aufruf für die Benutzung der Variante mit einem Funktionszeiger zur Bestimmung der Systemmatrix hat das Format:

```
void plss(get_Element_of_rmatrix A, get_Element_of_rmatrix B,
          cxsc::imatrix& X, int m, int n, int rhs, int procs, int mypid,
          int& errc, std::ofstream& out, struct plssconfig cfg);
```

Der einzige Unterschied zur vorherigen Version ist, dass sowohl A als auch B über einen entsprechenden Funktionszeiger bestimmt werden und beide somit zu keinem Zeitpunkt komplett in einem Prozess vorliegen müssen. Außerdem muss über den Parameter `rhs` die Zahl der rechten Seiten explizit angegeben werden, welche in der vorherigen mit dem Parameter `B` automatisch übergeben wurde.

Wie beim seriellen Löser kann bei beiden Versionen eine Inneneinschließung berechnet werden, indem ein zusätzlicher Parameter `Y` vom gleichen Typ wie die Lösung `X` übergeben wird. Die Verteilung von `Y` nach Ablauf des Lösers entspricht der von `X`. Die Matrizen `X` und `Y` bleiben auch nach Ablauf des Lösers auf die einzelnen Prozesse verteilt.

4.1.4. Tests und Zeitmessungen

In diesem Abschnitt sollen nun diverse Tests mit den bisher beschriebenen Lösern durchgeführt werden. Dabei werden sowohl Geschwindigkeit, Effizienz der Parallelisierung (sowohl durch die Shared-Memory-Parallelisierung beim seriellen Löser als auch durch die MPI-Parallelisierung der Cluster-Version) sowie die Qualität der berechneten Einschließungen betrachtet. Der nicht-MPI-Löser wird im Folgenden der Einfacheit halber, als Abgrenzung zur Cluster-Version, auch als serieller Löser bezeichnet, obwohl er über eine Shared-Memory-Parallelisierung verfügt.

Die Tests in diesem Abschnitt werden auf einem System mit zwei Intel Xeon E5520 Prozessoren mit jeweils 4 Kernen und einer Taktrate von 2.26 GHz durchgeführt. Das System verfügt über 24 GB DDR3 Hauptspeicher, welcher mit 1066 MHz getaktet wird. Als Betriebssystem kommt die 64 Bit Version von OpenSUSE 11.4 zum Einsatz. Als C++ Compiler wird der Intel Compiler in Version 12.1 verwendet, als BLAS und LAPACK Bibliothek die Intel MKL in Version 10.3.

Der parallele Löser wird in erster Linie auf einem einfachen Cluster von 24 Core2 Duo E6550 Systemen mit jeweils 2 Kernen, einer Taktrate von 2 GHz und 2 GB Hauptspeicher durchgeführt. Die Rechner sind in einem einfachen Gigabit-Lan vernetzt. Als Betriebssystem dient auch hier die 64 Bit Version von OpenSUSE 11.4. Einige Tests werden zusätzlich auch auf vollwertigen Großrechnern durchgeführt, deren genaue Konfiguration bei den jeweiligen Tests gesondert aufgeführt wird.

Um die erzielten Ergebnisse in Relation setzen zu können, werden zusätzlich Tests mit zwei weiteren Softwarepaketen durchgeführt. Zum einen wird die Matlab-Erweiterung Intlab [130] in Version 6 betrachtet. Für diese Tests wurde Matlab in der Version 7.12.0.635 (R2011a) verwendet. Auch Matlab nutzt die Intel MKL als BLAS und LAPACK Bibliothek, allerdings in Version 10.2. Einige Vergleichstests haben aber gezeigt, dass diese auf dem verwendeten Testsystem für die hier benötigten Operationen die gleiche Geschwindigkeit wie die für den C-XSC Löser verwendete MKL Version 10.3 aufwies. Einzig bei Verwendung von 8 Threads traten auf dem Testsystem leichte Vorteile für C-XSC auf, was aber vermutlich eher auf einen leichten Overhead durch die Matlab-Oberfläche selbst als auf die verwendete MKL-Version zurückzuführen ist.

Als zweites Softwarepaket für Vergleiche werden die von Mariana Kolberg im Zuge ihrer Doktorarbeit [77] erstellten C++ Löser herangezogen. Hard- und Software für diese Tests entsprechen obigen Angaben. Von diesem Löser gibt es Versionen sowohl für Intervalle in Infimum-Supremum-Darstellung als auch in Mittelpunkt-Radius-Darstellung. Damit eine gute Vergleichsmöglichkeit mit Intlab und dem C-XSC Löser gegeben ist, wird stets die Variante, welche Intervalle in Mittelpunkt-Radius Darstellung nutzt, verwendet. Diese Löser werden in der Folge der Einfacheit halber auch als „Brasilien-Löser“ bezeichnet. Neben den seriellen Versionen ist auch eine MPI-Version enthalten, die (neben einer älteren und langsamen C-XSC Implementierung für reelle Intervallgleichungssysteme von Grimmer [50]) einzige andere dem Autor bekannte Implementierung eines solchen Lösers, welche mit dem in dieser Arbeit vorgestellten parallelen Löser verglichen wird. Alle Löser von Kolberg unterstützen nur reelle Punkt- und Intervallsysteme.

Testergebnisse für den seriellen Löser

Zunächst werden Zufallsmatrizen unterschiedlicher Dimension betrachtet. Diese werden so erzeugt, dass ihre Elemente pseudozufällige Zahlen aus einer Standard Normalverteilung sind. Obwohl es sich um Zufallsmatrizen handelt, sind ihre Eigenschaften, u.a. auch die Kondition, aufgrund des zentralen Grenzwertsatzes der Wahrscheinlichkeitsrechnung recht genau vorhersagbar. Der Erwartungswert für die Kondition solcher Zufallsmatrizen in den hier betrachteten Dimensionen liegt in etwa zwischen 10^4 und 10^6. Die tatsächlich für die folgenden Tests generierten Zufallsmatrizen weisen (von der von Matlab berechneten Schätzung der Konditionszahl ausgehend) in der Tat Konditionszahlen in diesem Bereich auf. Für mehr Details zur Verwendung von Zufallsmatrizen als Testmatrizen und zu Testmatrizen allgemein siehe Higham [58].

Tests hinsichtlich Laufzeit und Ergebnisqualität mit diesen Zufallsmatrizen sind daher als Repräsentation des allgemeinen Falles geeignet, da ihre Konditionszahlen weder sehr niedrig noch sehr hoch sind. Die Laufzeit ändert sich für schlechter konditionierte Matrizen nur dadurch, dass möglicherweise mehr Schritte bei der Defektiteration sowie der abschließenden Verifikation nötig sind. Diese zusätzlichen Iterationsschritte fallen allerdings nur wenig ins Gewicht, da die Laufzeit von der Berechnung der Näherungsinversen und der Iterationsmatrix $\boldsymbol{C}$ dominiert wird. Die Ergebnisqualität für die hier getesteten Zufallsmatrizen entspricht der im Allgemeinen zu erwartenden Qualität. Genaue Tests für unterschiedliche Konditionszahlen folgen später in diesem Abschnitt.

Zunächst werden die Zeitmessungen für den C-XSC Löser mit Standardeinstellungen betrachtet (insbesondere also Präzision $K = 2$). Hierfür werden Zufallssysteme für alle Datentypen mit Dimension $n = 1000, \ldots, 5000$ unter Benutzung von $P = 1, 2, 4, 8$ Threads verwendet. Für Intervallsysteme werden die Elemente der Systemmatrix um einen Faktor 10^{-12} aufgebläht. Als rechte Seite wird jeweils der erste Einheitsvektor verwendet. Die Tabellen 4.1, 4.2, 4.3 und 4.4 zeigen die Ergebnisse.

Threads	$n = 1000$	$n = 2000$	$n = 3000$	$n = 4000$	$n = 5000$
1	0.831	5.88	19.03	43.68	84.35
2	0.454	3.12	10.03	22.95	43.57
4	0.258	1.68	5.32	12.16	22.86
8	0.174	1.06	3.19	7.15	13.26

Tabelle 4.1.: Zeitmessungen in Sekunden für reelle Zufallssysteme

Threads	$n = 1000$	$n = 2000$	$n = 3000$	$n = 4000$	$n = 5000$
1	1.07	7.71	24.83	57.61	110.9
2	0.61	4.16	13.25	30.27	57.7
4	0.34	2.30	7.19	16.17	30.7
8	0.26	1.50	4.38	9.73	18.5

Tabelle 4.2.: Zeitmessungen in Sekunden für reelle Intervall-Zufallssysteme

Threads	$n = 1000$	$n = 2000$	$n = 3000$	$n = 4000$	$n = 5000$
1	3.13	22.87	73.20	171.0	329.4
2	1.64	11.71	37.78	87.7	167.7
4	0.89	6.12	19.74	45.6	86.9
8	0.53	3.56	11.03	25.6	48.9

Tabelle 4.3.: Zeitmessungen in Sekunden für komplexe Zufallssysteme

Die angegebenen Zeiten belegen, dass durch Verwendung einer BLAS und LAPACK Bibliothek mit Multithreading-Unterstützung sowie durch die zusätzliche Nutzung von OpenMP in den anderen Teilen des Lösers eine effektive Parallelisierung erreicht wird. Bevor der Speed-Up genauer betrachtet wird, sollen die angegebenen Zeiten zunächst durch einen Vergleich mit Intlab und dem Brasilien-Löser in einen besser zu interpretierenden Kontext gesetzt werden. Dazu werden die Zeiten jeweils für einen und für acht

Threads	$n = 1000$	$n = 2000$	$n = 3000$	$n = 4000$	$n = 5000$
1	4.39	31.19	99.9	230.8	442.3
2	2.36	16.31	52.3	120.2	229.6
4	1.36	8.91	28.2	65.2	121.7
8	0.93	5.65	17.3	37.7	71.4

Tabelle 4.4.: Zeitmessungen in Sekunden für komplexe Intervall-Zufallssysteme

Threads für alle Datentypen verglichen, einmal für die Dimension $n = 1000$ (Tabelle 4.5) und einmal für die Dimension $n = 5000$ (Tabelle 4.6).

Löser	**real**	**interval**	**complex**	**cinterval**
C-XSC, $P = 1$	0.83	1.07	3.12	4.50
C-XSC, $P = 2$	0.45	0.61	1.64	2.36
C-XSC, $P = 4$	0.26	0.34	0.89	1.36
C-XSC, $P = 8$	0.17	0.26	0.59	0.95
Intlab, $P = 1$	0.75	0.99	3.20	3.39
Intlab, $P = 2$	0.42	0.56	1.92	1.93
Intlab, $P = 4$	0.28	0.35	1.34	1.35
Intlab, $P = 8$	0.21	0.27	1.09	0.98
Brasilien, $P = 1$	0.70	1.09	–	–
Brasilien, $P = 2$	0.38	0.67	–	–
Brasilien, $P = 4$	0.23	0.46	–	–
Brasilien, $P = 8$	0.16	0.37	–	–

Tabelle 4.5.: Zeitvergleich in Sekunden für Zufallssysteme mit Dimension $n = 1000$

Zunächst ist zu erkennen, dass der C-XSC Löser bei Verwendung von einem Thread etwas langsamer als Intlab ist. Dies erklärt sich durch die aufwändigeren Zwischenberechnungen mit höherer Präzision, welche C-XSC für die Defektiteration und die Residuumsberechnung verwendet. Für komplexe Intervalle ergibt sich sogar ein deutlicher Vorsprung zu Gunsten von Intlab. Dieser kommt dadurch zustande, dass Intlab für komplexe Intervalle Kreisscheibenarithmetik verwendet, während C-XSC in komplexer Rechteckarithmetik arbeitet. Da eine Konversion zwischen beiden Darstellungen für Zwischenrechnungen in der Regel zu sehr großen Überschätzungen führen würde (siehe auch Abschnitt 3.2.1), werden im C-XSC Löser Realteil und Imaginärteil getrennt in Mittelpunkt-Radius-Darstellung gespeichert. Dadurch erhöht sich insbesondere der Auf-

Löser	**real**	**interval**	**complex**	**cinterval**
C-XSC, $P = 1$	84.3	110.9	329.4	442.3
C-XSC, $P = 2$	43.6	57.7	167.8	229.6
C-XSC, $P = 4$	22.9	30.7	86.9	121.7
C-XSC, $P = 8$	13.3	18.5	49.0	71.4
Intlab, $P = 1$	82.0	108.8	334.0	364.6
Intlab, $P = 2$	43.2	56.9	179.5	193.7
Intlab, $P = 4$	23.5	30.7	100.4	108.3
Intlab, $P = 8$	15.2	19.8	66.9	70.6
Brasilien, $P = 1$	79.6	110.2	–	–
Brasilien, $P = 2$	41.8	59.4	–	–
Brasilien, $P = 4$	22.0	32.9	–	–
Brasilien, $P = 8$	13.3	20.9	–	–

Tabelle 4.6.: Zeitvergleich in Sekunden für Zufallssysteme mit Dimension $n = 5000$

wand bei der Berechnung der Iterationsmatrix $\boldsymbol{C}$ im C-XSC Löser um ein komplexes Matrix-Matrix-Produkt gegenüber dem Intlab-Löser, was dessen Zeitvorsprung erklärt.

Allerdings weist der C-XSC Löser deutliche Vorteile bei der Parallelisierung auf, welche unter anderem auf eine effektivere Parallelisierung der $\mathcal{O}(n^2)$ Teile des Lösers zurückzuführen ist. Wie schon beschrieben, scheint weiterhin bei Verwendung von 8 Threads durch den Verwaltungsaufwand für die Matlab-Oberfläche selbst ein geringer Zeitnachteil zu entstehen. Auffällig ist auch, dass bei den komplexen Systemen C-XSC deutlich stärker von einer Parallelisierung profitiert, so dass sich für komplexe Punktsysteme ein klarer Geschwindigkeitsvorteil ergibt und bei komplexen Intervallsystemen der Geschwindigkeitsnachteil durch die verwendete Rechteckarithmetik quasi aufgehoben wird.

Die Ursache für diesen enormen Unterschied liegt in der Matlab Routine zur Berechnung der Näherungsinversen. Diese ist (aus unbekannten Gründen) langsamer und weniger effizient parallelisiert als die entsprechende LAPACK-Routine. Der Unterschied liegt nicht an der von Matlab in den Tests verwendeten älteren MKL-Version, wie Vergleichstests mit älteren MKL-Versionen und C-XSC gezeigt haben, sondern scheint mit der internen Implementierung in Matlab zusammenzuhängen. Ähnliche Unterschiede ergaben sich auch bei Tests auf anderen Systemen, weshalb davon ausgegangen werden muss, dass es sich um einen grundsätzlichen Nachteil von Matlab (und damit Intlab) handelt.

Der Brasilien-Löser ist für Punktsysteme am schnellsten, für Intervallsysteme aber der langsamste Löser. Dies ist schwer nachzuvollziehen, da im Brasilien-Löser anscheinend auf eine Berechnung des Residuums in höherer Genauigkeit komplett verzichtet

wird. Dies lässt sich an der Breite der berechneten Einschließungen erkennen, auf welche im weiteren Verlauf dieses Abschnitts noch eingegangen wird. Der Verzicht auf eine genauere Residuumsberechnung erklärt aber die Vorteile bei Punktsystemen. Aufgrund der sehr breiten Einschließungen, gerade bei höheren Konditionszahlen, und der stark eingeschränkten Funktionalität (u.a. fehlende Unterstützung für komplexe Systeme) ist der Nutzen dieses Lösers allerdings beschränkt.

Zusammenfassend lässt sich zu den Laufzeiten sagen, dass C-XSC und Intlab insgesamt in etwa auf dem gleichen Niveau liegen, wobei Intlab im Allgemeinen für komplexe Intervallsysteme deutlich schneller, dafür bei Verwendung von Multithreading etwas (und im Komplexen sogar deutlich) langsamer ist. Der Brasilien-Löser kann eine etwas schnellere Alternative für reelle Punktsysteme sein, falls die Ergebnisgenauigkeit weniger wichtig ist.

Als nächstes soll nun zur Beurteilung der Effizienz der Parallelisierung der Speed-Up des C-XSC Lösers näher betrachtet werden. Die Ergebnisse für Dimension $n = 1000$ zeigt Abbildung 4.1, die Ergebnisse für Dimension $n = 5000$ zeigt Abbildung 4.2. Zu beachten ist hierbei, dass nur für $P = 1, 2, 4, 8$ Daten vorhanden sind

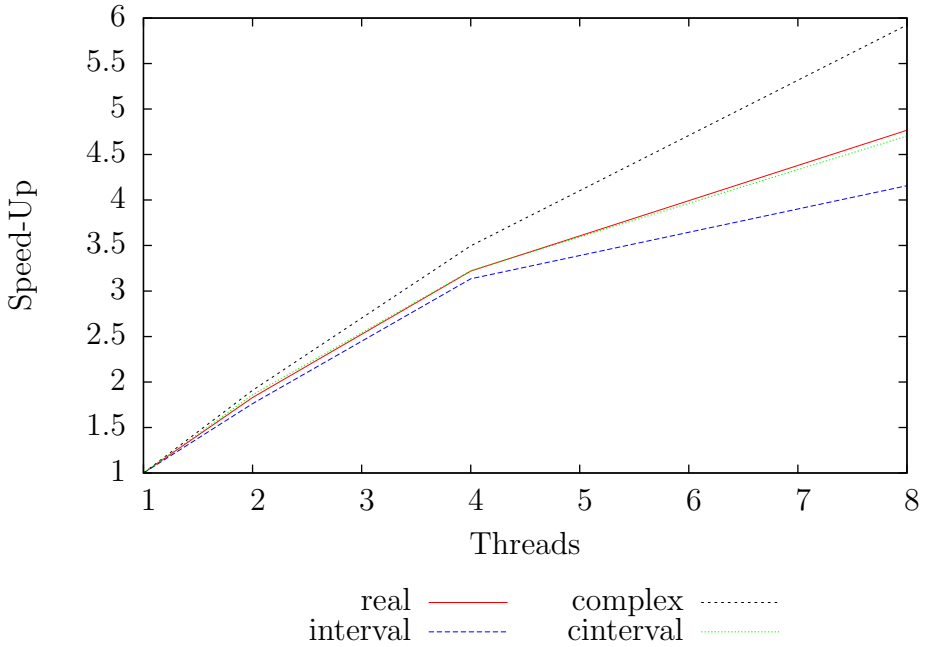


Abbildung 4.1.: Speed-Up des C-XSC Lösers für Dimension $n = 1000$

Die Abbildungen zeigen, dass im Allgemeinen ein sehr guter Speed-Up erreicht wird. Für $n = 1000$ steigt der Speed-Up für 8 Threads nicht mehr so stark an, da hier der Verwaltungsaufwand für die Parallelisierung gegenüber dem Aufwand für die eigentlichen Berechnungen stärker ins Gewicht fällt. Für $n = 5000$ ist der Aufwand deutlich höher, entsprechend zeigt sich auch ein besserer Speed-Up für 8 Threads. Allgemein ist der Speed-Up für Intervall-Systeme etwas schlechter, da hier vermehrt Umrechnungen zwi-

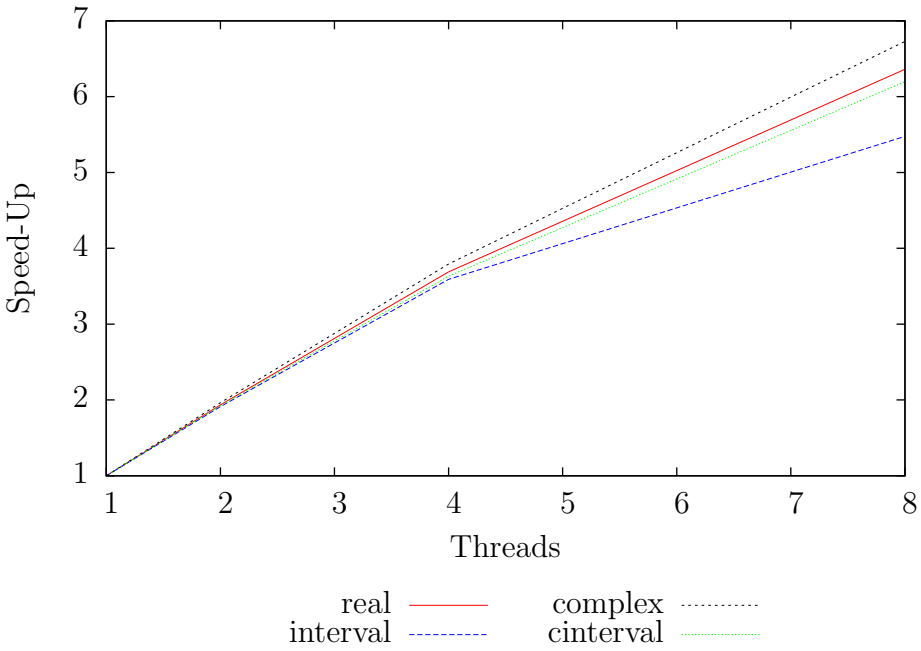


Abbildung 4.2.: Speed-Up des C-XSC Lösers für Dimension $n = 5000$

schen Mittelpunkt-Radius und Infimum-Supremum-Darstellung nötig werden. Insgesamt zeigt sich aber für alle Datentypen, dass die Parallelisierung des Lösers recht effizient ist und gut mit der Anzahl der Threads skaliert.

Zu beachten ist auch, dass die CPU des verwendeten Testsystems (wie die meisten modernen CPUs) eine Funktion namens Turbo Boost verwendet, welche, vereinfacht gesagt, die CPU automatisch übertaktet, falls nicht alle Kerne verwendet werden. Die Speed-Up Messungen in diesem und den folgenden Tests werden dadurch also insbesondere für Messungen mit 8 Threads negativ beeinflusst. Da aber die meisten modernen Mehrkern-Prozessoren diese bzw. ähnliche Funktionen verwenden, entsprechen die gemessenen Speed-Ups so eher den in der Praxis zu erwartenden Werten.

Als nächstes wird die Qualität der berechneten Einschließungen näher betrachtet. Zunächst werden dazu wieder die Ergebnisse für die bisher verwendeten Zufallsmatrizen mit Konditionszahlen zwischen etwa 10^4 und 10^6 betrachtet. Tabelle 4.7 zeigt den Vergleich der relativen Breite der berechneten Einschließung für Dimension $n = 1000$, Tabelle 4.8 für Dimension $n = 5000$.

Die Ergebnisqualität ist im Allgemeinen (auch für moderate Konditionszahlen wie in diesem Testfall) bei C-XSC durch die Verwendung der doppelten Präzision für die Residuumsberechnung signifikant besser als bei Intlab und beim Brasilien-Löser. Während C-XSC für reelle und komplexe Punktsysteme eine nahezu maximal genaue Einschließung liefert, verliert Intlab bereits 2-3 Stellen Genauigkeit. Die Kondition ist für $n = 5000$ jeweils etwas höher, wodurch auch die Genauigkeit bei Intlab gegenüber $n = 1000$ zurückgeht. Der Brasilien-Löser berechnet mit Abstand die breitesten Einschließungen, da offensichtlich das Residuum in einfacher Gleitkommarechnung bestimmt wird, wodurch

die relative Breite um mehrere Größenordnungen über dem von C-XSC und Intlab liegt.

Löser	**real**	**interval**	**complex**	**cinterval**
C-XSC, $P = 1$	$1.91 \cdot 10^{-16}$	$1.84 \cdot 10^{-7}$	$3.83 \cdot 10^{-16}$	$5.73 \cdot 10^{-7}$
C-XSC, $P = 8$	$1.91 \cdot 10^{-16}$	$1.84 \cdot 10^{-7}$	$3.83 \cdot 10^{-16}$	$5.73 \cdot 10^{-7}$
Intlab, $P = 1$	$2.67 \cdot 10^{-13}$	$1.84 \cdot 10^{-7}$	$2.84 \cdot 10^{-14}$	$4.55 \cdot 10^{-8}$
Intlab, $P = 8$	$3.67 \cdot 10^{-14}$	$1.84 \cdot 10^{-7}$	$2.96 \cdot 10^{-15}$	$4.55 \cdot 10^{-8}$
Brasilien, $P = 1$	$5.03 \cdot 10^{-7}$	$9.46 \cdot 10^{-6}$	–	–
Brasilien, $P = 8$	$5.03 \cdot 10^{-7}$	$9.46 \cdot 10^{-6}$	–	–

Tabelle 4.7.: Vergleich der relativen Breite der Einschließung für Zufallssysteme mit Dimension $n = 1000$

Löser	**real**	**interval**	**complex**	**cinterval**
C-XSC, $P = 1$	$1.91 \cdot 10^{-16}$	$2.62 \cdot 10^{-6}$	$3.83 \cdot 10^{-16}$	$3.94 \cdot 10^{-6}$
C-XSC, $P = 8$	$1.91 \cdot 10^{-16}$	$2.62 \cdot 10^{-6}$	$3.83 \cdot 10^{-16}$	$3.94 \cdot 10^{-6}$
Intlab, $P = 1$	$2.14 \cdot 10^{-12}$	$2.63 \cdot 10^{-6}$	$9.33 \cdot 10^{-13}$	$3.50 \cdot 10^{-7}$
Intlab, $P = 8$	$2.58 \cdot 10^{-13}$	$2.63 \cdot 10^{-6}$	$1.15 \cdot 10^{-13}$	$3.48 \cdot 10^{-7}$
Brasilien, $P = 1$	$4.26 \cdot 10^{-5}$	$3.19 \cdot 10^{-4}$	–	–
Brasilien, $P = 8$	$4.26 \cdot 10^{-5}$	$3.19 \cdot 10^{-4}$	–	–

Tabelle 4.8.: Vergleich der relativen Breite der Einschließung für Zufallssysteme mit Dimension $n = 5000$

Interessant ist auch, dass Intlab durch die Parallelisierung etwas an Genauigkeit gewinnen kann. Dieser Effekt ist oft zu beobachten, da bei Nutzung mehrerer Threads die auftretenden Skalarprodukte in kleinere Produkte aufgeteilt werden, welche häufig besser konditioniert sind. C-XSC profitiert hier nicht, da auch bei Verwendung mehrerer Threads durch die Art der Parallelisierung stets komplette Skalarprodukte berechnet werden. Außerdem erreicht C-XSC ohnehin schon einen nahezu optimale Einschließung, die sich nicht wesentlich verbessern lässt.

Für Intervallsysteme werden Rundungsfehler im Allgemeinen durch Überschätzungseffekte überlagert, weshalb sich hier keine großen Unterschiede ergeben. Beachtenswert ist aber zum einen, dass der Brasilien-Löser auch hier etwas breitere Einschließungen aufweist. Zum Anderen fällt auf, dass Intlab für komplexe Intervallsysteme etwas engere Einschließungen berechnet. Dieser Umstand ist wiederum der Verwendung der Kreisscheibenarithmetik in Intlab gegenüber der Rechteckarithmetik in C-XSC geschuldet, da die in C-XSC zum Testen verwendeten Rechteckintervalle Einschließungen der in Intlab

verwendeten Kreisscheiben sind (im umgekehrten Fall drehen sich die hier gezeigten Verhältnisse gerade um). Ein exakter Vergleich ist durch die verschiedenen Datenformate nicht möglich.

Im Folgenden soll nun die Güte der Einschließung für verschiedene Konditionszahlen getestet und verglichen werden. Dabei werden ausschließlich Punktsysteme betrachtet, da bei Intervallsystemen normalerweise die Überschätzung durch die Intervallarithmetik den Genauigkeitsverlust durch Rundungsfehler ohnehin überlagert. Weiterhin gelten für Intervallsysteme mit sehr engen Intervallen im Wesentlichen die gleichen Aussagen bezüglich der Ergebnisgenauigkeit wie für Punktsysteme.

Als Einstieg wird zunächst die berühmte Hilbert-Matrix betrachtet. Die Einträge der Hilbert-Matrix sind über die Formel

$$a_{ij} = \frac{1}{i+j-1}$$

definiert. Die Konditionszahl der Hilbert-Matrix steigt exponentiell mit der Dimension. Schon für sehr kleine Dimensionen ist die Matrix schlecht konditioniert, für die Dimension $n = 10$ z.B. weist sie bereits eine Kondition von ca. $1.6 \cdot 10^{13}$ auf [58].

In der Folge wird ein System mit der Hilbert-Matrix für die Dimensionen $n = 10$ betrachtet. Für $n = 10$ ist das System noch mit Teil eins des verwendeten Algorithmus lösbar. Mit dem zweiten Teil des Algorithmus, welcher eine Inverse doppelter Länge verwendet, ließe sich dieses System auch für größere Dimensionen (bis etwa $n = 20$) mit dem C-XSC Löser verifiziert lösen (Intlab und der Brasilien-Löser würden dann kein verifiziertes Ergebnis mehr liefern). Um Fehler in den Eingangswerten zu vermeiden, wird die Matrix mit dem kleinsten gemeinsamen Vielfachen der Nenner der Matrixeinträge multipliziert, so dass sie im Rechner exakt darstellbar ist. Die rechte Seite ist der erste Einheitsvektor, welcher ebenfalls mit dem kleinsten gemeinsamen Vielfachen multipliziert wird. Auf diese Weise ergeben sich mit C-XSC (für $K = 2$), Intlab und dem Brasilien-Löser für $n = 10$ die in Abbildung 4.3 gezeigten Ergebnisse.

Der C-XSC Löser liefert eine nahezu exakte Einschließung und verliert nur etwa eine Stelle Genauigkeit. Intlab liefert auch eine recht gute Einschließung, verliert aber ca. 6-7 Stellen. Beim Brasilien-Löser zeigt sich wiederum, dass keine höhere Präzision für die Residuumsberechnung verwendet wird, wodurch die Einschließungen sehr breit ausfallen. In diesem Beispiel liefert der Brasilien-Löser nur etwa eine Stelle. Da sich ein ähnliches Bild in allen Tests ergab, wird für die folgenden Ergebnisvergleiche für verschiedene Konditionszahlen der Brasilien-Löser nicht mehr betrachtet und es werden nur noch C-XSC und Intlab verglichen.

Um nun das Verhalten des C-XSC Lösers mit unterschiedlichen Präzisionen K sowie des Intlab Lösers für verschiedene Konditionszahlen zu testen, wird für die Konditionszahlen $c := 10^k, k = 1, \ldots 15$ mittels eines in [146] beschriebenen Verfahrens jeweils eine Zufallsmatrix der Dimension $n = 1000$ erzeugt, welche in etwa die jeweils gewünschte Kondition c aufweist. Danach wird für die C-XSC Löser jeweils mit $K = 1, 2, 3$ und den Intlab Löser das Punktsystem mit der jeweiligen Matrix und dem ersten Einheitsvektor als rechter Seite gelöst. Abbildung 4.4 zeigt die Entwicklung des relativen Fehlers für die jeweiligen Löser bei steigender Konditionszahl.

$[+9.999999999999943E+001,+1.000000000000006E+002]$
$[-4.950000000000002E+003,-4.949999999999998E+003]$
$[+7.919999999999999E+004,+7.920000000000001E+004]$
$[-6.006000000000001E+005,-6.005999999999999E+005]$
$[+2.522519999999999E+006,+2.522520000000001E+006]$
$[-6.306300000000001E+006,-6.306299999999999E+006]$
$[+9.609599999999998E+006,+9.609600000000002E+006]$
$[-8.751600000000002E+006,-8.751599999999998E+006]$
$[+4.375799999999999E+006,+4.375800000000001E+006]$
$[-9.237800000000001E+005,-9.237799999999999E+005]$

C-XSC

$[+9.999999938203096E+001,+1.000000006167175E+002]$
$[-4.950000003520177E+003,-4.949999996365704E+003]$
$[+7.919999994679834E+004,+7.920000005068271E+004]$
$[-6.006000004589672E+005,-6.005999995175278E+005]$
$[+2.522519997702649E+006,+2.522520002183151E+006]$
$[-6.306300005989145E+006,-6.306299993692330E+006]$
$[+9.609599989661772E+006,+9.609600009809865E+006]$
$[-8.751600009467393E+006,-8.751599990017198E+006]$
$[+4.375799994762696E+006,+4.375800004964544E+006]$
$[-9.237800010907662E+005,-9.237799988488659E+005]$

Intlab

$[+5.314354368008400E+001,+1.471595610281726E+002]$
$[-5.019238329456018E+003,-4.880691436440543E+003]$
$[+7.879507489751576E+004,+7.961505083698375E+004]$
$[-6.038764540004879E+005,-5.974020725684437E+005]$
$[+2.507413847895453E+006,+2.537981594392369E+006]$
$[-6.348691586250677E+006,-6.264833153030452E+006]$
$[+9.541574813926440E+006,+9.679073596751902E+006]$
$[-8.818700539335603E+006,-8.685838997052757E+006]$
$[+4.341251398777778E+006,+4.411025482224011E+006]$
$[-9.315472398733862E+005,-9.161558129804133E+005]$

Brasilien

Abbildung 4.3.: Vergleich der numerischen Ergebnisse der verschiedenen Löser für die Hilbert-Matrix mit $n = 10$

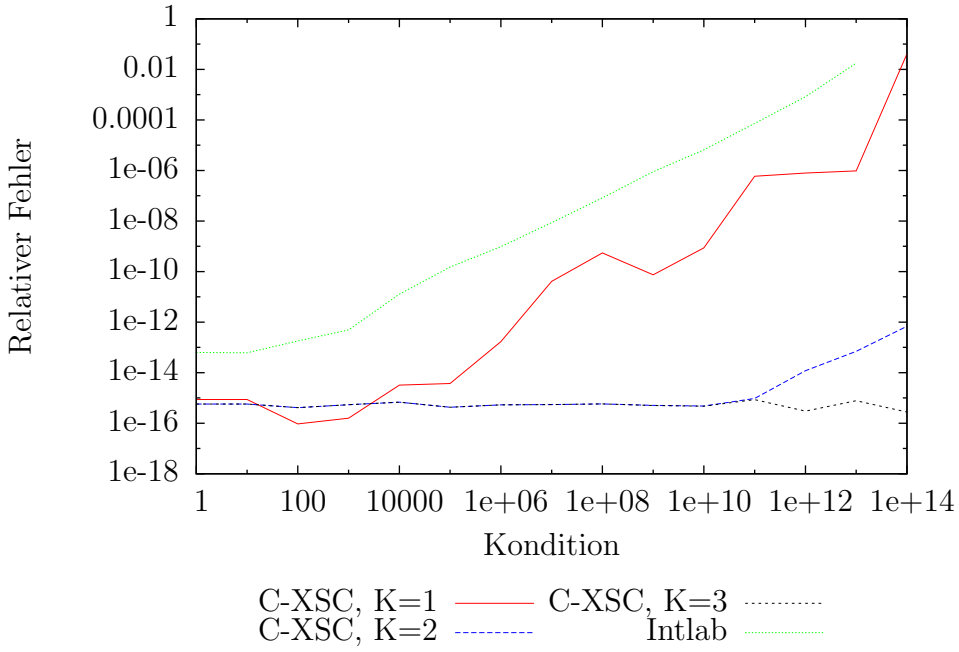


Abbildung 4.4.: Relativer Fehler der berechneten Einschließungen bei unterschiedlichen Konditionen

Die Ergebnisse entsprechen recht gut den Erwartungen, die sich aus den bisherigen Tests und der jeweils verwendeten Berechnungsmethode für das Residuum ergeben. Der relative Fehler steigt bei Intlab in etwa linear mit der Kondition des Systems an. Bemerkenswert bei den Intlab-Ergebnissen ist zum einen, dass auch für sehr niedrige Konditionszahlen bereits ein relativer Fehler in der Größenordnung 10^{-13} auftritt, zum anderen dass bereits für eine Kondition von etwa 10^{14} keine verifizierte Einschließung mehr bestimmt werden kann, obwohl C-XSC auch für dieses System noch eine verifizierte Lösung findet. Eventuell ist hier die von Matlab berechnete Näherungsinverse nicht von so guter Qualität wie die von C-XSC mittels LAPACK berechnete Näherungsinverse.

C-XSC zeigt für Präzision $K = 1$ insgesamt ein ähnliches Verhalten, allerdings gibt es zum einen deutlichere Schwankungen als bei Intlab, zum anderen liegt der relative Fehler stets deutlich unter dem von Intlab. Ursache hierfür ist vermutlich, dass C-XSC Zwischenergebnisse im Akkumulator speichert, wodurch Addition und Subtraktion exakt ausgeführt werden (auch wenn die auftretenden Skalarprodukte mit einfacher Genauigkeit berechnet werden). Dies führt trotz der gleichen Methode zur Berechnung des Residuums in diesem Fall zu leichten Vorteilen für C-XSC.

Für $K = 2$ bleibt die Einschließung der Lösung bis zu einer Kondition von etwa 10^{11} nahezu exakt und wächst erst danach leicht an. Selbst für Systeme, deren Kondition sehr nahe an 10^{16} liegt, ergibt sich ein relativer Fehler von nur ungefähr 10^{-12}. Mit $K = 3$ ist es dann sogar möglich, mit dem C-XSC Löser für alle Konditionszahlen bis etwa 10^{16}, also für alle überhaupt mit dem verwendeten Algorithmus lösbaren Systeme, ein nahezu bestmögliches Ergebnis zu erzielen.

Als nächstes wird die verifizierte Invertierung einer Matrix betrachtet. Hierbei soll vor allem der Geschwindigkeits-Vorteil des Matrix-Modus gegenüber dem normalen Vorgehen demonstriert werden. Weiterhin soll ein Vergleich zu Intlab gezogen werden. Dazu werden wieder jeweils Zufallssysteme für alle Grunddatentypen (Intervallsysteme werden wieder um einen Faktor 10^{-12} aufgebläht) mit Dimension $n = 1000$ betrachtet. Alle Berechnungen werden in diesem Test mit 8 Threads durchgeführt, C-XSC verwendet die Präzision $K = 1$. Tabelle 4.9 zeigt die Zeitmessungen, Tabelle 4.10 die relative Breite der berechneten Einschließung der entsprechenden Tests.

Löser	**real**	**interval**	**complex**	**cinterval**
C-XSC	115.9	39.1	455.5	173.0
C-XSC, Matrix-Modus	1.73	1.41	5.44	4.11
Intlab	2.60	1.39	17.4	4.14

Tabelle 4.9.: Zeitvergleich in Sekunden für die Invertierung eines Zufallsystems mit Dimension $n = 1000$, 8 Threads

Löser	**real**	**interval**	**complex**	**cinterval**
C-XSC	$2.91 \cdot 10^{-13}$	$3.58 \cdot 10^{-7}$	$6.59 \cdot 10^{-13}$	$1.33 \cdot 10^{-6}$
C-XSC, Matrix-Modus	$2.75 \cdot 10^{-13}$	$3.57 \cdot 10^{-7}$	$5.79 \cdot 10^{-13}$	$1.32 \cdot 10^{-6}$
Intlab	$6.30 \cdot 10^{-14}$	$3.57 \cdot 10^{-7}$	$2.30 \cdot 10^{-15}$	$3.57 \cdot 10^{-8}$

Tabelle 4.10.: Relative Breite der berechneten Einschließung für die Invertierung eines Zufallsystems mit Dimension $n = 1000$, 8 Threads

Bei den Zeitmessungen fällt zunächst auf, dass die Invertierung der Punktsysteme deutlich mehr Zeit in Anspruch nimmt als die Invertierung der Intervallsysteme. Grund dafür ist, dass bei der Invertierung von gut konditionierten Systemen der Fehler der Näherungslösung oft so eng eingeschlossen werden kann, dass der Radius der Einschließung bereits im Unterlaufbereich liegt. Für (echte) Intervallsysteme hingegen haben die Radiusmatrizen deutlich größere Einträge, weshalb hier kein Unterlauf auftritt. Da Rechnungen im Unterlaufbereich aufgrund fehlender Hardwareimplementierung meist deutlich langsamer als normale Gleitkommaberechnungen sind, kommt es hier also zu dem beobachteten Effekt, dass die Lösung der Intervallsysteme deutlich schneller als die Lösung der Punktsysteme berechnet werden kann.

Weiterhin zeigt der Test die großen Geschwindigkeitsvorteile des Matrix-Modus gegenüber dem normalen Modus für alle Datentypen. Verglichen mit Intlab ist C-XSC für Intervallsysteme in etwa gleich schnell. Bei reellen Punktsystemen ist Intlab etwas langsamer, vermutlich da C-XSC sich im Matrix-Modus grundsätzlich auf einen Iterationsschritt während der (nun sehr teuren) Defektiteration beschränkt. Für komplexe

Punktsysteme gibt es wiederum große Vorteile für C-XSC, welche auf ähnliche Ursachen wie in den bisherigen Tests (langsamere Matrixinvertierung in Matlab mit 8 Threads) zurückzuführen sind.

Durch die zusätzlichen Iterationsschritte während der Defektiteration ist der relative Fehler bei Intlab für Punktsysteme etwas niedriger. Für reelle Intervallsysteme ergibt sich wiederum nahezu das gleiche Ergebnis, während Intlab aus den schon erläuterten Gründen für die komplexen Intervallsysteme etwas bessere Einschließungen berechnet. Insgesamt zeigt sich aber, dass der Matrix-Modus mit Präzision $K = 1$ für die Lösung von Systemen mit vielen rechten Seiten, also z.B. bei der Bestimmung einer verifizierten Inversen, mittels des C-XSC Lösers eine sehr wichtige und nützliche Ergänzung ist.

Als nächstes folgt ein kurzer Test zu Aufwand und Qualität der Inneneinschließung. Hierzu wird ein reelles Intervallsystem mit Dimension $n = 1000$ und ein reelles Intervallsystem mit Dimension $n = 5000$ gelöst. Es wird jeweils eine Zufallsmatrix und Zufalls-Rechte-Seite erzeugt, welche um den Faktor 10^{-6} aufgebläht werden. Dabei wird der Löser jeweils einmal mit und einmal ohne Berechnung einer Inneneinschließung aufgerufen. Bei allen Tests werden 8 Threads und Präzision $K = 2$ verwendet. Außerdem wird die relative Breite der Außeneinschließung sowie der relative Abstand der Inneneinschließung von der Außeneinschließung berechnet. Tabelle 4.11 zeigt die Ergebnisse.

Löser	$n = 1000$	$n = 5000$
Zeit ohne Inneneinschließung	0.26	18.5
Zeit mit Inneneinschließung	0.36	21.1
Relative Breite	$1.04 \cdot 10^{-1}$	3.783
Relativer Abstand der Inneneinschließung	$1.36 \cdot 10^{-2}$	7.475

Tabelle 4.11.: Zeitmessung in Sekunden und Qualität der Inneneinschließung für reelle Intervallsysteme, 8 Threads, $K = 2$

Wie sich zeigt, ist der Aufwand zur Bestimmung der Inneneinschließung für kleinere Dimensionen recht hoch. Dies liegt daran, dass hier, wie in Abschnitt 4.1.1 erläutert, eine Innenrundung von $\boldsymbol{z}$ benötigt wird, welche mit dem langen Akkumulator von C-XSC berechnet werden muss. Je größer die Dimension, desto weniger fällt der Aufwand für die Bestimmung der Inneneinschließung ins Gewicht. Weiterhin ist zu sehen, dass sich der relative Abstand der Innen- von der Außeneinschließung jeweils im Bereich der relativen Breite der Außeneinschließung bewegt. Dies lässt auf eine gute Qualität der Inneneinschließung schließen.

Im nächsten Test wird Teil zwei des Lösers betrachtet, also die Abwandlung des Lösungsalgorithmus, welche eine Näherungsinverse doppelter Länge verwendet. Dieser zweite Teil ist deutlich langsamer als der erste, erlaubt aber dafür auch das Lösen von Systemen mit Konditionszahlen, die deutlich höher als 10^{16} liegen. Solche Systeme können weder mit Intlab noch mit dem Brasilien-Löser gelöst werden, weshalb diese hier nicht mit einbezogen werden.

Für diesen Test wird ein generiertes reelles Punktsystem mit einer Konditionszahl von etwa 10^{20} und der Dimension $n = 1000$ betrachtet. Das System wird mit dem C-XSC Löser mit den Präzisionen $K = 2, 3, 4$ und jeweils $P = 1, 2, 4, 8$ Threads gelöst. Dabei wird direkt der zweite Teil des Lösungsalgorithmus verwendet. Tabelle 4.12 zeigt die gemessenen Zeiten, Abbildung 4.5 den entsprechenden Speed-Up.

Threads	$K = 2$	$K = 3$	$K = 4$
1	77.9	130.2	154.0
2	38.2	64.6	77.1
4	19.1	31.9	37.9
8	10.1	17.0	20.2

Tabelle 4.12.: Zeitmessung in Sekunden für die Lösung eines Zufallssystems mit einer Kondition von etwa 10^{20} und Dimension $n = 1000$

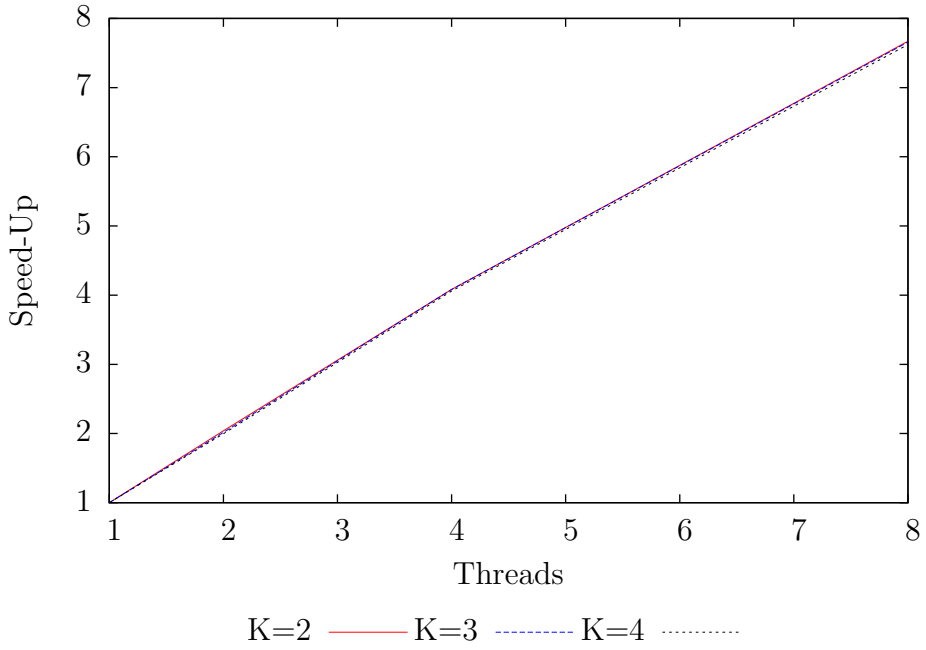


Abbildung 4.5.: Speed-Up für Teil zwei des Lösers mit reellem Punktsystem, Dimension $n = 1000$

Wie zu erwarten, sind die Laufzeiten für den zweiten Teil des Lösers deutlich höher als für Teil eins. Für $K = 1$ und einen Thread ergibt sich z.B. ein Verlangsamungs-Faktor von etwa 100 gegenüber Teil eins. Auf der anderen Seite ist die Parallelisierung hier noch einmal effizienter, im Wesentlichen wird ein idealer Speed-Up erreicht, unabhängig von der gewählten Präzision. Nur für acht Threads liegt der Speed-Up leicht

unter Idealniveau, was aber vermutlich auf die bereits erwähnte Verfälschung durch die Turbo-Boost Funktion zurückzuführen ist. Zum Abschluss wird nun der relative Fehler der berechneten Lösungseinschließungen betrachtet. Diese sind in Tabelle 4.13 zu sehen.

Threads	$K = 2$	$K = 3$	$K = 4$
1	$2.81 \cdot 10^{-7}$	$4.54 \cdot 10^{-13}$	$4.54 \cdot 10^{-13}$
2	$2.82 \cdot 10^{-7}$	$3.62 \cdot 10^{-13}$	$3.62 \cdot 10^{-13}$
4	$2.82 \cdot 10^{-7}$	$3.62 \cdot 10^{-13}$	$3.62 \cdot 10^{-13}$
8	$2.82 \cdot 10^{-7}$	$3.25 \cdot 10^{-13}$	$3.25 \cdot 10^{-13}$

Tabelle 4.13.: Relativer Fehler für die Lösung eines Zufallsystems mit einer Kondition von etwa 10^{20} und Dimension $n = 1000$

Zwar wird stets eine verifizierte Lösung gefunden, allerdings ist, wie zu erwarten, für $K = 2$ der relative Fehler vergleichsweise hoch. Für $K = 3$ hingegen ergibt sich ein deutlich kleinerer relativer Fehler der Größenordnung 10^{-13}, welcher durch die Wahl $K = 4$ erwartungsgemäß auch nicht mehr verbessert wird. In der Regel ist für Teil zwei des Lösers also eine Genauigkeit von $K = 3$ die beste Wahl.

Testergebnisse für den parallelen Löser

Im Folgenden werden einige Tests mit der parallelen Version des Lösers für Cluster-Computer basierend auf MPI beschrieben. Dabei werden auch Vergleichsmessungen mit der MPI-Version des Lösers von Mariana Kolberg [77] durchgeführt (in der Folge wiederum als Brasilien-Löser bezeichnet). Die parallele Version dieses Lösers geht analog zur seriellen Version vor, d.h. für Intervalle wird die Mittelpunk-Radius-Darstellung verwendet und es werden keinerlei Berechnungen in höherer Präzision durchgeführt. Der Löser verwendet also durchgehend ScaLAPACK bzw. PBLAS Routinen für alle Berechnungsschritte, Einschließungen werden wo nötig über die Manipulation des Rundungsmodus berechnet. Weiterhin unterstützt dieser Löser, ebenso wie die serielle Version, weder komplexe Systeme noch viele andere Funktionalitäten des C-XSC Lösers, wie z.B. mehrere rechte Seiten, den zweiten Teil des Lösers mit der Inversen doppelter Länge, über- und unterbestimmte Gleichungssysteme oder Inneneinschließungen. In der parallelen Version ist außerdem keine Funktion zum verifizierten Lösen reeller Punktsysteme enthalten.

Als erstes wird nun ein einfaches Netzwerk aus 24 Standard PCs mit Core2 Duo Prozessoren und einer Taktrate von 2.33 GHz sowie 2 GB Hauptspeicher betrachtet, welche in einem einfachen Gigabit Ethernet verbunden sind. Als Compiler kommt der GCC in Version 4.5.1 und als BLAS Bibliothek ATLAS BLAS in Version 3.8.1 zum Einsatz. Weiterhin wird ScaLAPACK in Version 2.0.1 verwendet. Für diesen Test wird explizit die Version der ATLAS Bibliothek ohne Multithreading-Unterstützung verwendet, d.h. es wird jeweils nur 1 Prozessorkern pro Knoten genutzt, so dass der Speed-Up ausschließlich von der MPI-Parallelisierung abhängt.

P	**real**	**interval**	**complex**	**cinterval**
1	124.5	180.8	589.9	690.3
2	78.7	103.3	295.7	346.5
4	53.7	69.7	187.4	212.3
8	39.2	51.4	119.1	133.9

Tabelle 4.14.: Zeit in s, Kondition 10^{10}, $n = 5000$, $K = 2$

P	**real**	**interval**	**complex**	**cinterval**
1	1.00	1.00	1.00	1.00
2	1.58	1.75	1.99	1.99
4	2.32	2.59	3.15	3.25
8	3.18	3.52	4.95	5.16

Tabelle 4.15.: Speed up, Kondition 10^{10}, $n = 5000$, $K = 2$

Getestet wird zunächst der C-XSC Löser mit einem generierten Zufallssystem der Kondition 10^{10} und Dimension $n = 5000$. Für die Intervallsysteme wurden die Matrixeinträge um einen Faktor 10^{-12} aufgebläht. Tabelle 4.14 zeigt die erzielten Zeiten, Tabelle 4.15 den Speed-Up und Tabelle 4.16 die durchschnittliche Anzahl an genauen Stellen (also Stellen, die in Ober- und Untergrenze identisch waren).

Die Unterschiede in den Laufzeiten für einen Prozess zwischen den verschiedenen Datentypen entsprechen in etwa den beim seriellen Löser beobachteten Zeitdifferenzen. Für den Speed-Up ist bei Programmen mit verteiltem Speicher das Verhältnis zwischen dem teuren Kommunikationsaufwand auf der einen und dem Rechenaufwand auf der anderen Seite von Bedeutung. Man sieht daher, dass der Speed-Up bei gleicher Prozess-Anzahl für die rechenaufwändigeren Problemstellungen, also vor allem im Komplexen, deutlich ansteigt.

Die Genauigkeit der berechneten Ergebnisse liegt auf einem zum seriellen Löser ver-

P	**real**	**interval**	**complex**	**cinterval**
1	14.6	5.0	(13.9, 14.0)	(3.8, 4.0)
2	14.6	5.0	(13.9, 14.0)	(3.8, 4.0)
4	15.3	5.0	(14.7, 14.8)	(3.8, 4.0)
8	15.3	5.0	(14.7, 14.8)	(3.8, 4.0)

Tabelle 4.16.: Durchschnittliche Stellen Genauigkeit, Kondition 10^{10}, $n = 5000$, $K = 2$

P	**Zeit in s**	**Speed-Up**	**Exakte Stellen**
1	181.4	1.0	3.04
2	137.7	1.32	3.04
4	75.1	2.42	3.31
8	55.8	3.25	3.53

Tabelle 4.17.: Vergleichswerte des Brasilien-Lösers für Intervallsysteme, Kondition 10^{10} mit Aufblähung um Faktor 10^{-12}, $n = 5000$

gleichbaren Niveau. Beim MPI-Löser werden durch die Parallelisierung allerdings auch die Skalarprodukte bei allen Matrix-Vektor-Operationen, also auch bei der Residuenberechnung, aufgeteilt. Dadurch entsteht ein ähnlicher Effekt wie bei der Verwendung mehrerer Threads mit Intlab: Durch die Aufteilung auf kleinere Unterskalarprodukte wird die Kondition in der Regel verbessert und die Ergebnisgenauigkeit steigt mit der Anzahl der Prozesse. Allerdings ist prinzipiell natürlich auch der gegenteilige Fall möglich.

Als nächstes wird zu Vergleichszwecken das identische Gleichungssystem auf dem gleichen Testsystem mit dem Brasilien-Löser gelöst. Da dieser nur reelle Intervallsysteme unterstützt, werden Zeitmessungen, Speed-Up und die durchschnittliche Anzahl exakter Stellen für diesen Testfall zusammengefasst in Tabelle 4.17 angegeben.

Die Zeiten liegen hier etwas überraschend über den Zeiten des C-XSC Lösers. Da der Brasilien-Löser auch in der parallelen Version durchgehend optimierte ScaLAPACK und PBLAS Routinen verwendet und auf Berechnungen mit (simulierter) höherer Präzision komplett verzichtet, wäre eigentlich ein leichter Zeitvorsprung zu erwarten. Anscheinend wird aber bei der Erstellung des Prozessgitters und/oder der Verteilung der Daten entsprechend der zweidimensionalen blockzyklischen Verteilung von ScaLAPACK nicht optimal vorgegangen. Dafür spricht auch, dass bei Verwendung von 2 Prozessen ein sehr deutlicher Geschwindigkeitsnachteil für den Brasilien-Löser gemessen wird (137.7 gegenüber 103.3 Sekunden). Ein solcher Abstand kann hier eigentlich nur durch eine nicht optimale Datenverteilung erklärt werden.

Weiterhin weist der Brasilien-Löser, wie schon im seriellen Fall, eine deutlich schlechtere Ergebnisgenauigkeit auf, wie an der durchschnittlichen Anzahl der exakten Stellen zu sehen ist. Allerdings profitiert die Genauigkeit hier aus den schon angesprochenen Gründen von der Benutzung weiterer Prozesse und der dadurch resultierenden Aufteilung der auftretenden Berechnungen. Die Ergebnisqualität steigt also im Allgemeinen bei Verwendung weiterer Prozesse leicht an. Insgesamt erweist sich die parallele Version des Brasilien-Lösers in den durchgeführten Tests aber nicht nur als langsamer, sondern auch als ungenauer als der C-XSC Löser.

Im nächsten Test wird ein schlecht konditioniertes System der Dimension $n = 1000$ betrachtet, für welches Teil zwei des Lösers für die Bestimmung einer verifizierten Lösung benötigt wird. Die Intervallsysteme werden wieder um einen kleinen Faktor aufgebläht.

P	**real**	**interval**	**complex**	**cinterval**
1	173.5	281.7	578.7	1047.3
2	87.6	138.1	284.9	516.0
4	42.5	69.5	147.7	260.5
8	25.1	39.0	75.2	133.8

Tabelle 4.18.: Zeit in s, Kondition 10^{17}, $n = 1000$, $K = 3$

P	**real**	**interval**	**complex**	**cinterval**
1	1.00	1.00	1.00	1.00
2	1.98	2.04	2.03	2.03
4	4.08	4.05	3.92	4.02
8	6.91	7.22	7.70	7.83

Tabelle 4.19.: Speed-Up, Kondition 10^{17}, $n = 1000$, $K = 3$

Zeiten und Speed-Up finden sich in den Tabellen 4.18 und 4.19, die Anzahl der exakten Stellen in Tabelle 4.20.

Wie schon im seriellen Fall zu sehen, ist der zweite Teil des Lösers deutlich aufwändiger und benötigt daher auch deutlich mehr Rechenzeit. Auf der anderen Seite tritt dadurch der Aufwand für die Kommunikation stärker in den Hintergrund, wodurch der Speed-Up klar besser ist, als für Teil eins des Lösers. Bereits für Dimension $n = 1000$ wird hier für die getesteten Prozess-Anzahlen ein idealer oder nahezu idealer Speed-Up erreicht. Teilweise liegt der Speed-Up sogar leicht über dem theoretischen Optimum, was vermutlich durch leichte Messungenauigkeiten erklärbar ist.

Tabelle 4.20 zeigt, dass für Punktsysteme nahezu maximal genaue Ergebnisse berechnet wurden. Für Intervallsysteme hingegen konnte jeweils keine verifizierte Einschließung berechnet werden. Dies liegt daran, dass schlecht konditionierte Systeme nahe an noch schlechter konditionierten oder sogar singulären Systemen liegen [129], welche durch die

P	**real**	**interval**	**complex**	**cinterval**
1	15.8	–	(15.8, 15.8)	–
2	15.8	–	(15.8, 15.8)	–
4	15.8	–	(15.8, 15.8)	–
8	15.8	–	(15.8, 15.8)	–

Tabelle 4.20.: Durchschnittliche Stelle Genauigkeit, Kondition 10^{17}, $n = 1000$, $K = 3$

Time in s	**P=20**	**P=50**	**P=100**
$n = 10000$	108.1	52.0	35.3
$n = 25000$	1188.0	532.2	299.7
$n = 50000$	-	-	1978.6

Tabelle 4.21.: Zeit in s, $K = 2$, gut konditioniertes reelles Punktsystem aufXC6000 Cluster

Speed Up	**P=20**	**P=50**	**P=100**
$n = 10000$	-	80.1%	59.0%
$n = 25000$	-	89.3%	79.3%
$n = 50000$	-	-	-

Tabelle 4.22.: Speed-Up in Prozent des theoretischen Optimums, $K = 2$, gut konditioniertes reelles Punktsystem auf XC6000 Cluster

hier verwendete Aufblähung von der entstehenden Intervallmatrix (vermutlich) mit eingeschlossen werden.

In der Folge sollen nun sehr große dicht besetzte Systeme betrachtet werden, welche auf einem einzelnen PC mit heute üblicher Hardware-Ausstattung nicht mehr lösbar sind. Als erstes wird dazu der XC6000 Cluster vom Karlsruher Institut für Technologie (KIT) genutzt. Dieser verfügt über 128 Intel Itanium 2 Prozessoren mit 1.5 GHz mit jeweils 6 GB Hauptspeicher, welche über ein Quadrics WsNet II Interconnect Netzwerk verbunden sind. Für diese Tests wird der Intel Compiler 10.0 und die MKL in Version 10.0 verwendet. Die Tabellen 4.21 und 4.22 zeigen Laufzeit und Speed-Up für ein reelles Punktsystem mit niedriger Konditionszahl, die Tabellen 4.23 und 4.24 für ein entsprechendes reelles Intervallsystem.

Mit - markierte Einträge in den Tabellen zur Laufzeit zeigen an, dass die jeweiligen Testsysteme zu groß waren, um mit der verwendeten Knotenanzahl gelöst werden zu können (d.h. der insgesamt zur Verfüng stehende Hauptspeicher war hier zu gering). Die Ergebnisse zeigen auch hier jeweils die Effektivität der Parallelisierung, welche auch bei

Time in s	**P=20**	**P=50**	**P=100**
$n = 10000$	136.0	64.6	42.2
$n = 25000$	1571.7	687.3	385.3
$n = 50000$	-	-	2561.1

Tabelle 4.23.: Zeit in s, $K = 2$, gut konditioniertes reelles Intervallsystem auf XC6000 Cluster

Speed Up	**P=20**	**P=50**	**P=100**
$n = 10000$	-	84.2%	64.5%
$n = 25000$	-	91.5%	81.6%
$n = 50000$	-	-	-

Tabelle 4.24.: Speed-Up in Prozent des theoretischen Optimums, $K = 2$, gut konditioniertes reelles Intervallsystem auf XC6000 Cluster

Time in s	**P=20**	**P=50**	**P=100**
$n = 10000$	95.8	44.4	30.7
$n = 25000$	1265.9	552.8	289.8
$n = 50000$	-	-	2130.0

Tabelle 4.25.: $K = 2$, gut konditioniertes reelles Punktsystem, JUMP Cluster

hundert Prozessoren einen sehr guten Speed-Up ermöglicht, sofern ein System entsprechender Größe gelöst werden soll. Auf diesem Cluster können also voll besetzte reelle Systeme der Dimension $n = 50000$ verifiziert gelöst werden.

Als letzter Testrechner wird der JUMP Cluster des Forschungszentrums Jülich betrachtet. Dieser besteht aus 14 Knoten mit je 32 IBM Power6 Prozessoren mit 4.7 GHz Taktrate und 128 GB Hauptspeicher pro Knoten, welche über ein Infiniband Netzwerk verbunden sind. Als Compiler wird hier der IBM XLC Compiler verwendet. Die Tabellen 4.25 und 4.26 zeigen Laufzeit und Speed-Up.

Die Ergebnisse entsprechen im Wesentlichen denen des XC6000 Clusters. Die absoluten Laufzeiten sind aufgrund der höheren Prozessorleistung etwas kürzer, der Speed-Up fällt sehr ähnlich aus. Auch auf diesem Cluster können Systeme der Dimension $n = 50000$ verifiziert gelöst werden.

Als abschließender Test wird nun noch Problem Nummer 7 aus dem SIAM 100 Digit Challenge [25] betrachtet. In diesem wird eine Matrix A mit den Primzahlen

$$2, 3, 5, 7, \ldots, 224737$$

Speed Up	**P=20**	**P=50**	**P=100**
$n = 10000$	-	86.3%	62.4%
$n = 25000$	-	91.6%	87.4%
$n = 50000$	-	-	-

Tabelle 4.26.: Speed-Up in Prozent des theoretischen Optimums, $K = 2$, gut konditioniertes reelles Punktsystem, JUMP Cluster

auf der Diagonalen sowie dem Eintrag 1 für alle Elemente a_{ij}, bei denen

$$|i-j| = 1, 2, 4, 8, \ldots, 16384$$

gilt, definiert. Gesucht ist nun das Element in Zeile eins und Spalten eins von A^{-1}.

In der Original-Aufgabe soll dieses Element auf 100 Stellen genau bestimmt werden. Dies ist mit dem parallelen Löser nicht bzw. nur mit einigen Modifikationen möglich, allerdings lässt sich eine Einschließung des korrekten Ergebnisses berechnen. Dazu wird die rechte Seite auf den ersten Einheitsvektor gesetzt. Das erste Element des Lösungsvektors ist dann eine Einschließung des korrekten Ergebnisses. Die exakte Lösung des Problems ist

$$A_{11}^{-1} = 0.725078346268401167\ldots$$

Der parallele Löser liefert die offensichtlich korrekte Einschließung

$$[0.7250783462684010, 0.7250783462684012].$$

Auf dem XC6000 Cluster konnte diese Einschließung in ca. 260 Sekunden berechnet werden.

Zusammenfassend haben die Tests in diesem Abschnitt gezeigt, dass sowohl die serielle bzw. mit Shared-Memory-Parallelisierung Methoden parallelisierte als auch die parallele Version des Lösers für allgemeine dicht besetzte Gleichungssysteme sehr gute Laufzeiten aufweisen, welche in etwa auf dem Niveau von Intlab liegen, dabei aber auch Einschließungen der Lösungsmenge von deutlich besserer Qualität liefern kann. Durch die vielfältigen Konfigurationsmöglichkeiten sind beide Löser zudem sehr flexibel und bieten etliche Funktionalitäten (wie z.B. Inneneinschließungen), die (soweit dem Autor bekannt) nicht in anderen entsprechenden Softwarepakten zu finden sind.

Der parallele Löser ermöglicht die verifizierte Lösung auch sehr großer dicht besetzter Gleichungssysteme. Er ist zudem der einzige Löser dieser Art, der auch das Lösen komplexer Systeme erlaubt. Weiterhin bietet er wesentlich mehr Optionen und deutlich bessere Ergebnisqualität als der einzige zur Verfügung stehende alternative Löser für diese Problemstellung.

4.2. Parametrische Gleichungssysteme

Die in Abschnitt 4.1 beschriebenen Löser sind ein nützliches Werkzeug zur Lösung allgemeiner linearer Gleichungssysteme. Sie eigenen sich gut für Anwendungen wie die Verifizierung der Regularität einer Matrix oder der Bestimmung einer verifizierten Einschließung der Inversen einer Matrix. Allerdings ist es mit ihnen nicht möglich, Abhängigkeiten in der Systemmatrix oder auch der rechten Seite auszudrücken, welche in praktischen Anwendungen häufig auftreten.

Bei symmetrischen Intervallmatrizen oder in Fällen, in denen Unsicherheiten in der realen Welt (z.B. Schwankungen in den Eigenschaften von Bauteilen) durch Intervalle modelliert werden können, sind oft lineare Intervall-Gleichungssysteme zu lösen, bei

denen verschiedene Intervalleinträge der Systemmatrix und/oder der rechten Seite voneinander abhängig sind. Werden diese Abhängigkeiten nicht berücksichtigt, wie z.B. bei Benutzung des allgemeinen Lösers aus Abschnitt 4.1, so kann es zu deutlichen Überschätzungen in der berechneten Einschließung der Intervallhülle der exakten Lösungsmenge kommen.

In diesem Abschnitt werden nun Löser vorgestellt, welche solche Abhängigkeiten berücksichtigen und für Systeme mit solchen Abhängigkeiten in der Regel bessere Einschließungen der Lösung berechnen. Dazu wird zunächst auf einen Ansatz nach Rump, mit einer Erweiterung nach Popova, eingegangen, welcher ebenfalls auf dem Krawczyk-Operator basiert und eine Modifizierung des in Abschnitt 4.1 beschriebenen Algorithmus darstellt. Dieser Ansatz wird auch genutzt, um einen Löser für symmetrische Intervallgleichungssysteme zu implementieren. Danach folgt die Beschreibung einer alternativen Methode nach Neumaier und Pownuk, welche für bestimmte Typen von Gleichungssystemen, die vor allem bei der für praktische Anwendungen sehr wichtigen Finite-Elemente-Methode auftreten, bessere Einschließungen liefern kann und die Klasse der lösbaren Systeme bei großen Unsicherheiten deutlich erweitert.

Neben dem theoretischen Hintergrund und Details zur Implementierung der jeweiligen Löser werden zum Abschluss dieses Abschnitts diverse Zeitmessungen und numerische Tests beschrieben, teilweise mit direktem Bezug zu praktischen Anwendungen wie der Modellierung elektrischer Schaltungen und Modellierungen mit der Finite-Elemente-Methode aus dem Bereich der Baustatik. Dabei wird auch untersucht, welche Auswirkung eine verifizierende Implementierung des Neumaier/Pownuk Algorithmus, welcher im Originalartikel [103] nur nicht selbstverifizierend realisiert und getestet wurde, auf Laufzeit und Ergebnisse hat und es wird ein Vergleich mit der Methode nach Popova [115] gezogen.

4.2.1. Theoretische Grundlagen

Allgemein wird in diesem Abschnitt das Problem betrachtet, eine Einschließung der Lösungsmenge eines linearen Gleichungssystems zu bestimmen, bei dem die Systemmatrix A und die rechte Seite b affin-linear von Parametern $p \in \mathbb{R}^k$ gemäß

$$a_{ij}(p) := a_{ij}^{(0)} + \sum_{\nu=1}^{k} p_\nu a_{ij}^{(\nu)}, \quad b_i(p) := b_i^{(0)} + \sum_{\nu=1}^{k} p_\nu b_i^{(\nu)}$$

abhängen, deren Werte innerhalb eines bestimmten, vorgegebenen Intervalls variieren können. Dabei sind die $A^{(i)}$ und $b^{(i)}$, $i = 0, \ldots, k$, die Koeffizientenmatrizen bzw. -vektoren für die einzelnen Parameter. $A^{(0)}$ und $b^{(0)}$ bezeichnen jeweils den parameterunabhängigen Teil. Es werden also Systeme der Form

$$A(p)x = b(p) \tag{4.16}$$

betrachtet, mit einer Systemmatrix $A \in \mathbb{R}^{n\times n}$ und rechter Seite $b \in \mathbb{R}^n$, deren Einträge von Parametern $p \in \mathbb{R}^k$ abhängen, wobei die Werte der Parameter innerhalb von $\boldsymbol{p} \in \mathbb{IR}^k$ liegen.

Bei manchen Problemstellungen können die Parameter komplexe Zahlen sein, dann variieren also die Parameter $p \in \mathbb{C}^k$ innerhalb $\boldsymbol{p} \in \mathbb{IC}^k$. Die folgenden Erläuterungen beschränken sich auf den reellen Fall. Falls der komplexe Fall besondere Überlegungen erfordert, so wird auf diese explizit eingegangen.

Eine Möglichkeit, Probleme der Form (4.16) zu Lösen ist, die zugehörige nicht parametrische Matrix $A(\boldsymbol{p}) := \square\{A(p) \mid p \in \boldsymbol{p}\}$ und die nicht-parametrische rechte Seite $b(\boldsymbol{p}) := \square\{b(p) \mid p \in \boldsymbol{p}\}$ zu bilden und das so entstandene lineare Intervall-Gleichungssystem mit der Methode aus Abschnitt 4.1 zu lösen. Die Lösungsmenge dieses nicht-parametrischen Systems ist

$$\Sigma(A([p]), b([p])) \;:=\; \{x \in \mathbb{R}^n \;\mid\; \exists A \in A([p]),\;\; \exists b \in b([p]) \;:\; Ax = b\}. \tag{4.17}$$

Die Lösungsmenge des parametrischen Systems hingegen ist definiert als

$$\Sigma_p \;=\; \Sigma(A(p), b(p), [p]) \;:=\; \{x \in \mathbb{R}^n \;\mid\; \exists p \in [p] : \;\; A(p)x = b(p)\}. \tag{4.18}$$

Die Beziehung

$$\Sigma_p \subseteq \Sigma \tag{4.19}$$

zwischen den beiden Lösungsmengen ist intuitiv verständlich, da im zugehörigen nicht-parametrischen System jedes Vorkommen eine Parameters $p_i \in \boldsymbol{p}$ unabhängig von weiteren Vorkommen behandelt wird. Einen formaler Beweis für (4.19) findet sich bei Popova [113]. Der Übergang auf das nicht-parametrische System und die Benutzung der Methode aus Abschnitt 4.1 ist für Probleme der Form (4.16) also in der Regel ungeeignet, da es zu starken Überschätzungen kommt.

Rump [127] führte daher eine Modifizierung dieser Methode in Form einer parametrisierten Fixpunkt-Iteration ein, welche zu besseren Ergebnissen führt. Hierbei ist das Ziel, eine enge Einschließung für $\boldsymbol{z} = R \cdot (b(\boldsymbol{p}) - A(\boldsymbol{p}) \cdot \tilde{x})$ zu berechnen, was durch die linearen Abhängigkeiten von $\boldsymbol{p}$ recht einfach möglich ist. Folgender Satz zeigt die entsprechende Modifizierung der Methode aus Abschnitt 4.1 für parametrische Systeme.

Satz 4.4. *Gegeben sei ein parametrisiertes lineares Gleichungssystem wie in (4.16). Weiterhin seien mit $R \in \mathbb{R}^{n \times n}$ und $\tilde{x} \in \mathbb{R}^n$*

$$\boldsymbol{z} \;:=\; R(b^{(0)} - A^{(0)}\tilde{x}) + \sum_{\nu=1}^{k} (R(b^{(\nu)} - A^{(\nu)}\tilde{x}))\boldsymbol{p}_\nu, \tag{4.20}$$

$$\boldsymbol{C} \;:=\; I - RA(\boldsymbol{p}). \tag{4.21}$$

Außerdem sei $\boldsymbol{v} \in \mathbb{IR}^n$ mit $\boldsymbol{y} \in \mathbb{IR}^n$ definiert über das Einzelschrittverfahren

$$1 \le i \le n : \;\; v_i := \{\diamond(\boldsymbol{z} + \boldsymbol{C} \cdot \boldsymbol{u})\}_i, \;\; \boldsymbol{u} := \{v_1, \ldots, v_{i-1}, y_i, \ldots, y_n\}^T. \tag{4.22}$$

Gilt nun

$$\boldsymbol{v} \subseteq \operatorname{int}(\boldsymbol{y}), \tag{4.23}$$

so sind R und $A(p)$ für alle $p \in \boldsymbol{p}$ regulär und die (eindeutige) Lösung von (4.16) liegt für alle $p \in \boldsymbol{p}$ in $\tilde{x} + \boldsymbol{v}$.

Hierbei wird üblicherweise $R := (A(\hat{p}))^{-1}$ mit $\hat{p} := \text{mid}\,(\boldsymbol{p})$ und $\tilde{x} := R \cdot b(\hat{p})$ verwendet. Da es sich um eine Modifizierung des Vorgehens aus Abschnitt 4.1 handelt, muss für den Beweis nur gezeigt werden, dass das so berechnete $\boldsymbol{z}$ tatsächlich $R \cdot (b(\boldsymbol{p}) - A(\boldsymbol{p}) \cdot \tilde{x})$ einschließt. Siehe dazu Rump [127].

Eine auf diese Art berechnete Einschließung wird in der Regel deutlich besser (also enger) sein, als eine für das nicht-parametrische System berechnete Einschließung. Allerdings führt dieses Verfahren (genau wie das Verfahren für allgemeine lineare Intervall-Gleichungssysteme) nur dann zum Erfolg, wenn die Systemmatrix stark regulär ist, da nur dann $\boldsymbol{C}$ einen Spektralradius kleiner 1 aufweist und die Iteration konvergiert. Da hier bei der Berechnung von $\boldsymbol{C}$ die nicht-parametrische Matrix $A(\boldsymbol{p})$ verwendet wird, muss also $A(\boldsymbol{p})$ stark regulär sein, damit das Verfahren aus Satz 4.4 zum Erfolg führen kann.

Um die Klasse der lösbaren Systeme zu erweitern, führte Popova eine Modifikation dieses Verfahrens ein. Dazu wird eine gesonderte Definition der starken Regularität für parametrische Matrizen verwendet [112].

Definition 4.1. *Eine parametrische Matrix $A(p)$ der Dimension $n \times n$ mit $p \in \boldsymbol{p} \in \mathbb{IR}^k$ heißt stark regulär, falls die Intervallmatrix*

$$\boldsymbol{B} := \square\{A^{-1}(\hat{p})A(p) \mid p \in \boldsymbol{p}\}$$

oder die Intervallmatrix

$$\boldsymbol{B}' := \square\{A(p)A^{-1}(\hat{p})\}$$

regulär ist.

Für die hier betrachteten Systeme mit affin-linearen Abhängigkeiten von den Parametern ergibt sich für das so definierte $\boldsymbol{B}$ [112]:

$$\boldsymbol{B} = A^{-1}(\hat{p})A^{(0)} + \sum_{\nu=1}^{k} \boldsymbol{p}_\nu (A^{-1}(\hat{p})A^{(\nu)}).$$

Mit diesem Wissen und dem Übergang auf die Näherungsinverse R wird nun (4.21) in Satz 4.4 ersetzt durch

$$\boldsymbol{C} \;:=\; I - RA^{(0)} - \sum_{\nu=1}^{k} (RA^{(\nu)})\boldsymbol{p}_\nu. \tag{4.24}$$

Die Iteration aus Satz 4.4 kann auf diese Art auch dann zum Erfolg führen, wenn die nicht-parametrische Systemmatrix $A(\boldsymbol{p})$ nicht stark regulär ist, die parametrische Systemmatrix $A(p)$, $p \in \boldsymbol{p}$ aber schon [112]. Die Klasse der so lösbaren Systeme wurde also vergrößert. Weiterhin wird $\boldsymbol{C}$ auf diese Art in der Regel auch einen geringeren Durchmesser aufweisen und die Einschließung der Lösung somit von besserer Qualität sein. Das folgende Beispiel zeigt, dass die Verwendung der Iterationsmatrix (4.24) auch dann zu einer verifizierten Lösungseinschließung führen kann, falls $A(\boldsymbol{p})$ nicht stark regulär ist.

Beispiel 4.1. *Betrachtet wird das parameterabhängige lineare Gleichungssystem*

$$A(p) = \begin{pmatrix} 3 & p & p \\ p & 3 & p \\ p & p & 3 \end{pmatrix}, \qquad b(p) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \boldsymbol{p} = [0, 2].$$

Die Systemmatrix ist offensichtlich für alle $p \in \boldsymbol{p}$ regulär. Allerdings ist die nicht-parametrische Matrix $A(\boldsymbol{p})$ nicht stark regulär, da

$$\rho(|mid(A([p]))^{-1}|rad(A([p]))) = \frac{6}{5} > 1.$$

Bei Benutzung der Iterationsmatrix (4.21) kann also keine Lösung bestimmt werden. Allerdings gilt

$$\begin{aligned} \boldsymbol{B} &= A^{-1}(\hat{p}) \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix} + A^{-1}(\hat{p}) \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \boldsymbol{p} \\ &= \begin{pmatrix} [0.8, 1.2] & [-0.3, 0.3] & [-0.3, 0.3] \\ [-0.3, 0.3] & [0.8, 1.2] & [-0.3, 0.3] \\ [-0.3, 0.3] & [-0.3, 0.3] & [0.8, 1.2] \end{pmatrix}. \end{aligned}$$

Alle Matrizen $B \in \boldsymbol{B}$ sind regulär, woraus folgt, dass die parametrische Matrix $A(p)$ stark regulär ist. Bei Verwendung der Iterationsmatrix (4.24) kann daher die Iteration aus Satz 4.4 zum Erfolg führen (und tut dies für dieses Beispiel auch).

Nachteil der Methode nach Popova mit der Iterationsmatrix (4.24) ist der im Allgemeinen deutlich höhere Berechnungsaufwand, da $k + 1$ Matrix-Matrix-Produkte mit reellen Punktmatrizen der Dimension $n \times n$ berechnet werden müssen, gegenüber einem Matrix-Matrix-Produkt zwischen einer reellen Punktmatrix und einer Intervallmatrix der Dimension $n \times n$ bei Verwendung der Iterationsmatrix (4.21). Allerdings sind die Matrizen $A^{(i)}$ häufig dünn besetzt, wodurch sich der Aufwand für die einzelnen Produkte deutlich verkleinert, falls die dünne Besetztheit bei der Implementierung berücksichtigt wird (siehe Abschnitt 4.2.2).

Wie bei den nicht-parametrischen Intervallgleichungssystemen, kann es von Vorteil sein, eine Inneneinschließung der Hülle der Lösungsmenge $\square(\Sigma_p)$ berechnen zu können. Die Berechnung erfolgt dabei analog zum nicht-parametrischen Fall.

Satz 4.5. *Führt die Iteration aus Satz 4.4 zum Erfolg (unabhängig davon ob $\boldsymbol{C}$ gemäß (4.21) oder (4.24) berechnet wird), so lässt sich mit $\boldsymbol{D} := \boldsymbol{Cv} \in \mathbb{IR}^n$ eine Inneneinschließung gemäß*

$$[\tilde{x} + \underline{z} + \overline{D}, \tilde{x} + \overline{z} + \underline{D}] \subseteq [\inf(\Sigma_p), \sup(\Sigma_p)]$$

berechnen.

Der Beweis folgt dabei dem des nicht-parametrischen Falls [127]. Bei einer Implementierung ist hier zu beachten, dass bei der Berechnung von $\boldsymbol{z}$ auch nach innen gerundet werden muss, also eine Oberschranke von $\underline{z}$ und eine Unterschranke von $\overline{z}$ benötigt werden. Da C-XSC, genau wie die meisten anderen Intervallbibliotheken, diese Möglichkeit nicht bzw. nur begrenzt vorsieht, ist bei der Implementierung einiges zu beachten. Insbesondere muss hier auch der komplexe Fall in der Implementierung gesondert betrachtet werden. Hierauf wird in Abschnitt 4.2.2 genauer eingegangen.

Wenn bei der Iteration nach Satz 4.4 eine Einschließung des Fehlers der zuvor berechneten Näherungslösung $\tilde{x}$ berechnet werden konnte, so kann diese mit einer Nachiteration oft noch verbessert werden. Dabei wird, mit der zuvor berechneten Einschließung $\boldsymbol{x}_0$, die Iteration

$$\boldsymbol{x}_{i+1} := \boldsymbol{z} + \boldsymbol{C}\boldsymbol{x}_i \cap \boldsymbol{x}_i, i = 0, 1, \ldots$$

durchgeführt. Es wird also die Iteration von einer zuvor bestimmten verifizierten Einschließung $\boldsymbol{x}_0$ ausgehend weiter fortgeführt, da sie danach in der Regel noch weiter kontrahiert. Dabei ist sichergestellt, dass kein Teil der tatsächlichen Lösungsmenge verloren geht [127]. Die Iteration wird so lange fortgeführt, bis der Unterschied zwischen zwei aufeinander folgenden Iterierten eine bestimmte, vorher festgelegte Grenze nicht überschreitet oder eine bestimmte Maximalzahl an Iterationsschritten erreicht ist.

Zum Abschluss dieses Abschnitts soll nun noch kurz auf symmetrische lineare Intervallgleichungssysteme eingegangen werden, also Systeme $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ mit symmetrischer Systemmatrix $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ und $\boldsymbol{b}, \boldsymbol{x} \in \mathbb{IR}^n$. Hierbei soll eine Einschließung $\boldsymbol{x}$ der Lösungen bezüglich aller in $\boldsymbol{A}$ enthaltenen symmetrischen Punktmatrizen bestimmt werden. Die Lösungsmenge eines solchen symmetrischen Intervallgleichungssystems ist also

$$\Sigma_{sym} := \{x \in \mathbb{R}^n \mid \exists A \in \boldsymbol{A}, b \in \boldsymbol{b} : \quad A = A^T \quad \textit{und} \quad Ax = b\}. \tag{4.25}$$

Solche Systeme können natürlich auch mit dem allgemeinen Löser aus Abschnitt 4.1 gelöst werden. Allerdings wird es dann im Allgemeinen zu starken Überschätzungen kommen, da die Symmetrie der Systemmatrix nicht berücksichtigt wird. Stattdessen lässt sich ein solches symmetrisches Intervallgleichungssystem in ein parametrisches Gleichungssystem umwandeln und mit der in diesem Abschnitt beschriebenen Methode lösen.

Dazu wird für die Elemente $\boldsymbol{a}_{ij}$ und $\boldsymbol{a}_{ji}$, $i = 1, \ldots, n$, $j = 1, \ldots, n$, $i < j$, sowie für die n Elemente auf der Hauptdiagonalen und die n Elemente der rechten Seite jeweils ein Parameter eingeführt. Insgesamt führt dies auf ein parametrisches System mit $2n+\frac{n(n-1)}{2}$ Parametern. Die zugehörigen Koeffizientenmatrizen sind sehr dünn besetzt, was in einer Implementierung beachtet werden sollte, um Rechenzeit- und Speicherbedarf klein zu halten. Eine konkrete Implementierung eines Lösers für symmetrische Gleichungssysteme, welcher eine Art Front-End für den parametrischen Löser darstellt, wird in Abschnitt 4.2.2 beschrieben.

Die mittels der in diesem Abschnitt beschriebenen Methode berechnete Lösung des so erzeugten parametrischen Systems wird in der Regel eine wesentlich bessere Einschließung von Σ_{sym} liefern als die Methode für allgemeine Intervallsysteme. Beispiele dazu finden sich in Abschnitt 4.2.4.

4.2.2. Implementierung

Die auftretenden Koeffizientenmatrizen sind oft dünn besetzt, können aber auch dicht besetzt sein. Idealerweise sollte also ein Löser für parametrische Gleichungssysteme eine Mischung aus dünn und dicht besetzten Koeffizientenmatrizen unterstützen.

Dies wird bei der Implementierung mit C-XSC dadurch erschwert, dass C-XSC keine Oberklasse für alle Matrixdatentypen verwendet (siehe auch Abschnitt 3.3.2). Stattdessen wird für die Implementierung des parametrischen Lösers für die Koeffizientenmatrizen eine Wrapper-Klasse namens `CoeffMatrix` eingeführt, welche entweder eine dünn (Typ `srmatrix`) oder einen dicht besetzte Matrix (Typ `rmatrix`) speichert. Die im Löser benötigten Funktionen und Operatoren für die Koeffizientenmatrizen werden dann für diese Klasse so definiert, dass sie auf die entsprechenden Funktionen und Operatoren des tatsächlich zu Grunde liegenden Matrixdatentyps zurückgeführt werden. Listing 4.10 zeigt den grundlegenden Aufbau der Klasse `CoeffMatrix`.

```
1  class CoeffMatrix {
2    private:
3      cxsc::rmatrix F;
4      cxsc::srmatrix S;
5      bool fullmatrix;
6
7    public:
8      CoeffMatrix() : fullmatrix(true) {}
9      CoeffMatrix(const cxsc::rmatrix& A) : fullmatrix(true), F(A) {}
10     CoeffMatrix(const cxsc::srmatrix& A) : fullmatrix(false), S(A) {}
11
12     bool isFull() const { return fullmatrix; }
13     cxsc::rmatrix& full() { return F; }
14     cxsc::srmatrix& sparse() { return S; }
15     const cxsc::rmatrix& full() const { return F; }
16     const cxsc::srmatrix& sparse() const { return S; }
17
18     //...
19
20 };
```

Listing 4.10: Grundlegender Aufbau der Klasse `CoeffMatrix` für Koeffizientenmatrizen

Die Klasse enthält jeweils ein Attribut zur Speicherung einer dünn und einer dicht besetzten Matrix, von denen jeweils nur eines belegt wird (das andere Attribut wird also als leere Matrix angelegt). Um diese Attribute zu initialisieren, gibt es jeweils einen Konstruktor mit einem dünn bzw. dicht besetzten Parameter.

Hinzu kommt ein `bool`-Attribut, mittels welchem festgehalten wird, ob es sich um eine dünn oder dicht besetzte Koeffizientenmatrix handelt. Alle Operatoren und Funktionen der Klasse `CoeffMatrix` fragen den Wert dieses Attributs ab und führen in Abhängigkeit dessen Wertes die jeweilige Operation für die gespeicherte dünn oder dicht besetzte Matrix aus.

Weiterhin kann der Wert des `bool`-Attributs über die Methode `isFull` abgefragt werden. Es ist auch möglich, über die Methoden `full` und `sparse` direkten Zugriff auf das

jeweilige Attribut von `CoeffMatrix` zu erhalten. Diese Option ist nötig, um später im eigentlichen Löser, falls erforderlich, eine separate Behandlung von dicht und dünn besetzten Koeffizientenmatrizen vornehmen zu können, die nicht automatisch von überladenen Operatoren und Standardfunktionen abgedeckt wird. Beispiele hierzu sowie zur Benutzung der Klasse `CoeffMatrix` bei der Definition des parametrischen Gleichungssystems durch den Benutzer vor dem Aufruf des Lösers folgen später in diesem Abschnitt.

Neben der Klasse `CoeffMatrix` für reelle Koeffizientmatrizen wird in analoger Weise auch eine Matrix `CoeffCMatrix` für komplexe Koeffizientenmatrizen definiert. Da bei einigen Berechnungen im Löser Einschließungen von Ausdrücken, welche Koeffizientenmatrizen enthalten, berechnet werden müssen, sind darüber hinaus Intervallversionen dieser Klassen notwendig. Daher werden zusätzlich die Klassen `CoeffIMatrix` für reelle Intervallmatrizen und `CoeffCIMatrix` für komplexe Intervallmatrizen definiert. Ihr Aufbau entspricht dem der Punkt-Koeffizientenmatrizen.

Wie die allgemeinen dicht besetzten Löser ist auch der parametrische Löser intern als Template-Funktion implementiert, welche von den für den Benutzer bereitgestellten Startfunktionen ausgeprägt und aufgerufen wird. Für den Benutzer sind also keine Templates sichtbar, aber im Kern verwendet der Löser für alle Varianten (hier also im Wesentlichen eine Version mit komplexen und eine mit reellen Intervallparametern) den gleichen Quellcode, was die Erweiterung und Wartung des Lösers stark vereinfacht. Der Kopf der Hauptfunktion des Lösers wird in Listing 4.11 angegeben.

```
1  template<typename TCoeff, typename TICoeff, typename Tb, typename Tp,
2           typename Tx, typename TInner, typename TR, typename TC,
3           typename TVec, typename TIVec, typename TDot, typename TIDot>
4   void ParLinSolveMain( vector<TCoeff>& Ap, Tb& bp, Tp& ip,
5                         Tx& xx, bool Inner, TInner& yy, int& Err,
6                         struct parlinsysconfig cfg );
```

Listing 4.11: Kopf der Hauptfunktion des parametrischen Lösers

Wie auch bei den allgemeinen dicht besetzten Lösern sind hierzu eine Vielzahl von Template-Parametern nötig, die im Folgenden kurz aufgezählt und erläutert werden:

- `TCoeff`: Der Typ der auftretenden Koeffizientenmatrizen, also `CoeffMatrix` oder `CoeffCMatrix`.
- `TICoeff`: Der passende Intervalltyp zum Typ der Koeffizientenmatrizen, also entweder `CoeffIMatrix` oder `CoeffCIMatrix`.
- `Tb`: Typ der rechten Seite. Hierbei handelt es sich in jedem Fall um einen Matrixdatentyp, da für jeden Parameter eine eigene Spalte reserviert wird. Bei mehreren rechten Seiten werden erst die Spalten der ersten rechten Seite, dann die der zweiten rechten Seite usw. aufgeführt. Hierbei kann es sich um eine voll besetzte Matrix (Typ `rmatrix` oder `cmatrix`) oder um eine dünn besetzte Matrix (Typ `srmatrix` oder `scmatrix`) handeln.
- `Tp`: Typ des Parametervektors. Dies ist stets ein Intervallvektor mit $k+1$ Einträgen (k ist die Anzahl der Parameter), wobei der erste Eintrag in der Regel den Wert 1.0

hat und für den parameterunabhängigen Teil des Systems verwendet wird. Hierbei kann es sich also um den Typ `ivector` oder den Typ `civector` handeln.

- `Tx`: Typ der berechneten Außeneinschließung. Bei einer rechten Seite ist dies ein Intervallvektor (Typ `ivector` oder `civector`), bei mehreren rechten Seiten eine Intervallmatrix (Typ `imatrix` oder `cimatrix`).
- `TInner`: Typ der berechneten Inneneinschließung. Hierfür gilt das gleiche wie für die Außeneinschließung.
- `TR`: Typ der Näherungsinversen. Diese ist in jedem Fall dicht besetzt, es kann sich also entweder um den Typ `rmatrix` oder um den Typ `cmatrix` handeln.
- `TC`: Typ der Iterationsmatrix $\boldsymbol{C}$. Auch diese ist in jedem Fall dicht besetzt und kann entweder vom Typ `imatrix` oder vom Typ `cimatrix` sein.
- `TVec`: Typ der bei der Berechnung vorkommenden Punktvektoren, also entweder `rvector` oder `cvector`.
- `TIVec`: Typ der bei der Berechnung vorkommenden Intervallvektoren, also entweder `ivector` oder `civector`.
- `TDot`: Der passende `dotprecision`-Datentyp für Punktberechnungen (entweder `dotprecision` oder `cdotprecision`).
- `TIDot`: Der passende `dotprecision`-Datentyp für Intervallberechnungen (entweder `idotprecision` oder `cidotprecision`).

Die Parameter der Hauptfunktion des Lösers haben folgende Bedeutung:

- `Ap`: Ein STL-`vector` von $n \times n$ Koeffizientenmatrizen für die einzelnen Parameter. Dabei gehört die Koeffizientenmatrix mit Index i zum i-ten Parameter, wobei die Matrix an erster Stelle den parameterunabhängigen Anteil der Systemmatrix repräsentiert.
- `bp`: Eine $(n \times k) \cdot s$ Matrix, wobei s die Anzahl der rechten Seiten darstellt. Der zum i-ten Parameter gehörige Anteil der s-ten rechten Seite findet sich dabei in Spalte $k \cdot (s-1) + i$. Die erste Spalte repräsentiert wieder den parameterunabhängigen Anteil.
- `ip`: Der Parametervektor der Länge $k+1$. Das erste Element ist in der Regel immer 1.0 und wird für den parameterunabhängigen Anteil verwendet.
- `xx`: Enthält nach erfolgreichem Abschluss der Lösers den Lösungsvektor bzw. bei mehreren rechten Seiten die Lösungsmatrix.
- `Inner`: Gibt an, ob eine Inneneinschließung berechnet werden soll (Wert `true`) oder nicht (Wert `false`). Dieser Parameter wird automatisch von den Startfunktionen gesetzt, abhängig davon, ob eine Startfunktion mit Parameter für eine Inneneinschließung aufgerufen wurde oder nicht.

- `yy`: Falls eine Inneneinschließung berechnet werden soll, so wird sie in diesem Parameter gespeichert.
- `Err`: Eine Fehlervariable, entsprechend der Fehlervariablen aus Abschnitt 4.1.
- `cfg`: Eine `struct` mit verschiedenen Konfigurationsvariablen. Diese wird im Folgenden noch genauer erläutert.

Die `struct parlinsysconfig` zur Konfiguration des parametrischen Lösers wird in Listing 4.12 gezeigt. Sie entspricht weitestgehend der `struct` für die dicht besetzten Löser aus Listing 4.3.

```
1  struct parlinsysconfig {
2    int    K;                //Dot product precision
3    bool   msg;              //Status message output?
4    int    threads;          //Number of threads for OpenMP
5    int    maxIterResCorr;   //maximum number of iterations during residual
6                             //correction (not available for K=1)
7    int    maxIterVer;       //maximum number of iterations during the
8                             //verification step
9    bool   refinement;       //Perform an iterative refinement?
10   int    maxIterRef;       //maximum number of iterations during the
11                            //refinement step
12   real   epsVer;           //Epsilon for the verification step
13   real   epsRef;           //Epsilon for the refinement step
14   bool   SharpC;           //Sharp (but slower) enclosure of iteration
15                            //matrix [C]?
16   parlinsysconfig() : K(2),msg(false),threads(-1),maxIterResCorr(10),
17                       maxIterVer(5), refinement(false), maxIterRef(5),
18                       epsVer(0.1), epsRef(1e-5), SharpC(true) {}
19 };
```

Listing 4.12: Die `struct parlinsysconfig` zur Konfiguration der parametrischen Löser

Bis auf das Attribut `SharpC` entsprechen alle Attribute und deren Standardwerte denen des allgemeinen Lösers aus Abschnitt 4.1. Der neue Parameter `SharpC` bestimmt, ob die genauere, aber aufwändiger zu berechnende Iterationsmatrix $\boldsymbol{C}$ nach Popova gemäß (4.24) oder die breitere, aber schneller zu berechnende Iterationsmatrix nach Rump gemäß (4.21) verwendet wird. Als Standard wird die Iterationsmatrix nach Popova verwendet. Der Matrix-Modus der allgemeinen Löser wird hier nicht benötigt, da die rechte Seite ohnehin als Matrix behandelt wird.

Die Implementierung des in Abschnitt 4.2.1 beschriebenen Vorgehens wird durch die vorhandenen Möglichkeiten von C-XSC und die in Kapitel 3 beschriebenen Erweiterungen sehr erleichtert. Allerdings erfordern folgende Bereiche besondere Aufmerksamkeit:

- Die Berechnung von $\boldsymbol{C}$, insbesondere die Unterscheidung der beiden Typen von Iterationsmatrizen und die Parallelisierung.
- Die Berechnung von $\boldsymbol{z}$, auch hier insbesondere die Parallelisierung.
- Die (optionale) Berechnung der Inneneinschließung.

Diese drei Punkte werden im Folgenden genauer betrachtet. Anzumerken ist außerdem, dass die Unterstützung mehrerer rechter Seiten ähnlich wie bei den allgemeinen Lösern implementiert ist. Die Berechnung von R und $\boldsymbol{C}$ erfolgt also nur einmal, alle anderen Berechnungen werden durch eine Schleife über alle rechten Seiten für jede rechte Seite nacheinander ausgeführt.

Bei der Berechnung der Iterationsmatrix $\boldsymbol{C}$ muss zunächst zwischen den beiden verschiedenen Berechnungsarten unterschieden werden. Zunächst wird der Fall betrachtet, dass die genauere Iterationsmatrix nach Popova berechnet werden soll, also das Attribut `SharpC` der Konfigurations-`struct` den Wert `true` hat. Der entsprechende Quellcode wird in Listing 4.13 gezeigt.

```
1  //Im Folgenden wird Intervallmatrix T:=C-I berechnet
2
3  #ifdef _OPENMP
4  #pragma omp parallel private(j,k) default(shared)
5  {
6     int size = n/omp_get_num_threads();
7
8     #pragma omp for
9     for(j=1 ; j<=omp_get_num_threads() ; j++) {
10      for(p=1 ; p<=k ; p++) {
11        T((j-1)*size+1,j*size,1,n) -=
12             ( R((j-1)*size+1, j*size, 1, n) * TICoeff(Ap[p-1]) ) * ip[p];
13      }
14    }
15
16    //Compute rest if matrix not evenly distributable among threads
17    #pragma omp single
18    {
19      if(n % omp_get_num_threads() != 0) {
20        for(p=1 ; p<=k ; p++) {
21          T(size*omp_get_num_threads()+1,n,1,n) -=
22          ( R(size*omp_get_num_threads()+1, n, 1, n) * TICoeff(Ap[p-1]) )
23                                                              * ip[p];
24        }
25      }
26    }
27 }
28
29 #else
30
31   for(p=1 ; p<=k ; p++) {
32     T -= (R * TICoeff(Ap[p-1])) * ip[p];
33   }
34
35 #endif
36
37 //C aus T berechnen
38 C = I + T;
```

Listing 4.13: Die Berechnung der Iterationsmatrix $\boldsymbol{C}$ nach Popova

Bei der Implementierung wird jeweils eine eigene Version mit und eine ohne OpenMP-Unterstützung verwendet. Da die Koeffizientenmatrizen oft dünn besetzt sind, ist eine implizite Parallelisierung über eine BLAS-Bibliothek mit Multithreading-Unterstützung nicht ausreichend. Daher wird die Berechnung $\diamond(-RA^{(0)} - \sum_{i=1}^{k}(RA^{(i)})\boldsymbol{p}_i)$ so ausgeführt, dass jeder Thread einen Block von $\lfloor \frac{n}{np} \rfloor$ Zeilen der Ergebnismatrix (np ist die Gesamtanzahl an Threads) berechnet. Sollte n nicht ohne Rest durch np teilbar sein, so werden die verbleibenden Zeilen danach vom Master-Thread alleine berechnet. Das Verwenden von Untermatrizen der beiden Variablen `R` und `T` ist hier relativ günstig, da es sich bei beiden immer um dicht besetzte Matrizen handelt.

Um eine Einschließung zu berechnen, werden die Koeffizientenmatrizen dabei zunächst in Intervallmatrizen umgewandelt. Hierfür sind die beiden eingangs beschriebenen Datentypen `CoeffIMatrix` und `CoeffCIMatrix` nötig. Ist OpenMP nicht aktiviert, so wird die Rechnung direkt mittels einer einfachen for-Schleife und mit Hilfe der überladenen Operatoren ausgeführt. Bei Aktivierung der BLAS-Unterstützung von C-XSC und Verwendung einer parallelisierten BLAS-Bibliothek oder der Nutzung der in Abschnitt 3.5.1 beschriebenen OpenMP-Unterstützung der entsprechenden Operatoren können aber trotzdem mehrere Prozessoren bzw. Prozessorkerne ausgenutzt werden, sofern es sich bei den Koeffizientenmatrizen um dicht besetzte Matrizen handelt.

In jedem Fall wird abschließend $\boldsymbol{C}$ mittels der Anweisung `C = I + T;` berechnet. Hierbei ist `I` die Einheitsmatrix, gespeichert als dünn besetzte Matrix. Die Addition ist also im Vergleich zu den restlichen Berechnungen günstig und eine Parallelisierung nicht lohnenswert. Als nächstes wird die Berechnung der einfachen Iterationsmatrix $\boldsymbol{C}$ nach Rump betrachtet, also der Fall, dass `SharpC` den Wert `false` aufweist. Listing 4.14 zeigt den entsprechenden Quellcode.

```
1  if(Ap[0].isFull())
2    T = Ap[0].full();
3  else
4    T = Ap[0].sparse();
5
6  for (i = 2; i <= p; i++)
7    T += ip[i]* Ap[i-1];
8
9  #ifdef CXSC_USE_BLAS
10
11   C = I - R*T;
12
13 #else
14
15 IAccu.set_k(1);
16 #ifdef _OPENMP
17 #pragma omp parallel for firstprivate(IAccu) private(i,j)  default(shared)
18 #endif
19 for (i = 1; i <= n; i++)
20   for (j = 1; j <= n; j++) {
21     IAccu = (i == j) ? 1.0 : 0.0;
22     accumulate(IAccu, -R[i], T[Col(j)]);
23     rnd(IAccu, C[i][j]);
```

```
24    }
25  IAccu.set_k(cfg.K);
26
27  #endif
```

Listing 4.14: Die Berechnung der einfachen Iterationsmatrix $\boldsymbol{C}$ nach Rump

In diesem Fall werden zunächst alle Koeffizientenmatrizen mit dem zugehörigen Parameterintervall multipliziert und anschließend aufaddiert. Für die erste Zuweisung muss dabei die `isFull`-Methode der Koeffizientenmatrix-Klasse aufgerufen und die Zuweisung von Hand vorgenommen werden, da der Zuweisungsoperator nur innerhalb der zugehörigen Klasse (hier also `imatrix`, da `T` eine dicht besetzte Intervallmatrix ist) überladen werden kann. Da die Klassen für die Koeffizientenmatrizen nur Hilfsklassen für den parametrischen Löser sind, wäre ein solcher Eingriff in die C-XSC Kernbibliothek zu diesem Zweck unpassend.

Danach wird für die weitere Berechnung zwischen aktivierter und deaktivierter BLAS-Unterstützung unterschieden. Bei aktivierter BLAS-Unterstützung kann die folgende Berechnung `C = I - R*T;` (`I` ist wieder die dünn besetzt gespeicherte Einheitsmatrix) direkt über die Operatoren durchgeführt werden. Sie wird dadurch automatisch (bei Benutzung einer passenden BLAS-Bibliothek) effektiv parallelisiert. Im anderen Fall wird die Berechnung als Skalarproduktausdruck mit zwei for-Schleifen implementiert, um so eine Parallelisierung mit OpenMP (falls aktiviert) durchführen zu können. Die Präzision wird dabei vorübergehend auf eins (also *double*-Präzision) gesetzt, was (wie beim allgemeinen Löser) für diese Berechnung ausreicht und enorme Geschwindigkeitsvorteile für das laufzeitmäßig teure Matrix-Matrix-Produkt bietet.

Als nächstes wird die Berechnung von $\boldsymbol{z}$ betrachtet, welche bei einer sehr großen Anzahl an Parametern recht aufwändig werden kann und daher auch parallelisiert werden sollte. Listing 4.15 zeigt den Quellcode zur Berechnung.

```
1  xx[Col(s)] = x;
2
3  #ifdef _OPENMP
4  #pragma omp parallel for private(k) default(shared)
5  #endif
6  for (k = 1; k <= p; k++)  {
7    T[Col(k)] = bp[Col((s-1)*p+k)] - Ap[k-1] * xx[Col(s)];
8  }
9
10 opdotprec = 1;
11 T = R * T;
12 zz = T * ip;
13 opdotprec = cfg.K;
```

Listing 4.15: Die Berechnung von $\boldsymbol{z}$

Hier wird dem Intervallvektor `xx` zunächst die Näherungslösung (Variable `x`) zugewiesen. Danach wird die Matrix `T` spaltenweise berechnet (jede Spalte wird für einen Parameter verwendet und speichert das Ergebnis von $b^{(i)} - A^{(i)} \cdot \tilde{x}$ für den i-ten Parameter). Die entsprechende for-Schleife wird mit OpenMP parallelisiert. Danach erfolgt die Multiplikation der Näherungsinversen `R` mit `T`. Das Ergebnis wird schließlich mit

dem Parametervektor multipliziert. Auch diese Berechnung wird in *double*-Genauigkeit durchgeführt, da hier (abhängig von der Anzahl der Parameter) ebenfalls ein größeres Matrix-Matrix-Produkt berechnet wird. Bei Aktivierung der BLAS-Unterstützung wird diese Berechnung durch die Parallelisierung der BLAS-Bibliothek auf Mehrkernsystemen deutlich beschleunigt. Bei deaktivierter BLAS-Unterstützung sollte auch hier wieder die OpenMP Parallelisierung der entsprechenden Operatoren genutzt werden.

Als nächstes wird die Berechnung der Inneneinschließung betrachtet. Wie bereits in Abschnitt 4.2.1 erwähnt, ist die direkte Umsetzung der Formel (siehe Satz 4.5) zur Berechnung der Inneneinschließung problematisch, da C-XSC bei Intervallberechnungen grundsätzlich nach außen rundet. Bei der Implementierung muss daher der lange Akkumulator und eine angepasste Berechnungsmethode verwendet werden.

Die Berechnung von $\boldsymbol{D} = \boldsymbol{C}\boldsymbol{x}$ ist dabei unkritisch, da hier die übliche Außenrundung vorgenommen wird. Allerdings wird eine Innenrundung von $\boldsymbol{z}$ benötigt. Um diese zu berechnen wird eine Methode nach Popova [116] verwendet, welche die Berechnung einer Inneneinschließung nur mit Außenrundung ermöglicht. Hierzu wird die Menge der reellen Intervalle $\mathbb{IR}$ um die Menge der entarteten Intervalle $\{[\underline{a}, \overline{a}] \mid \underline{a}, \overline{a} \in \mathbb{R}, \underline{a} \geq \overline{a}\}$ erweitert, wodurch man die Menge der verallgemeinerten Intervalle $\mathbb{IR}^*$ erhält. Die üblichen Intervalloperationen lassen sich direkt für diese verallgemeinern. Für Details sei an dieser Stelle auf Kaucher verwiesen [70]. Mit Hilfe einer Funktion Dual, definiert als $\mathrm{Dual}(\boldsymbol{a}) = [\overline{a}, \underline{a}]$ für $\boldsymbol{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}^*$, lässt sich eine Inneneinschließung von $\boldsymbol{z}$, hier bezeichnet als $\circ\boldsymbol{z}$, mittels

$$\circ\boldsymbol{z} = \mathrm{Dual}\left(\diamond\left(\diamond(z^{(0)}) + \sum_{i=1}^{p} \mathrm{Dual}(\boldsymbol{p}_i) \cdot \diamond(z^{(i)})\right)\right)$$

berechnen [116], wobei $z^{(i)} := R(b^{(i)} - A^{(i)}x)$ für $i = 1, \ldots, k$. Die Multiplikation eines entarteten Intervalls mit einem Intervall aus $\mathbb{IR}$ muss dabei über eine angepasste Multiplikationstabelle (siehe Popova [116]) berechnet werden. Listing 4.16 zeigt die Berechnung der Inneneinschließung.

```
1  #ifdef _OPENMP
2  #pragma omp parallel for private(cp,Accu,IAccu)
3  #endif
4  for (int j = 1; j <= n; j++) {
5
6    ivector cp = ip;
7
8    //Generalisierte Multiplikation
9    for (int i=1; i<=p; i++) {
10
11     if ( Inf(T[j][i]) >= 0 ) {
12
13       // T>=0
14       Dual(cp[i]);
15       if ( Sup(ip[i]) <= 0 ) Dual(T[j][i]);  // ip <= 0
16       else if ( Inf(ip[i]) < 0 ) UncheckedSetSup(T[j][i], Inf(T[j][i]));
17
18     } else if ( Sup(T[j][i]) <= 0 ) {
```

```
19
20        // T<=0
21        if ( Sup(ip[i]) <= 0 ) Dual(T[j][i]);
22        else if ( Inf(ip[i]) < 0 ) UncheckedSetInf(T[j][i], Sup(T[j][i]));
23
24      } else {
25
26        // 0 in T
27        if ( Inf(ip[i]) < 0 && Sup(ip[i]) > 0 ) cp[i] = interval(0);
28        else if ( Sup(ip[i]) <= 0 ) {
29          Dual(T[j][i]);
30          UncheckedSetInf(cp[i], Sup(ip[i]));
31        } else {
32          UncheckedSetSup(cp[i], Inf(ip[i]));
33        }
34      }
35    }
36
37    Accu = 0.0;
38    accumulate(Accu, Inf(T[j]), Inf(cp));
39    UncheckedSetInf(zz[j], rnd(Accu, RND_DOWN));
40
41    Accu = 0.0;
42    accumulate(Accu, Sup(T[j]), Sup(cp));
43    UncheckedSetSup(zz[j],  rnd(Accu, RND_UP));
44
45    IAccu = x[j];
46    IAccu += zz[j];
47    accumulate(IAccu, C[j], xx);
48    yy[j] = rnd(IAccu);s
49    Dual(yy[j]);
50  }
```

Listing 4.16: Berechnung einer Inneneinschließung

Die $z^{(i)}$ wurden bereits zuvor bei der Bestimmung der Außenrundung von $\boldsymbol{z}$ berechnet und in der Matrix `T` gespeichert. Diese kann hier wiederverwendet werden. Die innere for-Schleife entspricht der Umsetzung der angesprochenen Multiplikationstabelle. Die äußere for-Schleife über die einzelnen Elemente von $\circ\boldsymbol{z}$ wird mittels OpenMP parallelisiert.

Für komplexe Systeme wird das beschriebene Vorgehen angepasst. Das dort auftretende Matrix-Vektor Produkt von `T` und `ip` wird auf vier reelle Produkte aufgeteilt, für welche jeweils obiges Vorgehen verwendet wird. Implementierung und Parallelisierung entsprechen daher im Wesentlichen dem reellen Fall.

Bei der Berechnung der Inneneinschließung können sich leere Menge ergeben. In diesem Fall kann keine Aussage getroffen werden, die entsprechenden Elemente der Inneneinschließung y, bei denen $\underline{y_i} > \overline{y_i}$ ist, werden auf `SignalingNaN` gesetzt.

Im Folgenden soll nun kurz auf den Aufruf des Lösers eingegangen werden. Der Benutzer ruft nicht die Template-Funktion, sondern eine der vorgegebenen Startfunktionen auf. Es sind Startfunktionen für viele verschiedene Kombinationen von Datentypen (z.B. komplexe oder reelle Parameter, dünn oder dicht besetzte rechte Seiten, ...) defi-

niert. Diese haben eine ähnliche Form wie die in Listing 4.17 gezeigte Deklaration einer Startfunktion.

```
1 void ParLinSolve(std::vector<cxsc::CoeffMatrix>& A, cxsc::srmatrix& bp,
2                  cxsc::ivector& ip, cxsc::ivector& xx, cxsc::ivector& yy,
3                  int& err, struct parlinsysconfig cfg=parlinsysconfig());
```

Listing 4.17: Beispiel für eine Startfunktion des parametrischen Lösers

Die Startfunktion in diesem Beispiel dient der Lösung eines parametrischen Systems mit reellen Parametern und einer dünn besetzten rechten Seite. Wird der Parameter `yy` weggelassen, so wird keine Inneneinschließung berechnet. Die `struct` zur Konfiguration ist ein Default-Parameter, d.h. sie bekommt eine Instanz mit den Standardwerten zugewiesen, falls nicht explizit eine eigene Konfiguration übergeben wird.

Um zu demonstrieren, wie der Löser aufzurufen ist, wird das System aus Beispiel 4.1 betrachtet. Listing 4.18 zeigt, wie dieses System angelegt und der Löser aufgerufen werden kann.

```
1  vector<CoeffMatrix> Ap; //Koeffizientenmatrizen (als STL-vector)
2  rmatrix A(3,3);
3
4  //parameterunabhaengiger Anteil
5  A = 0.0;
6  A[1][1] = A[2][2] = A[3][3] = 3.0;
7  Ap.push_back(CoeffMatrix(A));
8
9  //parameterabhaengiger Anteil
10 A = 1.0;
11 A[1][1] = A[2][2] = A[3][3] = 0.0;
12 Ap.push_back(CoeffMatrix(A));
13
14 //rechte Seite
15 srmatrix bp(3,2);
16 bp = 0.0;
17 bp[1][1] = 1.0;
18
19 //Parametervektor
20 ivector p(2);
21 p[1] = 1.0;
22 p[2] = interval(0,2);
23
24 ivector x(3),y(3);
25 int Err;
26
27 //Eigene Konfiguration anlegen
28 parlinsysconfig cfg;
29 cfg.refinement = true; //Nachiteration aktivieren
30
31 //Aufruf des Loesers
32 //Loesung wird in x, Inneneinschliessung in y gespeichert
33 ParLinSolve(Ap, bp, p, x, y, Err, cfg);
```

Listing 4.18: Beispiel für den Aufruf des parametrischen Lösers

Beachtenswert ist hier, dass für das Anlegen der Koeffizientenmatrizen nur eine dicht besetzte reelle Matrix verwendet wird, welche nacheinander mit den auftretenden Koeffizientenmatrizen belegt und aus welcher dann jeweils eine `CoeffMatrix` erzeugt und dem `vector` für die Koeffizientenmatrizen hinzugefügt wird. Hierbei wäre natürlich auch eine Mischung von dicht und dünn besetzten Matrizen möglich. Für das Anlegen des Systems sind aber in jedem Fall eine dicht und eine dünn besetzte Matrix ausreichend, welche nacheinander entsprechend belegt werden und aus welchen dann die Koeffizientenmatrizen vom Typ `CoeffMatrix` erzeugt werden können. In diesem Beispiel wird eine Inneneinschließung mitberechnet, ebenso wird die Standardkonfiguration angepasst. Ein Aufruf ohne die Parameter `y` und `cfg` würde das gleiche System mit der Standardkonfiguration lösen, ohne eine Inneneinschließung zu berechnen.

Zum Abschluss dieses Kapitels soll noch kurz auf den Löser für symmetrische Gleichungssysteme eingegangen werden. Wie bereits in Abschnitt 4.2.1 beschrieben, können symmetrische Intervallgleichungssysteme in parametrische umgewandelt werden, wodurch in der Regel bessere Einschließungen der Lösungsmenge berechnet werden können. Der Löser `SymLinSolve` für symmetrische Gleichungssysteme ist entsprechend eine Art Front-End für den parametrischen Löser, welcher ein gegebenes symmetrisches Gleichungssystem in ein parametrisches Gleichungssystem umwandelt. Hierzu wird für die Elemente $\boldsymbol{a}_{ij}$ und $\boldsymbol{a}_{ji}$, $i = 1, \ldots, n$, $j = 1, \ldots, i-1$ ein gemeinsamer Parameter angelegt. Dazu kommt jeweils ein Paramter für die Elemente der Diagonalen von $\boldsymbol{A}$ sowie für die Elemente der rechten Seite. Die Koeffizientenmatrizen sind daher alle dünn besetzt und werden auch entsprechend gespeichert.

Zu beachten ist, dass jede dünn besetzte Matrix im CCS-Format einen Speicheraufwand von wenigstens $\mathcal{O}(n)$ aufweist. Dadurch entsteht also bei Verwendung dieses Lösers für symmetrische Gleichungssystem ein Speicherbedarf von $\mathcal{O}(n^3)$. Die hier genutzte Lösungsmethode ist also nur für kleinere Gleichungssysteme anwendbar.

Um den Aufruf des Lösers zu demonstrieren wird wieder das System aus Beispiel 4.1 betrachtet. Listing 4.19 zeigt das Anlegen dieses Systems und den Aufruf des symmetrischen Lösers.

```
1  //Systemmatrix
2  imatrix A(3,3);
3  A = interval(0,2);
4  A[1][1] = A[2][2] = A[3][3] = 3.0;
5
6  //rechte Seite
7  ivector b(3);
8  b = 0.0;
9  b[1] = 1.0;
10
11 ivector x(3),y(3);
12 int Err;
13
14 //Eigene Konfiguration anlegen
15 parlinsysconfig cfg;
16 cfg.refinement = true; //Nachiteration aktivieren
17
```

```
18 //Aufruf des Loesers
19 //Loesung wird in x, Inneneinschliessung in y gespeichert
20 SymLinSolve(A, b, x, y, Err, cfg);
```

Listing 4.19: Beispiel für den Aufruf des symmetrischen Lösers

Für die Systemmatrix und die rechte Seite wird eine einfache Intervallmatrix bzw. ein einfacher Intervallvektor angelegt. Wie zuvor wird eine eigene Konfiguration verwendet (die `struct` zur Konfiguration ist die gleiche wie beim parametrischen Löser) und eine Inneneinschließung wird berechnet. In diesem Beispiel ist zu beachten, dass genau genommen nicht das gleiche System wie beim Aufruf des parametrischen Lösers gelöst wird, da hier nur von einer elementweisen Symmetrie der einzelnen Elemente abseits der Hauptdiagonalen ausgegangen wird, während im ursprünglichen System aus Beispiel 4.1 eigentlich alle Elemente abseits der Hauptdiagonalen dem gleichen Parameter entsprechen. Für allgemeine symmetrische Systeme, wie sie in Abschnitt 4.2.1 definiert wurden, bietet dieser Löser eine gute Lösungsmöglichkeit mit deutlich besseren Einschließungen der Lösungsmenge als der allgemeine Löser aus Abschnitt 4.1. Darauf wird auch bei den Tests in Abschnitt 4.2.4 noch genauer eingegangen. Spezielle Symmetrien wie im angegebenen Beispiel sollten in den meisten Fällen aber manuell in ein parametrisches System umgewandelt werden, um bestmögliche Ergebnisse zu erzielen.

4.2.3. Ein alternativer Ansatz nach Neumaier und Pownuk

Der in Abschnitt 4.2.2 vorgestellte Löser auf Basis der theoretischen Überlegungen aus Abschnitt 4.2.1 liefert gute Ergebnisse für ein breites Spektrum von parametrischen Gleichungssystemen. Durch die optionale Verwendung der Iterationsmatrix nach Popova wird die Klasse der theoretisch lösbaren Systeme erweitert auf stark reguläre parametrische Systeme gemäß Definition 4.1 (allerdings auf dem Rechner eingeschränkt durch Rundungsfehler, welche insbesondere die Qualität der Näherungsinversen und damit auch direkt die Lösbarkeit beeinflussen).

Insbesondere bei Systemen, in denen die Parameter p in einem breiten Intervall $\boldsymbol{p}$ variieren, oder bei Systemen größerer Dimension, ist die starke Regularität der parametrischen Systemmatrix aber oft nicht mehr gegeben. Zur Demonstration dient das folgende Beispiel nach Neumaier [103].

Beispiel 4.2. *Betrachtet wird das parametrische Gleichungssystem*

$$\begin{pmatrix} p_1 + p_2 & p_1 - p_2 \\ p_1 - p_2 & p_1 + p_2 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 6 \\ 6 \end{pmatrix}$$

mit

$$p \in \boldsymbol{p} = \begin{pmatrix} [1-\delta, 1+\delta] \\ [1-\delta, 1+\delta] \end{pmatrix}.$$

Das System ist lösbar für $0 \leq \delta < 1$. *Die optimale Einschließung der Lösungsmenge lässt sich für dieses Beispiel analytisch berechnen. Es ergibt sich der Lösungsvektor*

$$\boldsymbol{x} = \begin{pmatrix} [\frac{3}{1+\delta}, \frac{3}{1-\delta}] \\ [\frac{3}{1+\delta}, \frac{3}{1-\delta}] \end{pmatrix}.$$

Für die Matrix $\boldsymbol{B}$ aus Definition 4.1 (Seite 210) zur Prüfung auf starke Regularität der parametrischen Systemmatrix ergibt sich

$$\boldsymbol{B} = \begin{pmatrix} [1.5-\delta, 1.5+\delta] & [0.5-\delta, 0.5+\delta] \\ [0.5-\delta, 0.5+\delta] & [1.5-\delta, 1.5+\delta] \end{pmatrix}.$$

Die Matrix $\boldsymbol{B}$ ist nur regulär für $\delta < 0.5$, woraus folgt, dass die parametrische Systemmatrix auch nur für $\delta < 0.5$ stark regulär ist (dies gilt auch bei Verwendung einer rechtsseitigen Inversen, also für die Matrix $\boldsymbol{B}'$ aus Definition 4.1). Für $\delta \geq 0.5$ führt die Methode aus Abschnitt 4.2.1 also nicht zum Erfolg.

Mit der bisherigen Methode berechnete Einschließungen der Lösungsmenge werden außerdem um so breiter, je näher δ an 0.5 liegt. In Abschnitt 4.2.4 folgt ein entsprechender Test, der dies verdeutlicht. In der Praxis, z.B. bei der Finite Elemente Methode für Stabtragwerke (diese werden ebenfalls in Abschnitt 4.2.4 näher betrachtet), treten häufig symmetrisch positiv definite Systeme auf, bei denen Schwankungen nur in den Steifigkeitskoeffizienten auftreten, wodurch das zugehörige System die Struktur

$$A^T \boldsymbol{D} A x = F\boldsymbol{b}$$

annimmt. Hierbei ist $\boldsymbol{D}$ eine Diagonalmatrix und Unsicherheiten in Form von Intervallen treten nur in $\boldsymbol{D}$ und $\boldsymbol{b}$ auf. Für solche Systeme gibt es eine alternative Methode nach Neumaier und Pownuk [103], welche die Struktur des Systems besser ausnutzt als die Methode aus Abschnitt 4.2.1 und daher auch für Systeme mit nicht stark regulärer parametrischer Systemmatrix, z.B. dem System in Beispiel 4.2, erfolgreich eine Einschließung berechnen kann. Diese Methode wird im Folgenden näher erläutert, später in diesem Abschnitt wird eine konkrete Implementierung für C-XSC beschrieben.

Zunächst wird die allgemeinere Form

$$(K + B\boldsymbol{D}A)\boldsymbol{x} = a + F\boldsymbol{b} \tag{4.26}$$

der angesprochenen Systeme betrachtet, mit $K, F \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $A \in \mathbb{R}^{m\times n}$, $D \in \mathbb{IR}^{m\times m}$, $a \in \mathbb{R}^n$, $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{IR}^n$. Intervalle treten also nur in $\boldsymbol{D}$ und $\boldsymbol{b}$ auf. Im Folgenden werden keine weiteren Voraussetzungen an die Struktur des Systems gestellt, allerdings liefert die beschriebene Methode im Allgemeinen nur dann gute Einschließungen, wenn es sich bei $\boldsymbol{D}$ um eine Diagonalmatrix handelt [103]. Zunächst wird ein entsprechendes Punktsystem betrachtet.

Satz 4.6. *Es sei ein System der Form (4.26), aber mit $D \in \mathbb{R}^{m\times m}$ und $b \in \mathbb{R}^n$, gegeben. Weiterhin sei mit $\hat{D} \in \mathbb{R}^{n\times n}$ die Matrix $K + B\hat{D}A$ invertierbar. Für die Lösung x des Systems gilt dann mit $v := Ax$, $C := (K + B\hat{D}A)^{-1}$ und $d := (\hat{D} - D)v$*

$$x = Ca + CFb + CBd \tag{4.27}$$

sowie

$$v = ACa + ACFb + ACBd. \tag{4.28}$$

Gibt es nun Vektoren $w, w', w'' \in \mathbb{R}^m$ *mit* $w \geq 0$, $w'>0$, *so dass*

$$w' \leq w - |\hat{D} - D||ACB|w$$

und

$$w'' \geq |\hat{D} - D||ACa + ACFb|,$$

dann gilt mit

$$\alpha := \max_i \frac{w_i''}{w_i'}$$

die Einschließung

$$d \in \boldsymbol{d} \subseteq [-\alpha w, \alpha w]. \tag{4.29}$$

Der Beweis findet sich bei Neumaier und Pownuk [103]. Beim Übergang auf Intervalle, also Systeme der Form (4.26) mit $\boldsymbol{D} \in \mathbb{IR}^{m\times m}$ und $\boldsymbol{b} \in \mathbb{IR}^n$, wird $\hat{D} = \operatorname{mid}(\boldsymbol{D})$ gewählt und es werden in den Formeln aus Satz 4.6 entsprechende Intervalloperationen verwendet. Ist dann die Bedingung $w' > 0$ aus Satz 4.6 erfüllt, so liefert (4.29) eine erste Einschließung für $\boldsymbol{d}$.

Mit der in (4.29) bestimmten Einschließung kann danach die Iteration

$$\boldsymbol{v} = \{(ACa) + (ACF)\boldsymbol{b} + (ACB)\boldsymbol{d}\} \cap \boldsymbol{v},\ \boldsymbol{d} = \{(\hat{D} - \boldsymbol{D})\boldsymbol{v}\} \cap \boldsymbol{d} \tag{4.30}$$

durchgeführt werden, um die Einschließung von $\boldsymbol{d}$ weiter zu verbessern. Die Iteration wird bis zum Eintreten eines Abbruchkriteriums, z.B. bis sich zwei aufeinander folgende Iterierte um weniger als einen bestimmten Faktor unterscheiden, durchgeführt. Mit dem so berechneten $\boldsymbol{d}$ lässt sich die gesuchte Einschließung der Lösung dann über

$$\boldsymbol{x} := (Ca) + (CF)\boldsymbol{b} + (CB)\boldsymbol{d}$$

berechnen. Die Werte in Klammern müssen (und sollten) bei der Iteration dabei nur einmal berechnet werden. Die Methode wird in einer verifizierenden und einer nicht verifizierenden Variante implementiert. Die vorausgesetzte Invertierbarkeit von $K+B\hat{D}A$ wird in der verifizierenden Version dadurch bestätigt, dass mittels des allgemeinen Lösers für dicht besetzte Systeme eine Einschließung der Inversen bestimmt wird.

Entscheidend ist nun, wie die Vektoren w, w', w'' aus Satz 4.6 zu wählen sind, um die Voraussetzungen des Satzes zu erfüllen und so überhaupt erst eine erste Einschließung von $\boldsymbol{d}$ berechnen zu können. Hierzu wird für w ein beliebiger Vektor $w \geq 0$ gewählt, es werden also z.B. alle Elemente von w auf eins gesetzt. Mit

$$w' := w - |\hat{D} - \boldsymbol{D}||ACB|w$$

und

$$w'' := |\hat{D} - \boldsymbol{D}||ACa + (ACF)\boldsymbol{b}|$$

sind die Voraussetzungen von Satz 4.6 dann erfüllt, falls $w' > 0$ gilt.

Eine im Allgemeinen bessere Methode für die Wahl von w ist, den größten Eigenwert der Matrix

$$M := |\hat{D} - \boldsymbol{D}||ACB|$$

zu bestimmen. Ist dieser kleiner eins, so wird jeder Vektor $w > 0$, der den zugehörigen Eigenvektor ausreichend gut annähert, die Bedingung $w^{'} > 0$ erfüllen. Am Rechner kann ein solches w z.B. über eine Lanczos Iteration berechnet werden.

Für sehr große Unsicherheiten, also sehr breite Intervalleinträge in $\boldsymbol{D}$, kann mit (4.29) aber häufig keine Einschließung bestimmt werden. Für den in der Praxis wichtigen Spezialfall von (4.26) mit $K = 0$, $a = 0$ und $B = A^T$, also Gleichungssysteme der Form

$$A^T \boldsymbol{D} A\boldsymbol{x} = F\boldsymbol{b},$$

bei welchen $\boldsymbol{D}$ eine Diagonalmatrix und positiv ist, gibt es eine alternative Methode, eine erste Einschließung für $\boldsymbol{d}$ und $\boldsymbol{v}$ finden, welche oft auch für breitere Intervalle zum Erfolg führt. Dazu wird wie folgt vorgegangen.

Mit $\boldsymbol{c} := (\hat{D}ACF)\boldsymbol{b}$ gilt in diesem Fall [103] für $i = 1, \ldots, m$ mit

$$\boldsymbol{z}_i = \frac{1}{2}\left(\boldsymbol{c}_i + [-1, 1]\sqrt{|\boldsymbol{c}_i|^2(1 - \overline{D}_{ii}/\underline{D}_{ii}) + \overline{D}_{ii}\sum_{k=1}^{m} |\boldsymbol{c}_k|^2/\underline{D}_{kk}} \right),$$

dass

$$\boldsymbol{v}_i := \frac{\boldsymbol{z}_i}{\boldsymbol{D}_{ii}}$$

und

$$\boldsymbol{d}_i := \left(\frac{\hat{D}_{ii}}{\boldsymbol{D}_{ii}} - 1 \right) \boldsymbol{z}_i$$

Einschließungen für $\boldsymbol{d}$ und $\boldsymbol{v}$ liefern.

Die eigentliche Iteration wird auch in diesem Fall gemäß (4.30) vorgenommen. Tests für diesen Spezialfall sowie den allgemeinen Fall und Vergleiche mit der Methode aus Abschnitt 4.2.1 folgen in Abschnitt 4.2.4.

Implementierung

Im Folgenden wird die Implementierung der hier beschriebenen Methode diskutiert. Hierbei sind im Wesentlichen, von Versionen mit dünn oder dicht besetzten Matrizen abgesehen, vier verschiedene Versionen des Lösers zu implementieren:

1. Ein Löser für Systeme der Form $(K + B\boldsymbol{D}A)\boldsymbol{x} = a + F\boldsymbol{b}$.
2. Eine verifizierende Version von 1.
3. Ein Löser für Systeme der Form $A^T \boldsymbol{D} A\boldsymbol{x} = F\boldsymbol{b}$, $\boldsymbol{D}$ Diagonalmatrix und positiv.
4. Eine verifizierende Version von 3.

Die verifizierenden Versionen werden automatisch aufgerufen, falls der Benutzer in der Konfigurations-`struct` des Lösers (diese wird später in diesem Abschnitt noch erläutert) die Verifizierung aktiviert. Für den Benutzer direkt aufrufbar sind also nur die

den Punkten 1. und 3. entsprechenden Versionen des Lösers, welche durch Aktivierung der Verifizierung auf die Fälle 2 und 4 übertragen werden können. Die Implementierung der Methode ist dabei direkt möglich, da alle benötigten Funktionalitäten in C-XSC vorliegen, insbesondere über die überladenen Operatoren der Matrix- und Vektordatentypen. Zu beachten für eine effiziente Implementierung ist nur, dass die in der Iteration (4.30) mehrfach benötigten Ausdrücke (in den Klammern) nur einmal zu Beginn erfolgen sollten.

In vielen Fällen werden die Matrizen K, B, A und F dünn besetzt sein, allerdings ist dies nicht zwingend der Fall. Da der Quellcode durch die Operatorüberladung in beiden Fällen identisch ist (die als Zwischenergebnis berechneten Matrizen wie ACB sind immer dicht besetzt, da die Inverse C dicht besetzt ist), wird die Implementierung wieder über Templates vorgenommen. Der Kopf der zwei verschiedenen Versionen des Lösers wird in Listing 4.20 angegeben.

```
1  //Loeser fuer Systeme der Form (K+BDA)u = a+Fb
2  template<typename TK, typename TB, typename TD, typename TA, typename TF>
3  void ParLinSolveNPMain(const TK& K, const TB& B, const TD& D, const TA& A,
4                         ivector& u, const rvector& a, const TF& F,
5                         const ivector& b, int& Err, parlinsysnpconfig cfg);
6
7  //Loeser fuer Systeme der Form (A^T*D*A)u = Fb
8  template<typename TA, typename TD, typename TF>
9  void ParLinSolveNPMainVer(const TA& A, const TD& D, ivector& u,
10                           const TF& F, const ivector& b, int& Err,
11                           parlinsysnpconfig cfg);
```

Listing 4.20: Funktionsköpfe der Hauptfunktionen des parametrischen Lösers nach Neumaier/Pownuk

Um eine Vorkompilierung des Lösers zu ermöglichen, werden diese Templates in einer Implementierungsdatei wieder von Startfunktionen aufgerufen, welche das jeweils benötigte Template explizit ausprägen. Nur diese Startfunktionen sind für den Benutzer sichtbar, die Verwendung von Templates erfolgt also wieder nur intern. Bei den Startfunktionen kann dann nach dünn oder dicht besetzten Matrizen unterschieden werden. Listing 4.21 zeigt als Beispiel eine Startfunktion. Zwischen der verifizierenden und nicht verifizierenden Version kann der Benutzer durch ein Attribut der zur Konfiguration verwendeten `struct` wählen, welche in Listing 4.22 zu sehen ist und im Folgenden genauer erklärt wird.

```
1  void ParLinSolveNP(const srmatrix& A, const simatrix& D, ivector& u,
2                     const srmatrix& F, const ivector& b, int& Err,
3                     parlinsysnpconfig cfg) {
4    if(cfg.verified) {
5      ParLinSolveNPMainVer(A,D,u,F,b,Err,cfg);
6    } else {
7      ParLinSolveNPMain(A,D,u,F,b,Err,cfg);
8    }
```

```
9 }
```

Listing 4.21: Beispiel für eine Startfunktion des parametrischen Lösers nach Neumaier/Pownuk

```
1  struct parlinsysnpconfig {
2    int    K;              //Precision for all operations
3    bool   msg;            //Status message output?
4    int    threads;        //Number of threads for OpenMP
5    int    maxIter;        //maximum number of iterations
6    real   epsIter;        //Epsilon for stopping criterion
7    bool   verified;       //Compute verified results (this is much slower,
8                           //but results are guaranteed enclosures)
9    parlinsysnpconfig() : K(1),msg(true),threads(-1),maxIter(20),
10                          epsIter(1e-6), verified(false) {}
11 };
```

Listing 4.22: Datenstruktur zur Konfiguration des parametrischen Lösers nach Neumaier/Pownuk

Die einzelnen Attribute der Datenstruktur `parlinsysnpconfig` zur Konfiguration des Lösers haben folgende Bedeutungen:

- `K`: Die Präzision für die Berechnung von Skalarprodukten. Dies betrifft hier im Wesentlichen die für die verwendeten Operatoren zu setzende Genauigkeit.
- `msg`: Gibt an, ob Status-Meldungen ausgegeben werden sollen oder nicht.
- `threads`: Die Anzahl an Threads für OpenMP. Bei einem Wert ≤ 0 wird die maximal verfügbare Anzahl an Threads (z.B. gemäß der entsprechenden Umgebungsvariablen) für OpenMP ausgewählt. Die Anzahl der OpenMP Threads kann auch die Zahl der von BLAS bzw. LAPACK verwendeten Threads bestimmen, je nach verwendeter BLAS bzw. LAPACK Bibliothek.
- `maxIter`: Maximale Anzahl an Iterationsschritten während der Iteration (4.30).
- `epsIter`: Abbruchkriterium für die Iteration. Unterscheiden sich zwei aufeinander folgende Iterierte um weniger als den hier angegebenen Faktor, so wird die Iteration vorzeitig abgebrochen.
- `verified`: Legt fest, ob eine verifizierte Berechnung durchgeführt werden soll (`true`) oder nicht (`false`). Es wird dann von der Startfunktion automatisch die passende Hauptfunktion aufgerufen.

Standardmäßig wird der Löser mit Präzision $K = 1$ aufgerufen, welche in der Regel ausreichend ist, da bei echten Intervalleinträgen in $\boldsymbol{D}$ und $\boldsymbol{b}$ eventuelle Rundungsfehler keinen entscheidenden Einfluss mehr auf die Breite der Lösungseinschließung haben. Weiterhin sind Statusmeldungen standardmäßig deaktiviert, die Anzahl der Threads folgt der Umgebungsvariablen und es wird nicht verifiziert gerechnet. Für die Iteration

werden maximal 20 Iterationsschritte verwendet, der Faktor für das Abbruchkriterium wird auf 10^{-6} gesetzt.

In vielen praktischen Fällen sind die Intervalle in $\boldsymbol{D}$ und $\boldsymbol{b}$ so breit, dass Rundungsfehler bei der Bestimmung der Inversen C und bei den übrigen nicht-Intervalloperationen nur sehr geringe Auswirkungen haben. Falls aber ein verifiziertes Ergebnis berechnet werden soll, lässt sich das oben beschriebene Vorgehen anpassen. Die verifizierenden Versionen müssen dabei Rundungsfehler bei der Berechnung der Inversen C von $K + B\hat{D}A$ sowie bei der Berechnung der für die Iteration nötigen Matrizen ACF, ACB, CF, CB und der Vektoren ACa und Ca berücksichtigen.

Dazu wird zunächst eine verifizierte Einschließung der Inversen mittels des Lösers aus Abschnitt 4.1 bestimmt (indem die rechte Seite auf die Einheitsmatrix gesetzt wird). Die Präzision wird dabei auf $K = 1$ gesetzt und der in Abschnitt 4.1 beschriebene Matrix-Modus wird aktiviert, um eine möglichst schnelle Berechnung der verifizierten Inversen zu gewährleisten. Die nötigen Matrizen und Vektoren werden über Intervalloperationen berechnet, so dass Einschließungen der exakten Ergebnisse bestimmt werden. Der Aufwand wird durch diese Änderungen deutlich erhöht.

4.2.4. Tests und Zeitmessungen

In diesem Abschnitt sollen nun die Ergebnisse einiger Tests präsentiert und analysiert werden, sowohl für den in den Abschnitten 4.2.1 und 4.2.2 beschriebenen parametrischen Löser nach Popova als auch für den in Abschnitt 4.2.3 beschriebenen alternativen Ansatz nach Neumaier/Pownuk. Teilweise wird bei geeigneten Testfällen auch ein direkter Vergleich gezogen. Die Tests umfassen sowohl Zeitmessungen, wobei auch die Auswirkung der Parallelisierung auf die Laufzeiten genau betrachtet wird, als auch Analysen der Güte der berechneten Einschließungen. Neben allgemeinen Tests werden auch einige Testfälle aus praktischen Anwendungen präsentiert.

Als Testsystem in diesem Abschnitt dient wieder ein System mit zwei Intel Xeon E5520 Prozessoren mit jeweils 4 Kernen und einer Taktrate von 2.26 GHz, sowie 24 GB DDR3 Hauptspeicher. Als Betriebssystem kommt wiederum die 64 Bit Version von OpenSUSE 11.4 zum Einsatz, als C++ Compiler dient der Intel Compiler in Version 12.1. Die Intel MKL in Version 10.3 wird als BLAS und LAPACK Bibliothek verwendet.

Tests des Lösers auf Basis des Krawczyk-Operators

Als erster allgemeiner Test für den parametrischen Löser nach Popova wird das sogenannte $Q(2, p)$ System betrachtet. Dieses ist für eine Systemmatrix Q beliebiger Dimension n und $i, j = 1, \ldots, n$ definiert als

$$q_{ij}(p) := \begin{cases} p_j & i \leq j \\ 0 & i = j + 2 \\ 1 & sonst \end{cases}$$

$$b(p) = (p_1, \ldots, p_n)^T$$

$$p_k \in [k \pm k \cdot \delta], \quad k = 1, \ldots, n$$

Hierbei gibt δ an, um welchen Faktor die Parameterintervalle aufgebläht werden sollen. Dieser Wert kann also in den Tests variiert werden, um das Verhalten des Lösers für unterschiedliche Intervallbreiten zu untersuchen. Das $Q(2, p)$-System ist abhängig von n Parametern, d.h. es werden $n + 1$ (dünn besetzte) Koeffizientenmatrizen benötigt. Das System eignet sich daher auch gut für einen Vergleich zwischen der Benutzung dünn besetzter und dicht besetzter Datentypen für Koeffizientenmatrizen in der Implementierung.

Weiterhin ist anzumerken, dass die $Q(2, p)$-Systemmatrix spaltenweise von den Parametern p_i abhängt. Insbesondere für diese Klasse von Matrizen ist die abgewandelte Iterationsmatrix (4.24) (im Folgenden auch als $[C(p)]$ bezeichnet) deutlich effektiver als die einfache Iterationsmatrix (4.21) (im Folgenden auch als $C(\boldsymbol{p})$ bezeichnet) [112]. Im Test sollte die Verwendung der Iterationsmatrix nach Popova also zum einen zu besseren Einschließungen führen und zum anderen auch für große n und δ noch zu verifizierten Ergebnissen führen, für welche mit der einfachen Iterationsmatrix keine Lösung mehr bestimmt werden kann.

Zu Vergleichszwecken wird in diesem Test auch eine bereits bestehende Implementierung des parametrischen Lösers für C-XSC von Popova [115] herangezogen. Dieser verwendet durchgehend den Akkumulator von C-XSC für alle Berechnungen und verfügt nicht über viele der Funktionalitäten des in dieser Arbeit beschriebenen Lösers. Unter anderem sind die Koeffizientenmatrizen hier grundsätzlich dicht besetzt. Bei den Tests ist also zu erwarten, dass dieser Löser deutlich langsamer ist als der neue C-XSC Löser. Die Ergebnisqualität könnte aber durch die Verwendung des (maximal genauen) Akkumulators etwas besser ausfallen, wobei ein solcher Effekt nur für sehr schmale Parameterintervalle zu beobachten sein sollte.

Mit dem $Q(2, p)$-System werden Tests mit $\delta = 0.1$ für die Dimensionen $n = 100, \ldots, 500$ durchgeführt. Dabei werden für die neuen Löser wieder Vergleichsmessungen mit $P = 1, 2, 4, 8$ Threads vorgenommen. Weiterhin wird jeder Test einmal mit dünn und einmal mit dicht besetzten Koeffizientenmatrizen, sowie einmal mit der Iterationsmatrix $[C(p)]$ und einmal mit der Iterationsmatrix $C(\boldsymbol{p})$ durchgeführt. Ansonsten werden die Standardeinstellungen des jeweiligen Lösers verwendet. Tabelle 4.27 zeigt den Vergleich der gemessenen Zeiten bei Verwendung der Iterationsmatrix $[C(p)]$.

Zunächst fällt auf, dass der alte parametrische Löser um mehrere Größenordnungen langsamer ist als alle Varianten des neuen Lösers. Dies entspricht den Erwartungen, da der alte Löser ausschließlich exakte Skalarproduktberechnungen mittels des langen Akkumulators von C-XSC verwendet. Weiterhin ist für diesen Testfall ein klarer Zeitvorteil durch die Benutzung dünn besetzter Koeffizientenmatrizen zu erkennen. Auch gibt es deutliche Laufzeitgewinne durch die Parallelisierung des neuen parametrischen Lösers. Für den rechenintensivsten Testfall mit Dimension $n = 500$ steht so eine Laufzeit von 1931.8 Sekunden für den alten parametrischen Löser einer Laufzeit von nur 1.61 Sekunden für die schnellste Variante des neuen Lösers (dünn besetzte Koeffizientenmatrizen und 8 Threads) gegenüber.

Als nächstes soll der durch die Parallelisierung erreichte Speed-Up untersucht werden.

Löser	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
ParLinSys (Orig.)	2.26	28.8	166.8	759.9	1931.8
Dense, $P = 1$	0.204	2.21	9.59	26.4	56.9
Dense, $P = 2$	0.126	1.31	5.37	14.9	32.9
Dense, $P = 4$	0.075	0.75	3.11	8.59	18.9
Dense, $P = 8$	0.070	0.55	2.92	6.29	16.3
Sparse, $P = 1$	0.095	0.72	2.01	4.79	9.58
Sparse, $P = 2$	0.048	0.36	1.17	2.76	5.53
Sparse, $P = 4$	0.025	0.19	0.60	1.42	2.85
Sparse, $P = 8$	0.018	0.10	0.35	0.76	1.61

Tabelle 4.27.: Zeitvergleich in Sekunden für das $Q(2, p)$ System bei Verwendung der Iterationsmatrix $[C(p)]$

Abbildung 4.6 zeigt den Speed-Up für dicht besetzte Koeffizientenmatrizen, 4.7 den Speed-Up für dünn besetzte Koeffizientenmatrizen.

Für dicht besetzte Koeffizientenmatrizen ist zunächst zu sehen, dass der Speed-Up für zwei und vier Threads relativ nah am Idealverlauf liegt. Weiterhin steigt der Speed-Up mit der Dimension, da der Rechenaufwand sich erhöht und so der Verwaltungsaufwand für die Parallelisierung immer weniger ins Gewicht fällt. Auffällig ist, dass der Speed-Up für acht Threads deutlich schwächer ansteigt. Ursache ist hier zum einen die Beeinflussung des Speed-Ups bei Ausnutzung aller Kerne durch die schon in Abschnitt 4.1.4 angesprochene Turbo-Boost Funktion des Prozessors. Zum anderen ist der Aufwand für die einzelnen auftretenden Matrix-Matrix-Produkte selbst für $n = 500$ hier noch relativ gering. Allgemein ist aber zu sehen, dass der Speed-Up auch für acht Threads mit der Dimension wächst.

Ein weiterer Faktor könnte sein, dass die Matrix-Matrix-Produkte hier mit einer parallelisierten BLAS-Bibliothek innerhalb eines parallelen Blocks berechnet werden, wodurch also innerhalb eines Threads weitere Threads erzeugt werden. Dieser sogenannte *nested parallelism* lässt sich in OpenMP (welches auch von der verwendeten MKL zur Parallelisierung genutzt wird) über die Funktion `omp_set_nested` deaktivieren. Allerdings wird diese Funktion scheinbar aktuell vom Intel Compiler nicht unterstützt.

Interessanterweise steigt der Speed-Up bei Verwendung von acht Threads für $n = 200$ und $n = 400$ aber deutlich steiler an als für $n = 100, 300, 500$. Dieser Effekt kommt dadurch zustande, dass 100, 300 und 500 nicht ohne Rest durch 8 teilbar sind, die Berechnung also nicht gleichmäßig auf alle Threads verteilt werden kann. Daher muss der verbleibende Rest einzeln vom Master-Thread in einer `single`-Region von OpenMP berechnet werden, wodurch zusätzlich zu dem nicht parallelisierten Teil der Berechnung ein gewisser Verwaltungsaufwand entsteht, welcher sich negativ auf den Speed-Up auswirkt.

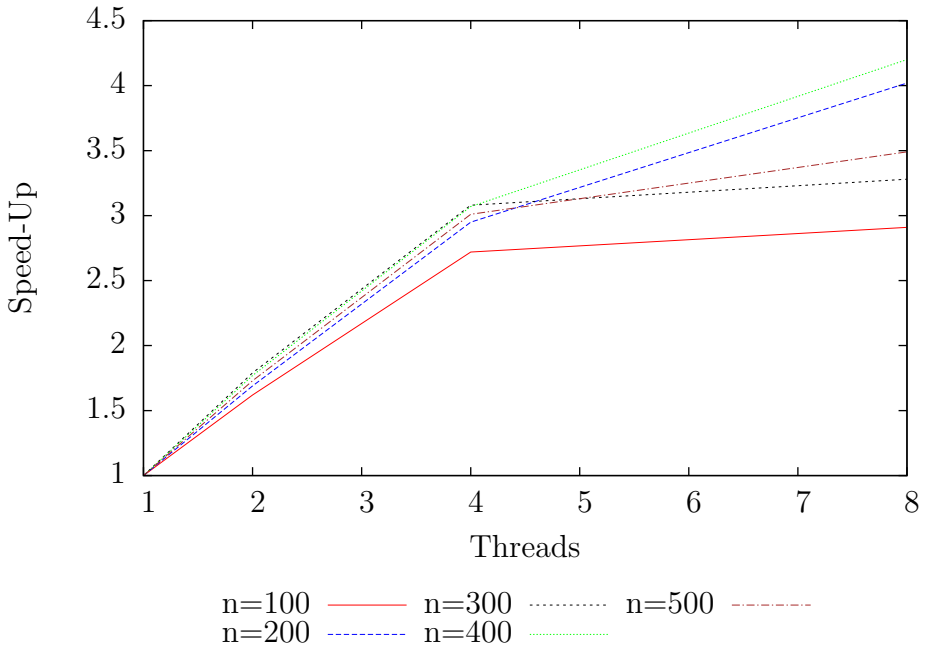


Abbildung 4.6.: Speed-Up des parametrischen Lösers bei Verwendung dicht besetzter Koeffizientenmatrizen

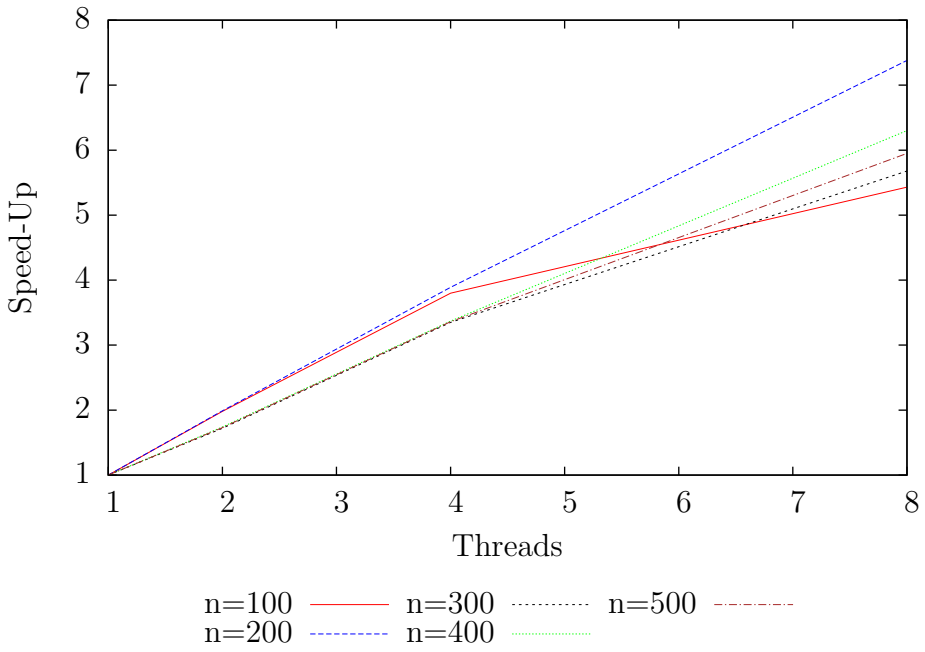


Abbildung 4.7.: Speed-Up des parametrischen Lösers bei Verwendung dünn besetzter Koeffizientenmatrizen

Für das $Q(2,p)$ System mit Dimension $n = 304$, also einer durch 8 ohne Rest teilbaren Dimension, beträgt der Speed-Up z.B. 3.84 gegenüber 3.21 für das nur unwesentlich weniger aufwändig zu lösende System mit Dimension $n = 300$.

Für dünn besetzte Koeffizientenmatrizen ist der Speed-Up insgesamt etwas besser und steigt insbesondere steiler an. Ursache hierfür ist zum einen, dass es zu keinen negativen Folgen durch *nested parallelism* kommen kann, da keine BLAS-Bibliothek verwendet wird. Zum anderen dürfte hier der negative Effekt durch die Turbo-Boost Funktion geringer sein, da Berechnungen mit dünn besetzten Matrizen, bedingt durch die verwendete Datenstruktur, Code mit sehr vielen Verzweigungen verwenden, wodurch z.B. das Pipelining hier deutlich weniger bewirken kann und der Effekt der zwischenzeitlichen Übertaktung bei Verwendung von einem Thread daher auch weniger zum Tragen kommt. Auch für dünn besetzte Koeffizientenmatrizen ist aber wieder ein negativer Effekt für $n = 100, 300, 500$ zu sehen, da wie im dicht besetzten Fall die Matrizen nicht gleichmäßig auf alle Threads verteilt werden können und daher der restliche Teil seriell berechnet werden muss.

Als nächstes werden die gleichen Tests mit Benutzung der schneller zu berechnenden Iterationsmatrix $C(\boldsymbol{p})$ durchgeführt. Die zugehörigen Zeitmessungen sind in Tabelle 4.28 gegeben.

Löser	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
ParLinSys (Orig.)	0.534	3.804	12.48	29.50	56.95
Dense, $P = 1$	0.107	0.831	2.78	6.71	14.3
Dense, $P = 2$	0.068	0.555	1.88	4.60	9.06
Dense, $P = 4$	0.037	0.295	1.01	2.59	4.92
Dense, $P = 8$	0.023	0.184	0.64	1.62	2.98
Sparse, $P = 1$	0.0093	0.040	0.105	0.215	0.361
Sparse, $P = 2$	0.0071	0.028	0.069	0.140	0.228
Sparse, $P = 4$	0.0055	0.022	0.052	0.103	0.164
Sparse, $P = 8$	0.0056	0.021	0.051	0.098	0.151

Tabelle 4.28.: Zeitvergleich in Sekunden für das $Q(2,p)$ System bei Verwendung der Iterationsmatrix $C(\boldsymbol{p})$

Insgesamt sind die gemessenen Zeiten hier erwartungsgemäß deutlich niedriger, da der Aufwand für die Berechnung der Iterationsmatrix stark gesunken ist. Ansonsten ergibt sich ein ähnliches Bild wie im vorherigen Test: Die Nutzung von dünn besetzten Koeffizientenmatrizen ist deutlich effektiver als die Nutzung dicht besetzter Koeffizientenmatrizen und durch die Parallelisierung können teilweise deutliche Geschwindigkeitsvorteile erzielt werden. Zur Verdeutlichung wird wieder der Speed-Up getrennt für dicht und dünn besetzte Koeffizientenmatrizen betrachtet. Die Abbildungen 4.8 und 4.9 zeigen die

Ergebnisse.

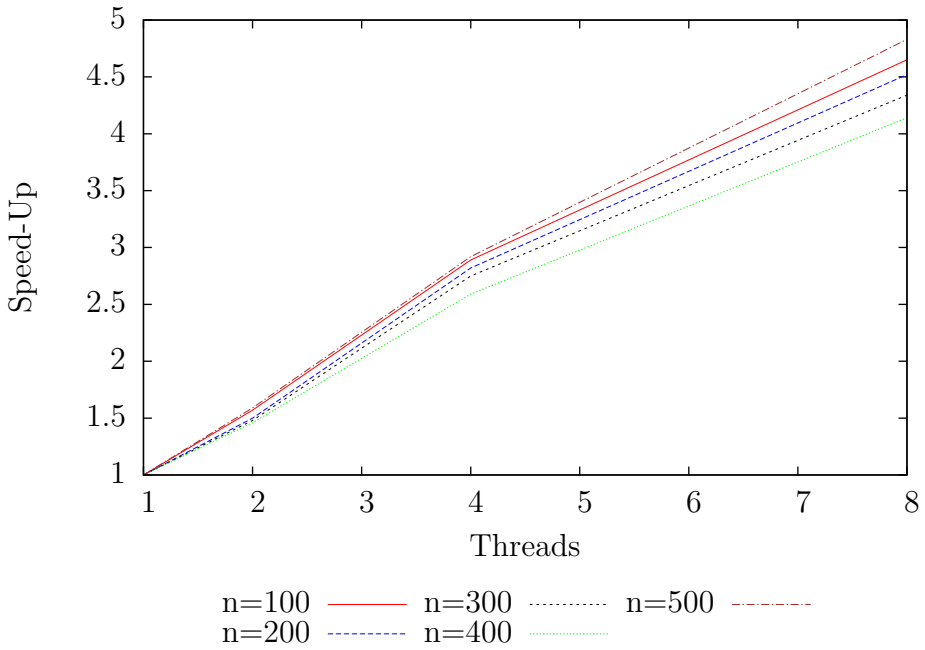


Abbildung 4.8.: Speed-Up des parametrischen Lösers mit Iterationsmatrix $C(\boldsymbol{p})$ bei Verwendung dicht besetzter Koeffizientenmatrizen

Für dicht besetzte Koeffizientenmatrizen zeigt sich ein ähnliches Verhalten wie zuvor, allerdings ist zum einen der Speed-Up insgesamt etwas besser, da nun für die Parallelisierung keine Teilmatrizen mehr ausgeschnitten werden müssen sondern alle Operationen (also die Matrixadditionen zur Bestimmung der Iterationsmatrix sowie das benötigte Matrix-Matrix-Produkt) für komplette Matrizen parallelisiert werden. Dadurch entfällt auch die Problematik für nicht ohne Rest durch die Anzahl der Threads teilbare Dimensionen.

Für dünn besetzte Matrizen ergeben sich zwar wiederum Laufzeitgewinne durch die Parallelisierung, diese Fallen aber deutlich geringer aus. Insbesondere für acht Threads nimmt der Speed-Up teilweise sogar gegenüber vier Threads ab. Ursache hierfür ist, dass in diesem Testfall vergleichsweise geringer Rechenaufwand benötigt wird und somit die Verwaltungskosten für die Parallelisierung im Vergleich viel schwerer ins Gewicht fallen als in den anderen Testfällen. Aus diesem Grund ist hier auch ein sehr ausgeprägter Zusammenhang zwischen der Erhöhung der Dimension und der Verbesserung des Speed-Ups zu erkennen. Ein Speed-Up, der an die Ergebnisse der anderen Testfälle heran reicht, wäre also erst für deutlich höhere Dimensionen und/oder deutlich mehr Parameter zu beobachten.

Als nächstes wird die Qualität der für das $Q(2, p)$ System berechneten Einschließungen betrachtet. Dabei wird zunächst die relative Breite der Einschließung für festes $\delta = 0.1$ betrachtet. Verglichen werden die alte C-XSC Implementierung von Popova mit Iterationsmatrix $[C(p)]$ sowie der neue Löser mit beiden Iterationsmatrizen, jeweils mit dünn

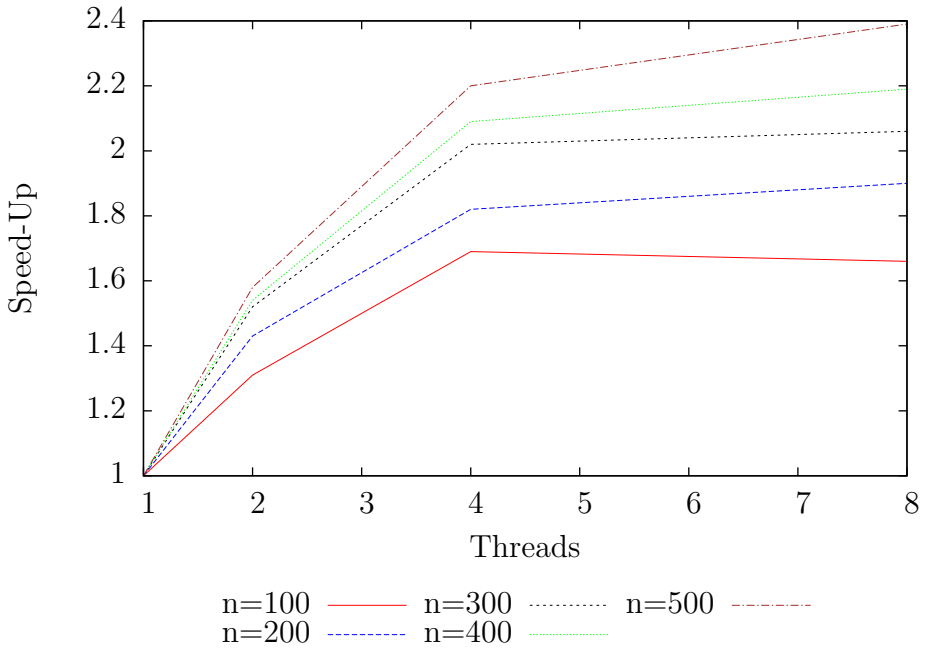


Abbildung 4.9.: Speed-Up des parametrischen Lösers mit Iterationsmatrix $C(\boldsymbol{p})$ bei Verwendung dünn besetzter Koeffizientenmatrizen

und dicht besetzten Koeffizientenmatrizen. Dabei wird nicht mehr zwischen der Anzahl der Threads unterschieden, da die relative Breite unabhängig von der Threadanzahl nahezu identisch ist. Tabelle 4.29 zeigt die Ergebnisse.

Löser	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
ParLinSys (Orig.)	$2.53 \cdot 10^{-3}$	$2.29 \cdot 10^{-3}$	$2.21 \cdot 10^{-3}$	$2.16 \cdot 10^{-3}$	$2.13 \cdot 10^{-3}$
Dense, $[C(p)]$	$2.53 \cdot 10^{-3}$	$2.29 \cdot 10^{-3}$	$2.21 \cdot 10^{-3}$	$2.16 \cdot 10^{-3}$	$2.13 \cdot 10^{-3}$
Dense, $C(\boldsymbol{p})$	$4.81 \cdot 10^{-3}$	$7.92 \cdot 10^{-3}$	$1.23 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$	$2.66 \cdot 10^{-2}$
Sparse, $[C(p)]$	$2.53 \cdot 10^{-3}$	$2.29 \cdot 10^{-3}$	$2.21 \cdot 10^{-3}$	$2.16 \cdot 10^{-3}$	$2.13 \cdot 10^{-3}$
Sparse, $C(\boldsymbol{p})$	$4.81 \cdot 10^{-3}$	$7.92 \cdot 10^{-3}$	$1.23 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$	$2.66 \cdot 10^{-2}$

Tabelle 4.29.: Relative Breite der Einschließung für $Q(2, p)$ System, $\delta = 0.1$

Die relative Breite (bei Benutzung der Iterationsmatrix $[C(p)]$) ist gegenüber dem ursprünglichen C-XSC Löser für alle Testfälle nahezu identisch. Trotz der enormen Geschwindigkeitsvorteile der neuen Löser sind also keine wesentlichen Einbußen in der Ergebnisqualität zu verzeichnen. Weiterhin zeigt sich, dass die relative Breite bei Verwendung dünn besetzter Koeffizientenmatrizen nahezu identisch zur relativen Breite bei Benutzung dicht besetzter Koeffizientenmatrizen ist. Auch in der schnellsten Variante (dünn besetzte Koeffizientenmatrizen und 8 Threads) sind also keine wesentlichen Ein-

bußen in der Qualität der berechneten Ergebnisse hinzunehmen. Diese Aussage dürfte allgemein gelten, sofern die Parameterintervalle nicht extrem schmal sind.

Bei Verwendung der schneller zu berechnenden Iterationsmatrix $C(\boldsymbol{p})$ ergibt sich, wie zu erwarten war, eine deutlich größerere relative Breite der berechneten Einschließung. Der Abstand zwischen den relativen Breiten für beide Iterationsmatrizen vergrößert sich zudem je höher die Dimension n ist.

Nun wird die relative Breite für das $Q(2, p)$ System für $\delta = 0.1, \ldots, 0.5$ betrachtet, die Parameterintervalle werden also langsam verbreitert. Da die Verwendung dicht oder dünn besetzter Koeffizientenmatrizen keinen entscheidenden Einfluss auf die relative Breite hat, wird nur die Version mit dünn besetzten Koeffizientenmatrizen betrachtet. Unterschieden wird dabei wieder zwischen den beiden Optionen für die Iterationsmatrix. Tabelle 4.30 zeigt die Ergebnisse.

Löser	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
$[C(p)]$, $\delta = 0.1$	$2.53 \cdot 10^{-3}$	$2.29 \cdot 10^{-3}$	$2.21 \cdot 10^{-3}$	$2.16 \cdot 10^{-3}$	$2.13 \cdot 10^{-3}$
$[C(p)]$, $\delta = 0.2$	$5.06 \cdot 10^{-3}$	$4.59 \cdot 10^{-3}$	$4.42 \cdot 10^{-3}$	$4.33 \cdot 10^{-3}$	$4.27 \cdot 10^{-3}$
$[C(p)]$, $\delta = 0.3$	$7.60 \cdot 10^{-3}$	$6.90 \cdot 10^{-3}$	$6.64 \cdot 10^{-3}$	$6.50 \cdot 10^{-3}$	$6.42 \cdot 10^{-3}$
$[C(p)]$, $\delta = 0.4$	$1.02 \cdot 10^{-2}$	$9.21 \cdot 10^{-3}$	$8.87 \cdot 10^{-3}$	$8.68 \cdot 10^{-3}$	$8.57 \cdot 10^{-3}$
$[C(p)]$, $\delta = 0.5$	$1.27 \cdot 10^{-2}$	$1.15 \cdot 10^{-2}$	$1.11 \cdot 10^{-2}$	$1.09 \cdot 10^{-2}$	$1.07 \cdot 10^{-2}$
$C(\boldsymbol{p})$, $\delta = 0.1$	$4.81 \cdot 10^{-3}$	$7.92 \cdot 10^{-3}$	$1.23 \cdot 10^{-2}$	$2.16 \cdot 10^{-3}$	$2.13 \cdot 10^{-3}$
$C(\boldsymbol{p})$, $\delta = 0.2$	$1.61 \cdot 10^{-2}$	$3.54 \cdot 10^{-2}$	$7.20 \cdot 10^{-2}$	$2.16 \cdot 10^{-3}$	$2.13 \cdot 10^{-3}$
$C(\boldsymbol{p})$, $\delta = 0.3$	$3.71 \cdot 10^{-2}$	$1.08 \cdot 10^{-1}$	$2.74 \cdot 10^{-1}$	$6.03 \cdot 10^{-1}$	-
$C(\boldsymbol{p})$, $\delta = 0.4$	$7.50 \cdot 10^{-2}$	$2.79 \cdot 10^{-1}$	-	-	-
$C(\boldsymbol{p})$, $\delta = 0.5$	$1.36 \cdot 10^{-1}$	$6.26 \cdot 10^{-1}$	-	-	-

Tabelle 4.30.: Relative Breite der Einschließung für $Q(2, p)$ System mit Parametern unterschiedlicher Breite

Die relative Breite wächst jeweils erwartungsgemäß mit δ. Allerdings steigt die Breit der Einschließung bei Verwendung von $C(\boldsymbol{p})$ wesentlich deutlicher an und ist allgemein auch größer als bei Verwendung von $[C(p)]$. Weiterhin ist es mit der einfachen Iterationsmatrix $C(\boldsymbol{p})$ für $\delta = 0.3$ und $n = 500$ sowie für $n = 300, 400, 500$ und $\delta = 0.4, 0.5$ bereits nicht mehr möglich, eine verifizierte Lösung zu berechnen, während mit der Iterationsmatrix $[C(p)]$ in allen Fällen eine verifizierte Lösung berechnet werden konnte.

Als nächstes werden Systeme mit komplexen Parametern betrachtet. Für einen allgemeinen Test wird dazu zunächst wieder das $Q(2, p)$ System herangezogen, wobei die Parameter jeweils mit $1 + i$ multipliziert werden. Hierbei wird nur die Version mit dünn besetzten Koeffizientenmatrizen betrachtet. Für das Verhältnis der Zeitmessungen und Ergebnisse zwischen dicht und dünn besetzten Matrizen gelten im Wesentlichen die glei-

chen Aussagen wie zuvor. Tabelle 4.31 zeigt die gemessenen Zeiten, Tabelle 4.32 die relative Breite der berechneten Einschließungen.

Löser	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
$[C(p)]$, $P = 1$	1.15	9.36	31.6	72.4	139.9
$[C(p)]$, $P = 2$	0.61	4.68	15.8	36.2	70.7
$[C(p)]$, $P = 4$	0.32	2.36	8.10	19.0	37.8
$[C(p)]$, $P = 8$	0.23	1.27	4.69	10.0	20.9
$C(\boldsymbol{p})$, $P = 1$	0.030	0.13	0.35	0.71	1.28
$C(\boldsymbol{p})$, $P = 2$	0.022	0.092	0.24	0.46	0.82
$C(\boldsymbol{p})$, $P = 4$	0.017	0.065	0.16	0.31	0.54
$C(\boldsymbol{p})$, $P = 8$	0.018	0.067	0.15	0.27	0.46

Tabelle 4.31.: Zeitvergleich in Sekunden für das $Q(2, p)$ System mit komplexen Parametern

Löser	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
Sparse, $[C(p)]$	$3.15 \cdot 10^{-3}$	$2.49 \cdot 10^{-3}$	$2.27 \cdot 10^{-3}$	$2.16 \cdot 10^{-3}$	$2.10 \cdot 10^{-3}$
Sparse, $C(\boldsymbol{p})$	$6.57 \cdot 10^{-3}$	$1.22 \cdot 10^{-2}$	$2.27 \cdot 10^{-2}$	$3.90 \cdot 10^{-2}$	$6.78 \cdot 10^{-2}$

Tabelle 4.32.: Relative Breite der Einschließung für $Q(2, p)$ System mit komplexen Parametern, $\delta = 0.1$

Im Wesentlichen lassen sich die bisherigen Aussagen auch auf Systeme mit komplexen Parametern übertragen, wobei aber durch den erheblich höheren Aufwand bei komplexen Berechnungen deutlich mehr Zeit benötigt wird. Sowohl der relative Unterschied zwischen den gemessenen Zeiten als auch das Verhalten der relativen Breite der Einschließung für die beiden unterschiedlichen Iterationsmatrizen verhält sich aber sehr ähnlich zum reellen Fall. Der Speed-Up fällt etwas höher aus als im Reellen, da hier durch den höheren Gesamtaufwand der Verwaltungsaufwand für die Parallelisierung an Bedeutung verliert, aber auch in dieser Hinsicht gibt es keine grundsätzlichen Unterschiede, weshalb auf ein Diagramm für den Speed-Up verzichtet wird.

Im Folgenden wird eine praktische Anwendung des parametrischen Lösers betrachtet. Bei der Entwicklung und Analyse elektrischer Schaltungen ist es wichtig, die möglichen Auswirkungen von Bauteiltoleranzen auf das Schaltungsverhalten zu untersuchen. Solche Toleranzen lassen sich mittels Intervallen modellieren. Als Beispiel wird die Wechselstrom-Schaltung in Abbildung 4.10 betrachtet [79].

Mittels Knotenanalyse sollen nun Schranken für die Knotenspannungen V_j, $j = 1, \ldots, 5$ gefunden werden. Ohne die Toleranzen der Bauteile zu berücksichtigen ergibt sich zu-

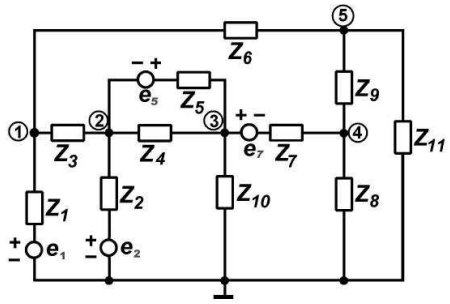


Abbildung 4.10.: Elektrische Schaltungen mit fünf Knoten (aus [79])

nächst mit

$$
\begin{aligned}
&e_1 = e_2 = 100V, e_5 = e_7 = 10V,\\
&Z_j = R_j + \imath X_j \in \mathbb{C},\ R_j = 100\Omega,\ X_j = \omega L_j - \frac{1}{\omega C_j},\ j = 1,\ldots,11,\\
&\omega = 50,\quad X_{1,2,5,7} = \omega L_{1,2,5,7} = 20,\quad X_3 = \omega L_3 = 30,\\
&X_4 = -\frac{1}{\omega C_4} = -300,\ X_{10} = -\frac{1}{\omega C_{10}} = -400,\ X_{6,8,9,11} = 0
\end{aligned}
$$

das lineare Gleichungssystem

$$
\begin{pmatrix}
\frac{1}{Z_1}+\frac{1}{Z_3}+\frac{1}{Z_6} & -\frac{1}{Z_3} & 0 & 0 & -\frac{1}{Z_6}\\
-\frac{1}{Z_3} & \frac{1}{Z_2}+\frac{1}{Z_3}+\frac{1}{Z_4}+\frac{1}{Z_5} & -\frac{1}{Z_4} & 0 & 0\\
0 & -\frac{1}{Z_4} & \frac{1}{Z_4}+\frac{1}{Z_5}+\frac{1}{Z_7}+\frac{1}{Z_{10}} & -\frac{1}{Z_7} & 0\\
0 & 0 & -\frac{1}{Z_7} & \frac{1}{Z_7}+\frac{1}{Z_8}+\frac{1}{Z_9} & -\frac{1}{Z_9}\\
-\frac{1}{Z_6} & 0 & 0 & -\frac{1}{Z_9} & \frac{1}{Z_6}+\frac{1}{Z_9}+\frac{1}{Z_{11}}
\end{pmatrix}
\begin{pmatrix} V_1\\ V_2\\ V_3\\ V_4\\ V_5 \end{pmatrix}
=
\begin{pmatrix}
\frac{e_1}{Z_1}\\
\frac{e_2}{Z_2}-\frac{e_5}{Z_5}\\
\frac{e_5}{Z_5}+\frac{e_7}{Z_7}\\
-\frac{e_7}{Z_7}\\
0
\end{pmatrix}
$$

mit

$$
\begin{aligned}
Z_1 &= Z_2 = Z_5 = Z_7 = 110 + i20\\
Z_3 &= 100 + i30\\
Z_4 &= 100 - i300\\
Z_6 &= Z_8 = Z_9 = Z_{11} = 100\\
Z_{10} &= 100 - i400.
\end{aligned}
$$

Um nun die Bauteiltoleranzen zu modellieren, werden die Impedanzen Z_i, $i = 1,\ldots,11$ jeweils mit 10% Toleranz versehen und in entsprechende Intervalle eingeschlossen. Um ein parameterabhängiges lineares Gleichungssystem mit linearer Abhängigkeit von den

Parametern zu erhalten, werden die Parameter des Systems als

$$p_i = \frac{1}{Z_i}$$

gewählt. Dieses System ist so mit dem Löser aus Abschnitt 4.2 lösbar. Als Einschließung für die Spannungen $V_j, j = 1, \ldots, 5$ ergibt sich

$$\begin{pmatrix} [4.9021900635077813E+001, 6.3782338144818816E+001] + i[-6.8400205603540068E+000, -1.0277087785239471E+000] \\ [4.0273840294108360E+001, 5.4752678432853387E+001] + i[-7.9124035354960772E+000, -1.5582600826862842E+000] \\ [1.3138301166837424E+001, 2.0989859759659016E+001] + i[-5.8195849568360592E-001, 6.5691430975830159E+000] \\ [5.6469311369255450E+000, 1.4149265965229715E+001] + i[-1.1666539312767415E+000, 2.7739633213572846E+000] \\ [1.6367827470687267E+001, 2.7832317823330030E+001] + i[-2.8213369595936980E+000, 7.3453030999456104E-001] \end{pmatrix}.$$

Für Gleichstromkreise erhält man ähnliche Gleichungssysteme mit reellen Parametern. Kolev [79, 80] und Dreyer [42] beschreiben alternative Verfahren zur Lösung dieses Problems, welche das System auf ein reelles Gleichungssystem zurückführen, wodurch sich die Dimension des Gleichungssystems verdoppelt und die Parameterabhängigkeiten in der Regel komplizierter werden. Für mehr Details siehe [114].

Eine weitere Anwendung ergibt sich bei der Lösung der Lyapunov-Gleichung

$$AX + XA^T = F,$$

mit bekannten $A, F \in \mathbb{R}^{n\times n}$ und unbekanntem $X \in \mathbb{R}^{n\times n}$. Diese Gleichung taucht vielfach in der Praxis auf, insbesondere im Bereich der Regelungstheorie, wo sie z.B. bei der Bestimmung von Steuerbarkeit und Beobachtbarkeit eines Systems eine Rolle spielt. Auch hier können wieder Toleranzen in den Matrixeinträgen, z.B. verursacht durch Messungenauigkeiten, mittels Intervallen modelliert werden, so dass die Intervall-Lyapunov-Gleichung

$$\boldsymbol{AX} + \boldsymbol{XA}^T = \boldsymbol{F}$$

mit $\boldsymbol{A}, \boldsymbol{X}, \boldsymbol{F} \in \mathbb{IR}^{n\times n}$ mit der Lösungsmenge

$$\Sigma_{Lyapunov} := \{X \in \mathbb{R}^n \mid \exists A \in \boldsymbol{A}, \exists F \in \boldsymbol{F} : AX + XA^T = F\}.$$

erhält. Diese lässt sich in ein parametrisches Gleichungssystem

$$\boldsymbol{Px} = \boldsymbol{f}$$

überführen [104, 134, 135], wobei $\boldsymbol{P} = I \otimes \boldsymbol{A} + \boldsymbol{A} \otimes I$ ($\otimes$ steht hierbei für das Kronecker-Produkt) und $\boldsymbol{f} \in \mathbb{IR}^n$ ein Vektor ist, in dem die Spalten von $\boldsymbol{A}$ untereinander notiert werden. Somit ergibt sich ein parametrisches Gleichungssystem der Dimension $n^2 \times n^2$, welches mit dem in Abschnitt 4.2 beschriebenen Löser gelöst werden kann.

Für ein Beispielsystem mit Dimension $n = 40$ (Beispiel 3.6 aus [55]) ergibt sich auf dem Testrechner mit 8 Threads und der Iterationsmatrix $[C(p)]$ eine Laufzeit von ca. 90 Sekunden. Das gegebene Problem auf diese Weise zu lösen ist nur für Systeme kleiner Dimension möglich, da sich ein Gesamtaufwand von $\mathcal{O}(n^6)$ sowohl hinsichtlich Laufzeit als auch hinsichtlich Speicherbedarf ergibt. Hashemi [55] beschreibt eine alternative Methode mit Aufwand $\mathcal{O}(n^3)$, welche allerdings in der Regel (oftmals wesentlich) breitere Einschließungen liefert.

Als weiteres allgemeines Beispiel wird eine Toeplitz-Matrix betrachtet. Bei Toeplitz-Matrizen hängt der Eintrag a_{ij} der Matrix nur von der Differenz $i-j$ ab, die Elemente auf der Haupt- und den Nebendiagonalen sind also jeweils gleich. Für Toeplitz-Matrizen mit Intervalleinträgen ist der parametrische Löser eine gute Lösungsmethode. Im Folgenden wird ein Test mit einer Toeplitz-Matrix der Dimension $n = 500, 1000, 2000$ mit Einträgen auf der Haupt- und den ersten beiden oberen und unteren Nebendiagonalen betrachtet. Hier ist, im Gegensatz zum $Q(2, p)$ System, die Dimension im Vergleich zur Anzahl der Parameter deutlich höher. Die Hauptdiagonale wird auf 4, die ersten Nebendiagonalen auf 2 und die zweiten Nebendiagonalen werden auf den Wert 1 gesetzt. Es wird jeweils eine unterschiedliche Aufblähung pro Diagonale um einen Faktor zwischen 10^{-2} und 10^{-4} gewählt, die rechte Seite wird komplett auf 1 gesetzt (ohne Parameterabhängigkeit).

Für den Test wird der neue Löser mit dünn besetzten Koeffizientenmatrizen, der genauen Iterationsmatrix $[C(p)]$ sowie $P = 1, 2, 4, 8$ Threads verwendet. Als Vergleich wird in diesem Test die Routine `verhullparam` aus dem auf Intlab aufbauenden Softwarepaket VerSoft von Rohn [118] verwendet. Diese wird mit dem gleichen Testsystem wie bisher, Matlab R2011a, Intlab 6 und 8 Threads ausgeführt. Tabelle 4.33 zeigt die erzielten Zeiten und die relative Breite der berechneten Einschließung.

Löser	$n = 500$	$n = 1000$	$n = 2000$
Zeit C-XSC, $P = 1$	0.78	3.24	19.2
Zeit C-XSC, $P = 2$	0.40	1.71	11.4
Zeit C-XSC, $P = 4$	0.23	0.98	6.70
Zeit C-XSC, $P = 8$	0.16	0.58	4.34
Relative Breite C-XSC	$3.00 \cdot 10^{-4}$	$2.99 \cdot 10^{-4}$	$2.99 \cdot 10^{-4}$
Zeit VerSoft, $P = 8$	5.4	22.5	128.9
Relative Breite VerSoft	0.96	1.35	1.92

Tabelle 4.33.: Zeit in s und relative Breite der Einschließung für Toeplitz-Matrix

Zum einen ist dabei an den Zeiten des C-XSC- Lösers zu sehen, dass durch die deutlich kleinere Anzahl an Parametern gegenüber dem $Q(2, p)$ System auch bei gleicher Dimension deutlich weniger Rechenaufwand benötigt wird. Für den Speed-Up gelten ähnliche Aussagen wie zuvor. VerSoft ist für alle Probleme nicht nur deutlich langsamer, sondern berechnet auch deutlich breitere Einschließungen. Die relative Breite der berechneten Einschließung ist um mehrere Größenordnungen höher als beim C-XSC Löser.

Als nächstes wird ein in der Literatur vielfach verwendetes Beispiel (siehe z.B. Behnke [17]) für ein symmetrisches 2×2 Intervallsystem betrachtet, für welches die exakte Intervallhülle bekannt ist. Es ist gegeben als

$$[A] = \begin{pmatrix} 3 & [1,2] \\ [1,2] & 3 \end{pmatrix}, \qquad [b] = \begin{pmatrix} [10, 10.5] \\ [10, 10.5] \end{pmatrix}.$$

Die beiden Intervalleinträge $[1,2]$ in der Systemmatrix müssen also in einem entsprechenden parametrischen Intervallsystem über einen Parameter modelliert werden. Das parametrische Intervallsystem hat also die Form

$$A(p) = \begin{pmatrix} 3 & p_1 \\ p_1 & 3 \end{pmatrix}, \quad b(p) = \begin{pmatrix} p_2 \\ p_3 \end{pmatrix}, \quad [p] = ([1,2],[10,10.5],[10,10.5])^T.$$

Eine Berechnung der Innen- und Außeneinschließung mit dem C-XSC Löser mit der Iterationsmatrix $[C(p)]$ ergibt

$$\begin{pmatrix} [2.075, 2.480] \\ [2.077, 2.479] \end{pmatrix} \subseteq {}^{\diamond}\Sigma = \begin{pmatrix} [1.810\ldots, 2.688\ldots] \\ [1.810\ldots, 2.688\ldots] \end{pmatrix} \subseteq \begin{pmatrix} [1.648, 2.907] \\ [1.648, 2.907] \end{pmatrix}.$$

Der allgemeine Löser hingegen liefert die deutlich gröbere Außeneinschließung

$$\begin{pmatrix} [0.888, 3.667] \\ [0.888, 3.667] \end{pmatrix}.$$

Im Allgemeinen ist, bei echten und nicht zu schmalen Intervalleinträgen, der Mehraufwand für die Lösung mittels des parametrischen Lösers gegenüber der Lösung mittels des allgemeinen Lösers in solchen Fällen also lohnend, da deutlich bessere Ergebnisse erzielt werden. Der Aufwand für die Umwandlung eines symmetrischen in ein parametrisches System ist im Allgemeinen vernachlässigbar.

Tests des Neumaier/Pownuk-Ansatzes und Vergleichstests

Im Folgenden sollen nun einige Tests mit dem alternativen Ansatz nach Neumaier/-Pownuk aus Abschnitt 4.2.3 durchgeführt werden (in den folgenden Abbildungen und Tabellen aus Platzgründen nur noch als Neumaier-Löser bezeichnet). Wo sinnvoll, werden dabei auch Vergleiche mit dem parametrischen Löser nach Popova durchgeführt. Zunächst wird dazu wieder Beispiel 4.2 aus Abschnitt 4.2.3 betrachtet.

Wie dort gesehen, ist das zugehörige parametrische System für $0 \leq \delta < 1$ regulär. Allerdings ist die parametrische Matrix $A(p)$ nur für $\delta < 0.5$ stark regulär, so dass das System auch nur in diesem Bereich mit dem parametrischen Löser nach Popova lösbar ist. Um das Verhalten der jeweiligen Löser zu verdeutlichen, zeigt Abbildung 4.11 die relative Breite der berechneten Einschließung für dieses System für steigende Werte von δ. Der Popova-Löser verwendet dabei die genaue Iterationsmatrix $[C(p)]$, der Neumaier/Pownuk-Löser wird in der verifizierenden Version ausgeführt. Das System lässt sich einfach in die vom Neumaier/Pownuk-Löser benötigte Form umschreiben [103]. Die Intervallhülle der exakten Lösung lässt sich jeweils über die Formel $\boldsymbol{x}_1 = \boldsymbol{x}_2 = [3/(1+\delta), 3/(1-\delta)]$ berechnen [103], die relative Breite der Einschließung wird ebenfalls in Abbildung 4.11 dargestellt.

Für niedrige δ-Werte liegen die Ergebnisse für beide Löser noch relativ nah an der exakten Intervallhülle. Je mehr sich δ dem Wert 0.5 nähert, desto stärker wächst die Breite der mit dem Löser nach Popova berechneten Einschließung. Ab dem Wert 0.5 kann keine Einschließung mehr berechnet werden. Der Neumaier/Pownuk-Löser hingegen

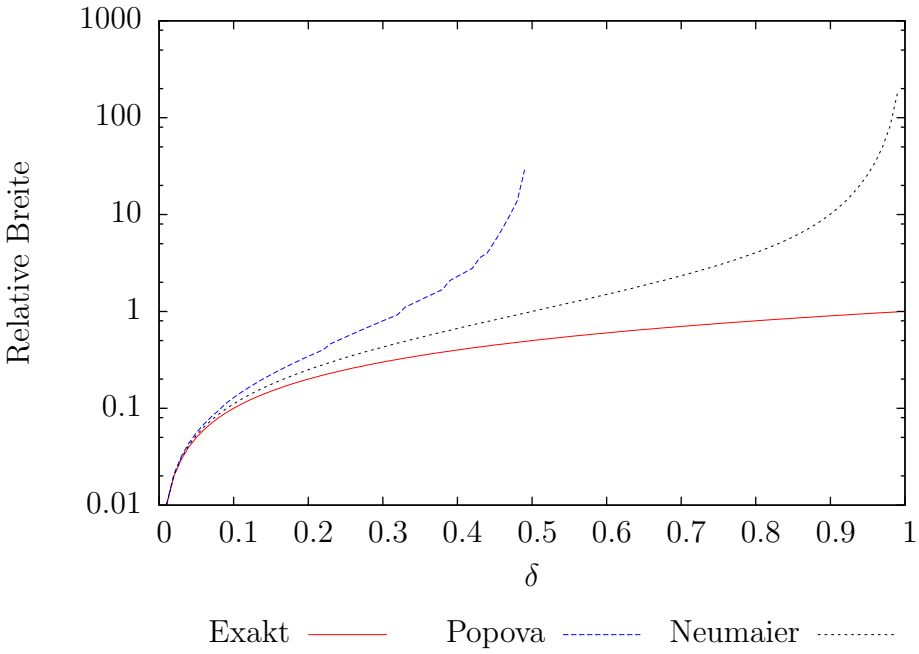


Abbildung 4.11.: Entwicklung der relativen Breite der berechneten Einschließung für beide Löser für das Neumaier-Beispiel

berechnet für alle $0 \leq \delta < 1$ eine Einschließung und liegt dabei in weiten Teilen recht nahe an der exakten Lösung. Erst ab einem Wert für δ von etwa 0.9 steigt die Breite der berechneten Einschließung deutlich an, da sich das System dann der Singularität nähert.

Ein wichtiges praktisches Beispiel für die Verwendung des Löser nach Neumaier/-Pownuk besteht in dessen Anwendung im Rahmen der Finite-Elemente-Methode für Stabtragwerke. Ein solches Tragwerk ist in Abbildung 4.12 gegeben. Hierbei sind die Eigenschaften der einzelnen Stäbe (Elastizität, Querschnitt und Länge) sowie die auf die einzelnen Knoten (die Verbindungspunkte zwischen den einzelnen Stäben) einwirkenden Kräfte bekannt und es sollen die resultierenden Knotenverschiebungen berechnet werden. Unsicherheiten bei Elastizität und Querschnitt der Stäbe (z.B. durch Schwankungen bei der Herstellung) sowie der von außen einwirkenden Kräfte (z.B. durch Messungenauigkeiten) können über Intervalle modelliert werden.

Bei der Verwendung der Finite-Elemente-Methode bei Stabtragwerken wird für jeden Stab ein Element verwendet. Da hier ein zweidimensionales Problem betrachtet wird, wirken an jedem Knoten Kräfte in x- und y-Richtung ein, woraus Verschiebungen in x- und y-Richtung entstehen. Der Zusammenhang wird mittels der lokalen Steifigkeitsmatrix ausgedrückt. Diese wird mit einer Drehmatrix C_e und einer Matrix U_e, welche die lokalen Koordinaten in globale Koordinaten überführt, multipliziert. Ein Aufaddieren aller Steifigkeitsmatrizen ergibt die globale Steifigkeitsmatrix K. Mit dieser ergibt sich das für die Problemstellung zu lösende lineare Gleichungssystem

$$Ku = f$$

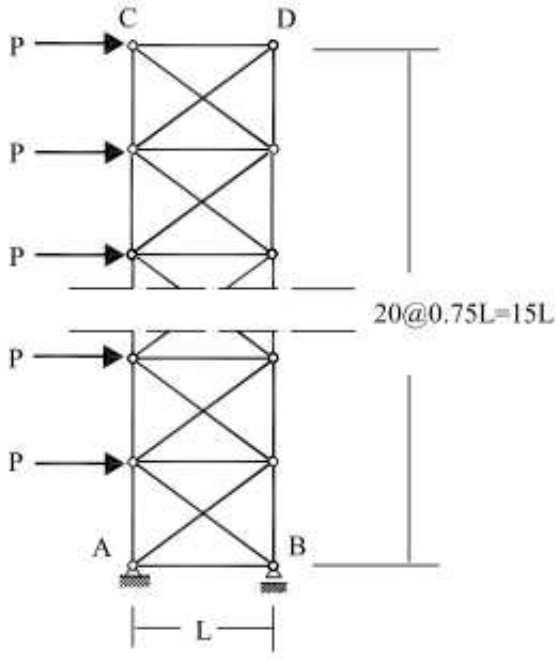


Abbildung 4.12.: Beispiel für ein Stabtragwerk (aus [103])

mit den gesuchten Knotenverschiebungen u und den gegebenen Kräften f.

Die Dimension des Systems entspricht der Anzahl der Freiheitsgrade. Im Beispiel aus Abbildung 4.12 entspricht dies (bei 20 Etagen) der Dimension $(20 * 2 + 2) * 2 - 3$ (42 Knoten, welche je in x- und y-Richtung verschoben werden können, wobei der Knoten unten links fest verankert ist und daher keine Freiheitsgrade aufweist, währen der Knoten unten rechts nur in y-Richtung verschiebbar ist). Für weitere Details zu dieser Art von Problem und zur Finite-Elemente-Methode allgemein siehe z.B. [144].

Die globale Steifigkeitsmatrix K lässt sich auch in der Form

$$K = A^T DA$$

schreiben [103], mit einer Diagonalmatrix D, welche die einzelnen Elementparameter x_e enthält. Nach dem oben angesprochenen Übergang auf Intervalle ergibt sich somit ein Gleichungssystem der Form

$$A^T \boldsymbol{D} A \boldsymbol{u} = F \boldsymbol{b},$$

welches genau der für die Anwendung der Neumaier/Pownuk-Methode benötigten Form entspricht. Das so entstandene parametrische Gleichungssystem lässt sich sowohl mit der Methode nach Popova als auch mit der nach Neumaier/Pownuk lösen.

Im Folgenden wird nun ein Vergleich zwischen beiden Methoden für das Beispielproblem aus Abbildung 4.12 durchgeführt. Dabei wird die Anzahl der Etagen s zwischen 10 und 50 variiert. Daraus ergibt sich jeweils ein Gleichungssystem mit $5s + 1$ Parametern und der Dimension $4s + 4 - 3$. Weiterhin wird die Toleranz für die Elementparameter x_e und die Kräfte F zwischen 1% und 5% variiert, um Intervalle unterschiedlicher Breite

zu erhalten. Verglichen wird jeweils die Laufzeit und die Breite der berechneten Einschließung für den Löser nach Popova mit Iterationsmatrix $[C(p)]$ sowie den Löser nach Neumaier/Pownuk in verifizierender und nicht verifizierender Version. Es werden jeweils 8 Threads verwendet. Tabelle 4.34 zeigt die gemessenen Zeiten, Tabelle 4.35 die erzielten Ergebnisse.

Toleranz	**Löser**	$s = 10$	$s = 20$	$s = 30$	$s = 40$	$s = 50$
1%	Popova	0.0046	0.015	0.039	0.083	0.161
	Neumaier	0.0055	0.0069	0.014	0.021	0.030
	Neumaier ver.	0.032	0.029	0.061	0.107	0.169
5%	Zeit Popova	0.0029	0.014	0.038	0.083	0.161
	Neumaier	0.0019	0.0054	0.015	0.021	0.032
	Neumaier ver.	0.0076	0.027	0.063	0.111	0.173

Tabelle 4.34.: Zeiten in s für Stabtragwerk-Berechnung

Tol.	**Löser**	$s = 10$	$s = 20$	$s = 30$	$s = 40$	$s = 50$
1%	Popova	$2.48 \cdot 10^{-2}$	$3.07 \cdot 10^{-2}$	$4.17 \cdot 10^{-2}$	$6.04 \cdot 10^{-2}$	$9.19 \cdot 10^{-2}$
	Neumaier	$2.14 \cdot 10^{-2}$	$2.11 \cdot 10^{-2}$	$2.10 \cdot 10^{-2}$	$2.10 \cdot 10^{-2}$	$2.10 \cdot 10^{-2}$
	Neumaier ver.	$2.14 \cdot 10^{-2}$	$2.11 \cdot 10^{-2}$	$2.10 \cdot 10^{-2}$	$2.10 \cdot 10^{-2}$	$2.10 \cdot 10^{-2}$
5%	Popova	$2.73 \cdot 10^{-1}$	1.23	8.48	77.8	771.8
	Neumaier	$1.13 \cdot 10^{-1}$	$1.11 \cdot 10^{-1}$	$1.11 \cdot 10^{-1}$	$1.10 \cdot 10^{-1}$	$1.10 \cdot 10^{-1}$
	Neumaier ver.	$1.13 \cdot 10^{-1}$	$1.11 \cdot 10^{-1}$	$1.11 \cdot 10^{-1}$	$1.10 \cdot 10^{-1}$	$1.10 \cdot 10^{-1}$

Tabelle 4.35.: Relative Breite der berechneten Einschließung für Stabtragwerk-Berechnung

Zunächst zeigt sich, dass der Neumaier/Pownuk-Löser in der nicht verifizierenden Variante im allgemeinen deutlich schneller ist als der Löser nach Popova. Für steigende Dimensionen nimmt der Unterschied darüber hinaus deutlich zu (dies wird in späteren Tests noch genauer zu sehen sein). Die verifizierende Version des Lösers nach Neumaier ist für kleine Dimensionen hingegen der langsamste. Für größere Dimensionen hingegen nähert sich die Laufzeit dem Löser nach Popova an und überholt diesen ab einem gewissen Punkt (auch dazu folgen genauere Tests später). Die kleinen Laufzeitunterschiede bei verschiedenen Toleranzen erklären sich jeweils durch eine unterschiedliche Anzahl an nötigen Iterationsschritten.

Die gemessenen relativen Breiten der Einschließungen zeigen, dass der Löser nach Neumaier/Pownuk gegenüber dem Löser nach Popova deutliche Vorteile hat. Diese nehmen allgemein mit steigender Dimension zu. Insbesondere für größere Toleranzen kann

die relative Breite des Lösers nach Popova für das betrachtete Problem deutlich zunehmen und auch für kleinere Dimensionen schon nicht mehr zu einer verifizierten Lösung führen. Die verifizierende Version des Lösers nach Neumaier/Pownuk hingegen liefert nahezu identische Ergebnisse zur nicht verifizierenden Version. Wie bereits erwähnt, werden Rundungsfehler im Allgemeinen durch Überschätzungseffekte überlagert, solange nicht mit sehr dünnen Intervallen gerechnet wird. Allgemein ist der Löser nach Neumaier/-Pownuk für diese konkrete Problemstellung also ein gutes Werkzeug, welches auch in der schnellen Version ohne Verifizierung nützliche Ergebnisse liefert.

Im Folgenden soll zum Abschluss dieses Kapitels noch ein Test mit Zufallssystemen durchgeführt werden, welcher insbesondere die Laufzeit und den Speed-Up des Lösers nach Neumaier/Pownuk genauer untersucht.

Zunächst wird dabei ein System der Form $(K + B\boldsymbol{D}A)\boldsymbol{x} = a + F\boldsymbol{b}$ betrachtet. Dabei wird für K und F die Einheitsmatrix verwendet, B und A sind Bandmatrizen mit Bandbreite 2 und Zufallseinträgen und alle Elemente von a haben den Wert 1. Die Matrix $\boldsymbol{D}$ ist eine Diagonalmatrix mit $\boldsymbol{d}_{ii} = i$ und $\boldsymbol{b}$ ist ebenfalls ein Vektor mit allen Elementen gleich 1. Die Elemente von $\boldsymbol{D}$ und $\boldsymbol{b}$ werden außerdem um den Faktor 10^{-5} aufgebläht. Dieses System wird dann sowohl mit dem Neumaier/Pownuk-Löser in verifizierender und nicht verifizierender Version als auch mit dem Popova-Löser gelöst, für welchen das System passend umgewandelt wird. Für alle Löser werden, soweit möglich, dünn besetzte Matrizen verwendet. Tabelle 4.36 zeigt die Zeiten für die Dimensionen $n = 100, 500, 1000$ und einen bzw. acht Threads. Abbildung 4.13 zeigt den Speed-Up für $n = 500$, Abbildung 4.14 den Speed-Up für $n = 1000$.

Threads	**Löser**	$n = 100$	$n = 500$	$n = 1000$
1	Popova	0.434	4.41	34.6
	Neumaier	0.0017	0.048	0.236
	Neumaier ver.	0.0122	0.556	3.22
8	Popova	0.0086	0.790	6.80
	Neumaier	0.0019	0.040	0.187
	Neumaier ver.	0.012	0.311	1.59

Tabelle 4.36.: Zeit in s für parametrisches Zufallssystem

Für dieses Testsystem sind in der Regel sowohl der Neumaier/Pownuk-Löser als auch der verifizierende Neumaier/Pownuk-Löser deutlich schneller als der Popova-Löser. Durch die Verifizierung verlangsamt sich der Neumaier/Pownuk-Löser in etwa um den Faktor 8. Die Ergebnisqualität für diese Tests ist hier für alle Löser nahezu gleich, mit leichten Vorteilen für die beiden Neumaier/Pownuk-Löser.

Für $n = 500$ sind die Speed-Ups allgemein aufgrund des niedrigen Gesamtaufwands recht gering. Der nicht-verifizierende Neumaier/Pownuk-Löser profitiert besonders wenig von der Parallelisierung, da er bei Verwendung dünn besetzter Matrizen einzig für die Invertierung der Mittelpunktsmatrix die zur Verfügung stehenden Threads ausnutzt. Beim

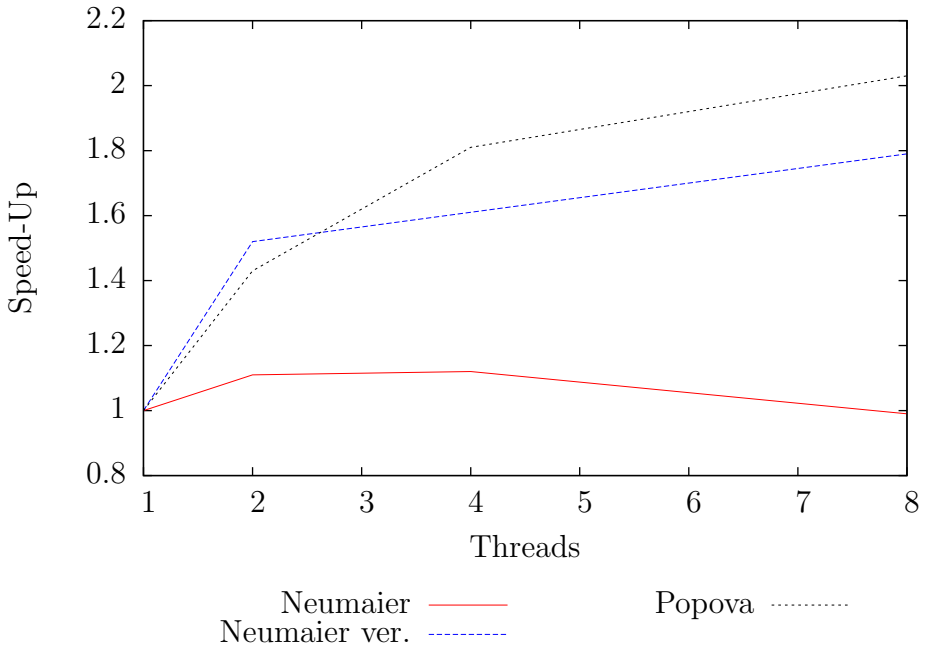


Abbildung 4.13.: Speed-Up der verschiedenen Löser für parametrisches Zufallssystem der Dimension $n = 500$

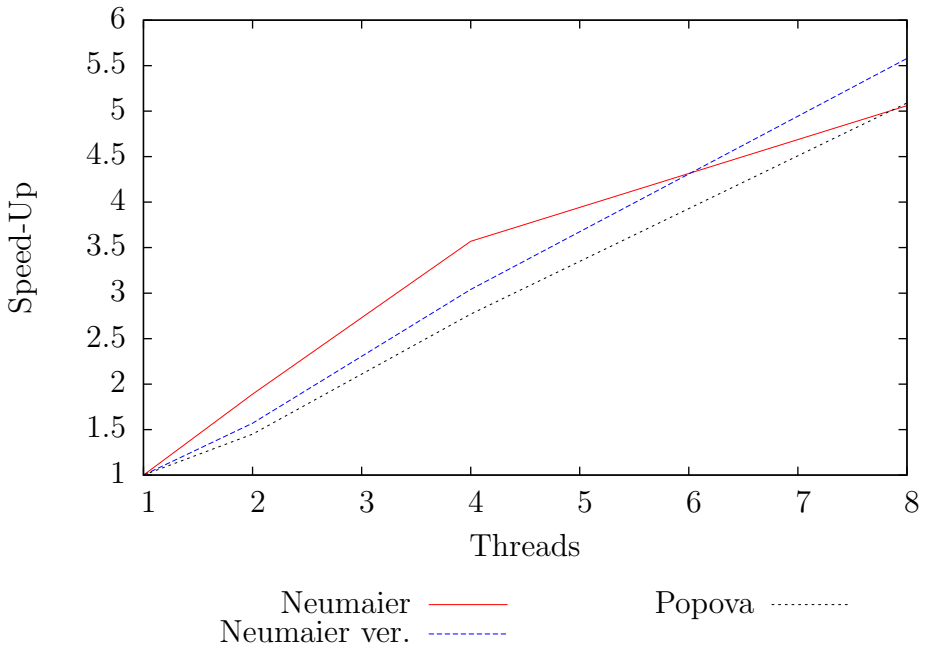


Abbildung 4.14.: Speed-Up der verschiedenen Löser für parametrisches Zufallssystem der Dimension $n = 1000$

verifizierenden Löser wird ebenfalls die Berechnung der Inversen parallelisiert, da diese aber verifiziert erfolgt profitiert hier ein wesentlich stärkerer Anteil des Gesamtaufwands von der Parallelisierung. Der Popova-Löser profitiert wie schon zuvor gesehen recht gut von den zur Verfügung stehenden Threads, auch bei niedrigem Gesamtaufwand.

Für das System mit Dimension $n = 1000$ zeigen alle Löser einen deutlich besseren Speed-Up. Auch der einfache Neumaier/Pownuk-Löser profitiert hier stärker, da bei dieser Dimension der Gesamtaufwand noch mehr von der Berechnung der Mittelpunktsinversen dominiert wird. Auch die beiden anderen Löser zeigen durch den gestiegenen Gesamtaufwand einen absolut besseren Speed-Up, welcher auch mehr von einer steigenden Thread-Anzahl profitiert.

Insgesamt zeigen die Tests, dass beide parametrischen Löser ein wichtiges Werkzeug für verschiedene Aufgabenstellungen sein können. Der Neumaier/Pownuk-Löser kann bei Systemen, welche sich in die benötigte Form umwandeln lassen, vor allem bei Systemen, welche aus der Finite-Elemente-Methode resultieren, wesentlich bessere Ergebnisse erzielen und ist dabei noch deutlich schneller. Der Popova-Löser bietet dafür eine höhere Flexibilität und ist allgemein für parametrischen Systeme anwendbar, bei welchen die parametrische Systemmatrix starke Regularität aufweist. Für nicht zu breite Parameterintervalle bzw. spezifische Parameter-Abhängigkeiten (z.B. spaltenweise Abhängigkeiten) erzielt auch dieser Löser im Allgemeinen sehr gute Einschließungen.

5. Verifizierende Löser für dünn besetzte lineare Gleichungssysteme

In Kapitel 4.1 wurden Methoden zur Lösung allgemeiner linearer Gleichungssysteme sowie deren Implementierung vorgestellt. Dabei wurde davon ausgegangen, dass es sich bei der Systemmatrix um eine dicht besetzte Matrix handelt. In vielen Anwendungen treten allerdings Systeme mit allgemein dünn besetzten und insbesondere mit Band-Systemmatrizen auf, z.B. bei der Diskretisierung partieller Differentialgleichungen. Zwar lassen sich solche Systeme prinzipiell auch mit den in Kapitel 4 beschriebenen Methoden lösen, allerdings wird dabei weder im Hinblick auf den Rechenaufwand noch auf den Speicherbedarf die Besetztheitsstruktur der Systemmatrix ausgenutzt.

In diesem Kapitel werden daher spezielle Methoden und optimierte Implementierungen zur verifizierten Lösung dünn besetzter Gleichungssysteme vorgestellt, welche im Allgemeinen wesentlich effektiver sind. Hierbei handelt es sich zwar um bereits bekannte Methoden, die hier vorgestellte Implementierung verbessert gegenüber bestehenden Lösern aber zum einen die Ergebnisqualität oftmals deutlich und bietet zum anderen in vielen Fällen signifikante Geschwindigkeitsvorteile.

Es werden Löser mit zwei grundsätzlich verschiedenen Ansätzen betrachtet: Der erste Löser verwendet eine Modifizierung der auch für dicht besetzte Systeme genutzten Methode basierend auf dem Krawczyk-Operator. Hierbei tritt das Problem auf, dass Dreieckssysteme mit intervallwertiger rechter Seite gelöst werden müssen. Wie noch zu sehen sein wird, sind dazu in der Regel spezielle Algorithmen nötig, da eine einfache intervallmäßige Vorwärts- bzw. Rückwärtssubstitution oft zu starken Überschätzungen führt.

Ein zweiter Ansatz verwendet eine Methode nach Rump, bei der eine verifizierte Unterschranke für den kleinsten Singulärwert der Systemmatrix berechnet wird. Mit dieser kann dann eine normweise Abschätzung des Fehlers einer zuvor berechneten Näherungslösung vorgenommen werden, womit sich eine verifizierte Einschließung der exakten Lösungsmenge bestimmen lässt.

Der Schwerpunkt in diesem Kapitel liegt insbesondere auf einer effizienten Implementierung der vorgestellten Methoden. Hierbei wird wieder von den in Kapitel 3 beschriebenen Erweiterungen der C-XSC Bibliothek Gebrauch gemacht, vor allem von den Datentypen für dünn besetzte Vektoren und Matrizen.

Die beiden verwendeten Methoden zur verifizierten Lösung dünn besetzter (Intervall-) Gleichungssysteme haben spezifische Stärken und Schwächen, auf die in diesem Kapitel sowohl theoretisch als auch im Zuge von ausführlichen Tests eingegangen wird. Bei den Tests werden sowohl Laufzeit als auch numerische Qualität der verschiedenen Verfahren aufgezeigt und verglichen und es werden, soweit möglich, Vergleiche mit anderen Soft-

warepaketen durchgeführt. Auch werden einige schon in Kapitel 4 durchgeführte Tests mit den dünn besetzten Lösern wiederholt, um die Vorteile der Verwendung von speziell angepassten selbstverifizierenden Lösern für dünn besetzte Gleichungssysteme zu verdeutlichen.

5.1. Theoretische Grundlagen

In diesem Abschnitt sollen zunächst die theoretischen Grundlagen für die verifizierte Lösung dünn besetzter linearer (Intervall-) Gleichungssysteme dargelegt werden. Die beiden grundsätzlichen Ansätze (Krawczyk-Operator bzw. verifizierte normweise Abschätzung) werden dabei jeweils in einem eigenen Abschnitt behandelt. Weiterhin wird auf die Vor- und Nachteile der Lösungsmethoden genauer eingegangen und die Klasse der jeweils lösbaren Systeme wird diskutiert.

Im Folgenden wird zumeist nur der reelle Fall (also die verifizierte Lösung reeller Punkt- oder Intervallsysteme) betrachtet. Die beschriebenen Methoden lassen sich in der Regel direkt auf komplexe Systeme übertragen. Sollte dies nicht der Fall sein bzw. sollten besondere Anpassungen nötig sein, so wird darauf gesondert eingegangen.

5.1.1. Lösungsansatz basierend auf dem Krawczyk-Operator

Für die verifizierte Lösung dicht besetzter Gleichungssysteme wurde bereits in Abschnitt 4.1.1 der Krawczyk-Operator und ein auf diesem basierendes Verfahren zur Einschließung des Defekts $x - \tilde{x}$ einer zuvor berechneten Näherungslösung $\tilde{x}$ eines linearen Gleichungssystems $Ax = b$, mit $A \in \mathbb{R}^{n \times n}$, $b, x \in \mathbb{R}^n$ vorgestellt. Mit einer Näherungsinversen $R \approx A^{-1}$ wurde dabei die Iteration

$$\boldsymbol{x}^{(k+1)} = R(b - A\tilde{x}) + (I - RA)\boldsymbol{x}^{(k)} \tag{5.1}$$

durchgeführt. Satz 4.1 hat gezeigt, dass falls bei dieser Iteration für ein $k \in \mathbb{N}$

$$\hat{\boldsymbol{x}} := \boldsymbol{x}^{(k+1)} \subseteq \operatorname{int}(\boldsymbol{x}^{(k)})$$

gilt, also eine Einschließung einer Iterierten ins Innere der vorhergehenden Iterierten erreicht wurde, folgt, dass A und R regulär sind und für die Lösung x des Gleichungssystems $x \in \hat{\boldsymbol{x}} + \tilde{x}$ gilt.

Für Intervallsysteme wurde das gleiche Vorgehen verwendet, wobei die Näherungsinverse und die Näherungslösung für das zugehörige Mittelpunktsystem bestimmt wurden. Entscheidend für den Erfolg der Methode ist, dass der Spektralradius von $|I - R\boldsymbol{A}|$ kleiner eins ist. Somit sind in der Regel Punktsysteme mit Konditionszahlen bis etwa 10^{16} bzw. Intervallsysteme mit entsprechend konditionierter Mittelpunktsmatrix und hinreichend kleinem Radius lösbar. Dabei erhält man in der Regel eine elementweise enge Einschließung des Defekts.

Beim Übergang auf dünn besetzte Matrizen ergibt sich nun das Problem, dass die Inverse einer dünn besetzten Matrix im Allgemeinen dicht besetzt ist. Die Näherungsinverse R wird also in den meisten Fällen eine dicht besetzte Matrix sein, wodurch starke

Nachteile hinsichtlich Speicher- und Rechenzeitbedarf der Methode entstehen, welche eine Lösung großer dünn besetzter Systeme entweder unmöglich oder gemäß Abschnitt 4.1.3 nur auf einem Cluster mit ausreichenden Ressourcen durchführbar machen würde. Zwar existieren Algorithmen, die eine dünn besetzte Näherungsinverse bestimmen können [34,51], welche die gleiche Besetztheitsstruktur wie A bzw. eine vorgegebene Besetztheit aufweist. Allerdings ist die Qualität solcher dünn besetzter Näherungsinversen in der Regel nicht ausreichend, um die Bedingung $\rho(I - RA) < 1$ zu erfüllen.

Aus diesem Grund wird (5.1) so modifiziert, dass statt einer Näherungsinversen nun eine näherungsweise LU-Zerlegung $A \approx LU$ verwendet wird. Die Matrizen L und U können mit angepassten Algorithmen als dünn besetzte Matrizen berechnet werden. Die Iteration wird dann gemäß

$$\boldsymbol{x}^{(k+1)} = U^{-1}L^{-1}(b - A\tilde{x} + (LU - A)\boldsymbol{x}^{(k)}) \tag{5.2}$$

umgestellt. Die Aussagen von Satz 4.1 bleiben dabei bestehen. Entscheidend für die Konvergenz ist in dieser Variante, dass der Spektralradius von $U^{-1}L^{-1}(LU - A)$ kleiner als eins ist. Diese Methode wird u.a. auch bei Cordes [30,31] verwendet.

Die Matrizen L^{-1} und U^{-1} in (5.2) werden nicht direkt berechnet, da es sich auch bei diesen Inversen im Allgemeinen um dicht besetzte (Dreiecks-)Matrizen handeln würde. Stattdessen müssen die entsprechenden Dreieckssysteme gelöst werden. Hierbei tritt aber der entscheidende Nachteil dieser Methode zu Tage: Da es sich bei der rechten Seite des jeweiligen Dreieckssystems um einen Intervallvektor handelt, führt einfache Vorwärts- bzw. Rückwärtssubstitution in vielen Fällen zu deutlichen Überschätzungen. In der Folge wird nun genauer auf diesen Umstand und verschiedene Lösungsmöglichkeiten für die hier auftretenden Dreieckssysteme eingegangen.

Lösungsmöglichkeiten für die auftretenden Dreieckssysteme

Im Folgenden werden nur untere Dreieckssysteme diskutiert, d.h. das Ziel ist, eine möglichst enge Einschließung der Intervallhülle $\boldsymbol{x} \in \mathbb{IR}^n$ der exakten Lösungsmenge des linearen Gleichungssystems

$$L\boldsymbol{x} = \boldsymbol{b} \tag{5.3}$$

zu finden, mit einer unteren Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$ und rechter Seite $\boldsymbol{b} \in \mathbb{IR}^n$. Die Anpassung der beschriebenen Methoden auf obere Dreieckssysteme ist jeweils direkt möglich und wird nicht gesondert behandelt.

Der einfachste Ansatz ist zunächst, gewöhnliche Vorwärtssubstitution zu verwenden und die dabei auftretenden Operationen durch Intervalloperationen zu ersetzen. Da es sich bei L um eine dünn besetzte Matrix handelt, welche auf der Maschine im spaltenorientierten *Compressed Column Storage* Format (siehe Abschnitt 3.3) gespeichert wird, wird eine spaltenorientierte Variante der Vorwärtssubstitution verwendet. Algorithmus 5.1 zeigt das entsprechende Vorgehen. Die Notation folgt dabei Abschnitt 3.3 und es wird von einer Null-basierten Indizierung ausgegangen.

Bei diesem Vorgehen können allerdings im Allgemeinen beliebig große Überschätzungen auftreten, wie Beispiel 5.1 nach Neumaier [102] zeigt.

```
Eingabe : Eine dünn besetzte untere Dreiecksmatrix L im CCS-Format und die
          rechte Seite b
Ausgabe : Lösungsvektor x
x = b
for j=0 to n-1 do
    x[j] /= L[j][j]
    for k=L.p[j]+1 to L.p[j+1]-1 do
        x[L.ind[k]] -= L.x[k] * x[j]
```

Algorithmus 5.1: Lösung eines unteren Dreieckssystems mit Vorwärtssubstitution

Beispiel 5.1. *Betrachtet wird ein System mit der Systemmatrix*

$$L = \begin{pmatrix} 1 & & & & & & \\ 1 & 1 & & & & & \\ 1 & 1 & 1 & & & & \\ & & & \ddots & & & \\ & & & & 1 & 1 & 1 \end{pmatrix}$$

und der rechten Seite

$$\boldsymbol{b}_i = [-1, 1].$$

Intervall-Vorwärtssubstitution liefert nun

$$\begin{aligned} \boldsymbol{x}_1 &= [-1,1] \\ \boldsymbol{x}_2 &= [-1,1] - \boldsymbol{x}_1 = 2[-1,1] \\ \boldsymbol{x}_3 &= [-1,1] - \boldsymbol{x}_1 - \boldsymbol{x}_2 = 4[-1,1] \\ \boldsymbol{x}_4 &= [-1,1] - \boldsymbol{x}_2 - \boldsymbol{x}_3 = 7[-1,1] \\ \boldsymbol{x}_5 &= [-1,1] - \boldsymbol{x}_3 - \boldsymbol{x}_4 = 12[-1,1] \\ &\dots \end{aligned}$$

Der Durchmesser der berechneten Einschließungen wächst hier exponentiell. Die tatsächliche Lösungsmenge lässt sich berechnen über die Formel $\boldsymbol{x}_i = (i - \lfloor \frac{i}{3} \rfloor) \cdot [-1, 1]$ *[126].*

Ursache für diese Überschätzung der tatsächlichen Lösungsmenge ist das Abhängigkeitsproblem. Dies lässt sich erkennen, wenn man allgemein die Formel zur Berechnung des i-ten Elementes von $\boldsymbol{x}$ betrachtet:

$$\boldsymbol{x}_i = \frac{\boldsymbol{b}_i}{l_{ii}} - \sum_{k=1}^{i-1} l_{ik} \boldsymbol{x}_k.$$

Durch Einsetzen der Berechnungsformeln für die $\boldsymbol{x}_k$ in der Summe erhält man eine Formel, in der jedes Element $\boldsymbol{b}_j$ der rechten Seite mit $j < i$ genau $i - j$ mal auftaucht. Da jedes Vorkommen bei der Berechnung als unabhängiges Intervall betrachtet werden muss

(andernfalls müsste man die Inverse von L berechnen), kommt es so zu Überschätzungseffekten, sofern die verschiedenen Vorkommen von $\boldsymbol{b}_j$ nicht während der Berechnung stets mit Koeffizienten mit gleichem Vorzeichen multipliziert werden.

Allgemein kann die Vorwärtssubstitution auch mit Hilfe der Ostrowski-Matrix $\langle L \rangle$ formuliert werden. Bei symmetrischer rechter Seite $\boldsymbol{b}$ (der allgemeine Fall lässt sich leicht auf diesen übertragen) gilt [102]

$$L^{-1}\boldsymbol{b} = [-\langle L \rangle^{-1} \inf(\boldsymbol{b}), \langle L \rangle^{-1} \sup(\boldsymbol{b})],$$

wobei hier (und im weiteren Verlauf) $L^{-1}\boldsymbol{b}$ mittels Vorwärtssubstitution bestimmt werde. Die exakte Lösung hingegen lässt sich über

$$L^{-1}\boldsymbol{b} = [-|L^{-1}|\inf(\boldsymbol{b}), |L^{-1}|\sup(\boldsymbol{b})]$$

berechnen. Es gilt $|L^{-1}| \leq \langle L \rangle^{-1}$, d.h. bei Gleichheit lässt sich das Dreieckssystem mittels Vorwärtssubstitution ohne Überschätzung (abgesehen von Rundungsfehlern auf der Maschine) lösen. Der Fall $|L^{-1}| = \langle L \rangle^{-1}$ tritt insbesondere für folgende Matrixklassen auf:

- L ist eine M-Matrix.
- L ist eine Diagonalmatrix.
- L besitzt nur Einträge auf der Diagonalen und auf einer Nebendiagonalen.

Falls die Systemmatrix A eine M-Matrix ist, so sind auch die Faktoren L und U aus der LU-Zerlegung M-Matrizen. Einfache Vorwärtssubstitution ist also insbesondere für Systeme mit M-Matrizen erfolgreich. Handelt es sich bei A um eine Tridiagonalmatrix und wird die LU-Zerlegung ohne Pivotisierung ausgeführt, so erhält man Dreiecksmatrizen mit nur einer Nebendiagonalen. Auch solche Systeme sind über einen Krawczyk-Operator mit einfacher Vorwärts- und Rückwärtssubstitution lösbar.

Im Komplexen ist entscheidend, ob mit komplexen Rechteckintervallen oder mit Kreisscheiben gerechnet wird. Da man bei Rechteckintervallen Real- und Imaginärteil getrennt als reelle Intervalle im Infimum-Supremum-Format betrachtet, werden hier wieder zusätzliche Abhängigkeiten eingeführt, so dass einfache Vorwärtssubstitution noch deutlich seltener zum Erfolg führt. Bei Verwendung von komplexen Kreisscheiben hingegen lassen sich obige Aussagen direkt übertragen, da das komplexe Intervall hier in den Berechnungen stets als Ganzes betrachtet wird und somit keine zusätzlichen Abhängigkeiten auftreten.

Im Allgemeinen ist die naive Vorgehensweise also nur in wenigen Fällen erfolgreich und es stellt sich die Frage nach Alternativen. Eine Möglichkeit ist, eine Einschließung der Inversen der Dreiecksmatrix L zu berechnen. Im Allgemeinen wird diese zwar eine dicht besetzte Dreiecksmatrix sein, falls aber aufgrund der Struktur von L bekannt ist oder vermutet werden kann, dass es sich auch bei L^{-1} um eine dünn besetzte Matrix handelt, so wäre eine solche Lösungsmethode von Vorteil.

Mit Algorithmus 5.2 kann das Produkt $L^{-1}\boldsymbol{b}$ berechnet werden, ohne dass L^{-1} komplett im Speicher gehalten werden muss.

Eingabe : Untere Dreiecksmatrix L und symmetrische rechte Seite $\boldsymbol{b}$
Ausgabe : $x = L^{-1}\boldsymbol{b}$
$R := I$
for *j=1* **to** n **do**
  $R_{j\bullet} = R_{j\bullet}$ / l_{jj}
  $x_j = |R_{j\bullet}|\boldsymbol{b}$
  for *i=j+1* **to** n **do**
    $R_{i\bullet} = R_{i\bullet} - l_{ij}R_{j\bullet}$

Algorithmus 5.2: Algorithmus Berechnung von $L^{-1}\boldsymbol{b}$

Nach dem j-ten Durchlauf der äußeren for-Schleife und der Ausführung der ersten Anweisung innerhalb der for-Schleife entspricht die j-te Zeile von R der j-ten Zeile von L^{-1}. Bei einer Implementierung sollte dabei R dünn besetzt gespeichert werden. Die j-te Zeile von R kann am Ende jedes Durchlaufs der äußeren for-Schleife gelöscht werden, wodurch niemals die komplette Inverse gespeichert werden muss.

Weiterhin sollten in einer Implementierung sämtliche Operationen so ausgeführt werden, dass die dünne Besetztheit ausgenutzt wird. Insbesondere benötigt die innere for-Schleife so nicht $n-j$, sondern nur $\mathrm{nnz}(L_{\bullet j})-1$ Durchläufe. Da die einzelnen Durchläufe dieser Schleife voneinander unabhängig sind, lässt sie sich direkt parallelisieren. Allerdings müssen in einer Implementierung mögliche Rundungsfehler beachtet werden (es muss also jeweils eine Ober- und Unterschranke von $R_{j\bullet}$ berechnet werden), wodurch der Aufwand wieder etwas ansteigt. Durch die Verwendung einer symmetrischen rechten Seite und des Betrages $|R_{j\bullet}|$ bei der Berechnung von x_j lässt sich die am Rechner bestimmte Einschließung mit weniger Aufwand berücksichtigen als es bei einer komplett intervallmäßigen Auswertung der Fall wäre. Darauf wird genauer in Abschnitt 5.2.2 eingegangen.

Vorteil der Methode ist, dass Überschätzungseffekte durch das Abhängigkeitsproblem vermieden werden. Auf der Maschine treten durch Rundungsfehler trotzdem Überschätzungen auf. Allerdings sollte diese Methode für die Konditionszahlen, für welche Rumps Algorithmus basierend auf dem Krawczyk-Operator zum Erfolg führt, in der Regel zu einer erfolgreichen Einschließung ins Innere bei Iteration (5.2) gelangen.

Nachteil ist, dass L^{-1} in den meisten Fällen dicht (oder zumindest deutlich dichter als L) besetzt ist und somit enormer Rechenaufwand entstehen kann, auch wenn der Speicherbedarf in vielen Fällen bei angepasster Implementierung klein genug für eine erfolgreiche Durchführung ist. Insgesamt ist diese Methode somit in der Praxis in der Regel (außer zu Testzwecken) untauglich, es sei denn es ist vorab bekannt, dass L^{-1} dünn besetzt ist.

Eine weitere Alternative zur Lösung der hier auftretenden Dreieckssysteme lässt sich für den Fall, dass es sich bei L um eine Bandmatrix mit kleiner Bandbreite handelt, formulieren. Falls die ursprüngliche Systemmatrix A eine Bandmatrix ist, so kann durch Verzicht auf Pivotisierung eine LU-Zerlegung berechnet werden, bei der L und U Dreiecksmatrizen mit der gleichen Bandbreite wie A sind.

Die im Folgenden beschriebene Methode geht auf Lohner zurück [83]. Es wird davon ausgegangen, dass die (untere Dreiecksmatrix) L eine Bandmatrix mit unterer Bandbreite m ist. Die Grundidee ist, die durch das Abhängigkeitsproblem auftretenden Überschätzungseffekte zu reduzieren, indem statt mit Standard-Intervallen mit sogenannten Parallelepipeden gearbeitet wird:

Definition 5.1. *Es sei $B \in \mathbb{R}^{n\times n}$ regulär, $\boldsymbol{z} \in \mathbb{IR}^n$ und $y \in \mathbb{R}^n$. Dann wird*

$$P := P(B, \boldsymbol{z}, y) := \{Bz + y \mid z \in \boldsymbol{z}\}$$

als Parallelepiped bezeichnet. Weiterhin wird die Matrix B als Basismatrix dieses Parallelepipeds bezeichnet.

Eine Teilmenge des $\mathbb{R}^n$ wird hier also als (nicht ausgewertetes) Produkt einer Basismatrix und eines Intervallvektors dargestellt. Die Ränder von P sind dabei parallel zu den Spaltenvektoren von B. Der Vektor y stellt eine einfache Verschiebung dar und kann auch direkt zu $\boldsymbol{z}$ addiert werden, wodurch man die einfachere Form

$$P := P(B, \boldsymbol{z}) := \{Bz \mid z \in \boldsymbol{z}\}$$

erhält, welche im Folgenden benutzt wird.

Bei der Verwendung von Parallelepipeden ist entscheidend, dass das Produkt von P mit einer Matrix $A \in \mathbb{R}^{n\times n}$ gemäß

$$A \cdot P(B, \boldsymbol{z}) := \{ABz \mid z \in \boldsymbol{z}\} = P(AB, \boldsymbol{z})$$

vorgenommen wird. Hierbei entsteht also keinerlei Überschätzung und es können beliebig viele entsprechende Multiplikationen exakt durchgeführt werden. Nur bei einer Abbildung auf karthesische Koordinaten, also bei der Bestimmung einer Intervallhülle von $P(B, \boldsymbol{z})$ mittels der Auswertung von $B\boldsymbol{z}$, entsteht eine Überschätzung.

In Satz 5.1 wird die Vorwärtssubstitution nun so umgeschrieben, dass statt Standardintervallen Parallelepipede verwendet werden.

Satz 5.1. *Gegeben sei ein unteres Dreieckssystem mit der Systemmatrix $L \in \mathbb{R}^{n\times n}$, L Bandmatrix mit unterer Bandbreite m, sowie rechter Seite $\boldsymbol{b} \in \mathbb{IR}^n$. Es bezeichne $\hat{\boldsymbol{x}} \in \mathbb{IR}^n$ eine Einschließung der Intervallhülle $\boldsymbol{x}$ der exakten Lösungsmenge. Weiterhin seien bereits Einschließungen $\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_m$ gegeben. Für $i = 0, \ldots, n-m-1$ und $k := i+m+1$ sei*

$$\boldsymbol{d}^{(i)} := (0, \ldots, \boldsymbol{b}_k / l_{k,k})^T \in \mathbb{IR}^m$$

und

$$L^{(i)} = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ \frac{-l_{k,i+1}}{l_{k,k}} & \frac{-l_{k,i+2}}{l_{k,k}} & \ldots & \frac{-l_{k,i+m}}{l_{k,k}} \end{pmatrix} \in \mathbb{R}^{m\times m}.$$

Eine Einschließung $\hat{\boldsymbol{x}}$ der Lösung des Systems über eine Vorwärtssubstitution mit Parallelepipeden statt Standardintervallen lässt sich dann mit $B^{(0)} = I$ und $\boldsymbol{z}^{(0)} = (\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_m)^T$ über die Iteration

$$\boldsymbol{z}^{(i+1)} = ((B^{(i+1)})^{-1} L^{(i)} B^{(i)}) \boldsymbol{z}^{(i)} + (B^{(i+1)})^{-1} \boldsymbol{d}^{(i)} \tag{5.4}$$

berechnen, wobei die jeweiligen Basismatrizen $B^{(i)} \in \mathbb{R}^{n \times n}$ beliebige reguläre Matrizen sind. Weiterhin können die jeweiligen Iterierten über

$$\boldsymbol{y}^{(i+1)} = B^{(i+1)} \boldsymbol{z}^{(i+1)} \tag{5.5}$$

auf Standardintervalle abgebildet werden. Dann gilt im (i+1)-ten Iterationsschritt

$$\hat{\boldsymbol{x}}_k := \boldsymbol{y}_m^{(i+1)} \supseteq \boldsymbol{x}_k.$$

Nach $n - m$ Schritten ist also eine Einschließung $\hat{\boldsymbol{x}} \supseteq \boldsymbol{x}$ bestimmt.

Beweis: Zunächst wird die einfache Vorwärtssubstitution mit Standardintervallen umformuliert. Mit

$$\boldsymbol{y}^{(i)} := (\hat{\boldsymbol{x}}_{i+1}, \ldots, \hat{\boldsymbol{x}}_{i+m})^T \in \mathbb{IR}^m$$

entspricht

$$\boldsymbol{y}^{(i+1)} = L^{(i)} \boldsymbol{y}^{(i)} + \boldsymbol{d}^{(i)} \tag{5.6}$$

mit $i = 0, \ldots, n - m - 1$ offensichtlich der einfachen Vorwärtssubstitution, wenn im $i + 1$-ten Schritt

$$\hat{\boldsymbol{x}}_{m+i+1} = \boldsymbol{y}_m^{(i+1)}$$

gesetzt wird. Nach $n - m$ Schritten erhält man also eine Einschließung $\hat{\boldsymbol{x}} \supseteq \boldsymbol{x}$.

Nun wird statt der Iterierten $\boldsymbol{y}^{(i)}$ das Parallelepiped $B^{(i)} \boldsymbol{z}^{(i)}$ eingesetzt. Gleichung (5.6) ändert sich damit zu

$$B^{(i+1)} \boldsymbol{z}^{(i+1)} = (L^{(i)} B^{(i)}) \boldsymbol{z}^{(i)} + \boldsymbol{d}^{(i)}. \tag{5.7}$$

Da für die Basismatrizen $B^{(i)}$ beliebige (reguläre) $m \times m$ Matrizen gewählt werden können, muss in jedem Iterationsschritt also nur $\boldsymbol{z}^{(i+1)}$ bestimmt werden. Da $B^{(i)}$ als regulär vorausgesetzt ist, kann (5.7) zu

$$\boldsymbol{z}^{(i+1)} = ((B^{(i+1)})^{-1} L^{(i)} B^{(i)}) \boldsymbol{z}^{(i)} + (B^{(i+1)})^{-1} \boldsymbol{d}^{(i)} \tag{5.8}$$

umgestellt werden.

Nach Konstruktion können die Iterierten $\boldsymbol{z}^{(i)}$ über

$$\hat{\boldsymbol{y}}^{(i)} = B^{(i)} \boldsymbol{z}^{(i)} \tag{5.9}$$

auf entsprechende Intervallboxen abgebildet werden. Für $i = 0$ erhält man so $\hat{\boldsymbol{y}}^{(0)} = \boldsymbol{y}^{(0)}$ (da $B^{(0)} = I$ und $\boldsymbol{z}^{(0)} = \boldsymbol{y}^{(0)}$) und somit

$$\hat{\boldsymbol{y}}^{(0)} \supseteq (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)^T.$$

Es gelte nun für ein $i \in (0, \ldots, n-m-1)$

$$\hat{\boldsymbol{y}}^{(i)} \supseteq (\boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_{m+i})^T.$$

Dann folgt für $\hat{\boldsymbol{y}}^{(i+1)}$:

$$\begin{aligned} \hat{\boldsymbol{y}}^{(i+1)} &= B^{(i+1)}\boldsymbol{z}^{(i+1)} \\ &= B^{(i+1)}(((B^{(i+1)})^{-1}L^{(i)}B^{(i)})\boldsymbol{z}^{(i)} + (B^{(i+1)})^{-1}\boldsymbol{d}^{(i)}) \\ &\supseteq (L^{(i)}B^{(i)})\boldsymbol{z}^{(i)} + \boldsymbol{d}^{(i)}. \end{aligned}$$

Da $\hat{y}^{(i)} = B^{(i)}\boldsymbol{z}^{(i)}$ gilt für alle $x \in \boldsymbol{x}$

$$(x_{i+1}, \ldots, x_{m+i})^T \in B^{(i)}\boldsymbol{z}^{(i)}$$

und somit

$$L^{(i)} \cdot (x_{i+1}, \ldots, x_{m+i})^T \in (L^{(i)}B^{(i)})\boldsymbol{z}^{(i)}.$$

Damit gilt für alle $x \in \boldsymbol{x}$ und alle $b \in \boldsymbol{b}$ (und somit alle $d^{(i)} \in \boldsymbol{d}^{(i)}$)

$$L^{(i)} \cdot (x_{i+1}, \ldots, x_{m+i})^T + d^{(i)} \in (L^{(i)}B^{(i)})\boldsymbol{z}^{(i)} + \boldsymbol{d}^{(i)}.$$

Hierbei entspricht $L^{(i)}(x_{i+1}, \ldots, x_{m+i})^T + d^{(i)}$ der Gleichung (5.6) für Punktvektoren, also einem Schritt aus der einfachen Vorwärtssubstitution. Damit folgt

$$\hat{\boldsymbol{y}}^{(i+1)} \supseteq (\boldsymbol{x}_{i+2}, \ldots, \boldsymbol{x}_{m+i+1}).$$

□

Es ist zu beachten, dass die eigentliche Iteration mit den Parallelepipeden $B^{(i)}\boldsymbol{z}^{(i)}$ durchgeführt wird. Die Abbildung auf eine Intervallbox $\boldsymbol{y}^{(i)}$ wird nur zur letztlichen Berechnung der einzelnen Elemente der Einschließung $\hat{\boldsymbol{x}}$ verwendet.

Die benötigten Startwerte $\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_m$ können entweder über normale Vorwärtssubstitution oder, um Überschätzungen schon in den Startwerten zu vermeiden, über die bereits besprochene Methode der impliziten Berechnung der Inversen berechnet werden. Auf dem Rechner sind die Inversen $(B^{(i+1)})^{-1}$ sowie die Produkte $(B^{(i+1)})^{-1}L^{(i)}B^{(i)}$ im Allgemeinen nicht exakt berechenbar. Daher muss jeweils eine Intervalleinschließung der Inversen und der Matrixprodukte berechnet werden.

Entscheidend ist nun die Wahl der Basismatrizen $B^{(i)}$. Offensichtlich hängt die Qualität der nach Satz 5.1 berechneten Einschließungen von der gewählten Basismatrix ab, da z.B. die Wahl $B^{(i)} = I$ direkt auf die Vorwärtssubstitution mit den angesprochenen Problemen führt.

Davon ausgehend, dass die rechte Seite $\boldsymbol{b}$ des Systems einen kleinen Durchmesser hat (im Rahmen des Krawczyk-Operators handelt es sich hierbei um den Fehler der Näherungslösung, welcher bei Punktsystemen oder intervallwertigen rechten Seiten mit kleinem Durchmesser auch einen entsprechend kleinen Durchmesser besitzt) erscheint es sinnvoll, $B^{(i+1)} = L^{(i)}B^{(i)}$ (bzw. auf der Maschine die entsprechende Mittelpunktsmatrix

mid ($[L^{(i)}B^{(i)}]$)) zu wählen. Die Einschließung der Inversen kann mit dem Löser für dicht besetzte Gleichungssysteme aus Abschnitt 4.1 berechnet werden.

Problematisch dabei ist, dass sich für die i-te Basismatrix dann (bei exakter Rechnung)

$$B^{(i)} = L^{(i-1)} \cdot L^{(i-2)} \cdot \ldots \cdot L^{(1)}$$

ergibt. Die Kondition von $B^{(i)}$ wird daher im Laufe der Iteration in vielen Fällen (abhängig von den Matrizen $L^{(i)}$) so stark ansteigen, dass keine verifizierte Einschließung der Inversen mehr bestimmt werden kann. Weiterhin ist die Berechnung dieser Einschließung, sofern sie möglich ist, recht aufwändig.

Die Basismatrizen sollten also so gewählt werden, dass die Kondition gering bleibt und eine Einschließung der Inversen möglichst leicht zu berechnen ist. Hierzu bietet es sich an, nur den orthogonalen Anteil des Produktes $L^{(i)}B^{(i)}$ für die Basismatrix $B^{(i+1)}$ zu wählen, welchen man über eine QR-Zerlegung berechnet. Bei exakter Berechnung würde dann $(B^{(i+1)})^{-1} = (B^{(i+1)})^T$ gelten. Auf dem Rechner ist dies durch Rundungsfehler in der Regel nicht der Fall, allerdings lässt sich für nahezu orthogonale Matrizen eine entsprechende Einschließung der Inversen mit Hilfe der Neumann-Reihe vergleichsweise günstig berechnen, wie folgender Satz zeigt [83]:

Satz 5.2. *Es sei $Q \in \mathbb{R}^{n\times n}$ eine nahezu orthogonale Matrix und $q := ||I - QQ^T||_\infty < 1$. Dann gilt*

$$Q^{-1} \in Q^T + [-\epsilon, \epsilon]E,$$

mit $E \in \mathbb{R}^{n\times n}$ und allen Elementen gleich 1 sowie $\epsilon := \frac{q}{1-q}||Q^T||_\infty$.

Beweis: Mit $A := I - QQ^T$ und der Neumannschen Reihe gilt:

$$(I - A)^{-1} = (QQ^T)^{-1} = (Q^T)^{-1}Q^{-1} = \sum_{k=0}^{\infty} A^k.$$

Mit $F = Q^T \sum_{k=1}^{\infty} A^k$ gilt also

$$Q^{-1} = Q^T + F.$$

Der Abstand F von Q^T zur exakten Inversen lässt sich mit Hilfe der geometrischen Reihe abschätzen. Es ergibt sich:

$$|f_{ij}| \leq ||F||_\infty \leq ||Q^T||_\infty \sum_{k=0}^{\infty} ||A||_\infty^k = ||Q^T||_\infty \sum_{k=0}^{\infty} q^k = ||Q^T||_\infty \frac{q}{1-q} = \epsilon.$$

□

Bei Verwendung der orthogonalen Matrix Q aus der QR Zerlegung von $L^{(i)}B^{(i)}$ für die Basismatrix $B^{(i+1)}$ kann das Vorgehen für $n = 2$ auf eingängige Weise geometrisch interpretiert werden, wie Beispiel 5.2 zeigt.

Beispiel 5.2. *In diesem Beispiel wird das Produkt einer Punktmatrix $A \in \mathbb{R}^{2\times 2}$ mit einem Intervallvektor $\boldsymbol{x} \in \mathbb{IR}^2$ betrachtet. Dies entspricht im Wesentlichen einem Iterationsschritt in (5.4) (die Addition von $\boldsymbol{d}^{(i)}$ entspricht einer einfachen Translation entlang der m-ten Achse (bei Bandbreite m) und ändert nichts Wesentliches an der Aussagekraft des Beispiels). Betrachtet wird das Produkt $A\boldsymbol{x}$ von*

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

und

$$\boldsymbol{x} = \begin{pmatrix} [1,2] \\ [1,3] \end{pmatrix}.$$

Jedes $x \in \boldsymbol{x}$ lässt sich mit $0 \leq \lambda, \mu \leq 1$, $\lambda, \mu \in \mathbb{R}$ auch in der Form

$$x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} \lambda \\ 2\mu \end{pmatrix}$$

beschreiben. Somit kann die exakte Lösungsmenge in der Form

$$\left\{ \begin{pmatrix} 2 \\ 2 \end{pmatrix} + \lambda \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \mu \begin{pmatrix} 4 \\ 2 \end{pmatrix} \mid 0 \leq \lambda, \mu \leq 1 \right\}$$

beschrieben werden. Einfache Intervallmultiplikation liefert die Intervallhülle der exakten Lösungsmenge:

$$A\boldsymbol{x} = \begin{pmatrix} [2,6] \\ [2,5] \end{pmatrix}.$$

Nun soll $A\boldsymbol{x}$ durch ein Parallelepiped mit orthogonaler Basismatrix B dargestellt werden. Für B wird die Matrix

$$Q = \frac{1}{\sqrt{5}} \begin{pmatrix} -2 & -1 \\ -1 & 2 \end{pmatrix}$$

aus der QR-Zerlegung $A^* = QR$, wobei A^* der Matrix A mit vertauschten Spalten entspricht, verwendet. Auf diese Wahl wird später noch genauer eingegangen. Der Intervallvektor $\boldsymbol{z}$ aus $B\boldsymbol{z} = A\boldsymbol{x}$ lässt sich nun über

$$\boldsymbol{z} = (B^{-1}A)\boldsymbol{x} = \frac{1}{\sqrt{5}} \begin{pmatrix} [-17,-6] \\ [2,4] \end{pmatrix}$$

bestimmen.

Abbildung 5.1 zeigt nun sowohl die exakte Lösungsmenge als auch die Intervallhülle und die Einschließung durch das Parallelepiped $P(B, \boldsymbol{z})$. Der ausgefüllte Bereich zeigt die exakte Lösungsmenge, der schraffierte Bereich die Einschließung durch $P(B, \boldsymbol{z})$.

$P(B, \boldsymbol{z})$ liefert hier eine deutlich bessere Einschließung als die einfache Intervalloperation $A\boldsymbol{x}$. Zwar wäre $B\boldsymbol{z}$, also die Intervallhülle des Parallelepipeds, eine breitere Einschließung als $A\boldsymbol{z}$, allerdings wird in der Iteration (5.4) stets mit den Parallelepipeden $P^{(i)}(B^{(i)}, \boldsymbol{z}^{(i)})$ weiter gerechnet, das heißt das Produkt $B\boldsymbol{z}$ wird nicht ausgewertet. Die

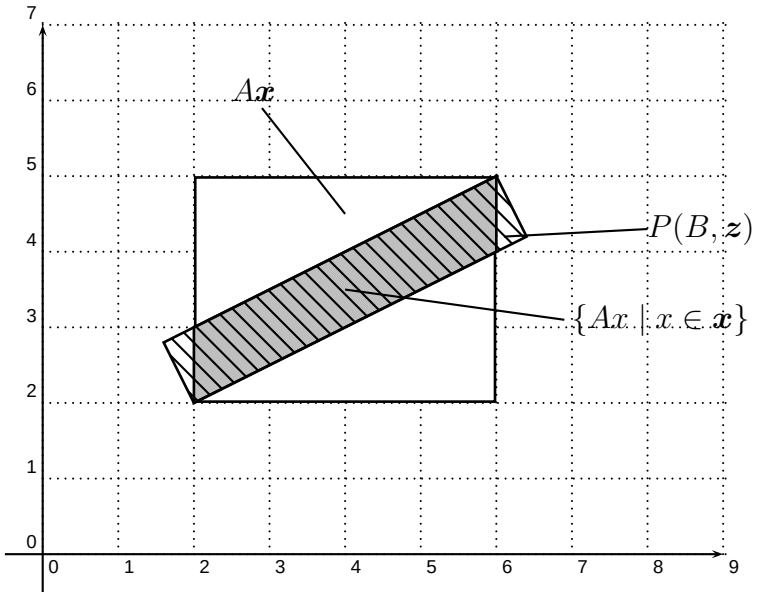


Abbildung 5.1.: Exakte Lösungsmenge und Einschließungen von $A\boldsymbol{x}$

folgenden Multiplikationen mit $L^{(i)}$ beziehen sich so also jeweils auf die enge Einschließung und können (abgesehen von Rundungsfehlern auf der Maschine) sogar exakt erfolgen. Erst durch die Überführung in $P^{(i+1)}$ und die Darstellung als Parallelepiped mit orthogonaler Basismatrix $B^{(i+1)}$ entsteht wieder eine Überschätzung.

Die Seiten jedes Parallelepipeds $P^{(i)}$ sind parallel zu den Spaltenvektoren von $B^{(i)}$. Da diese orthogonal sind, ist es für eine gute Einschließung wichtig, wie die Basismatrix $B^{(i)}$ bestimmt wird. Bei der QR-Zerlegung einer Matrix A, z.B. mittels Householder-Transformationen, wird von der Richtung der ersten Spalte von A ausgegangen und die zweite Spalte dazu orthogonalisiert. Dieses Vorgehen wird dann weitergeführt. Für das Parallelepiped bedeutet das, dass die Güte der Einschließung von der Reihenfolge der Spalten in A abhängt. Abbildung 5.2 verdeutlicht dies für das Beispiel 5.2. Die graue Fläche gibt wiederum die exakte Lösungsmenge an, die schraffierte Fläche jeweils die durch $P(B, \boldsymbol{z})$ eingeschlossene Fläche.

In der linken Abbildung wird bei der Orthogonalisierung von der längsten Seite der Lösungsmenge ausgegangen. Dies ist gleichbedeutend damit, dass bei der Orthogonalisierung von der Spalte ausgegangen wird, für welche

$$||A_{\bullet i}||_2 \cdot \operatorname{diam}(\boldsymbol{x}_i)$$

am größten ist. Im Beispiel müssen dazu die Spalten von A vertauscht werden. Die rechte Seite der Abbildung zeigt den Fall, dass bei der Orthogonalisierung von der kürzeren Seite der Lösungsmenge ausgegangen wird. Die entsprechende Einschließung überschätzt die tatsächliche Lösungsmenge deutlich stärker, in diesem Fall entspricht sie sogar der Intervallhülle des exakten Wertebereichs.

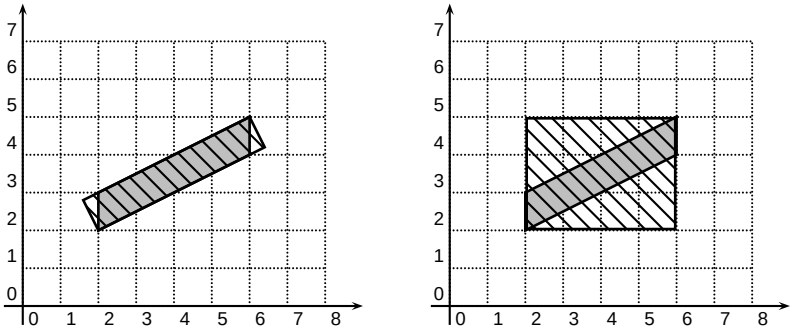


Abbildung 5.2.: Gestalt von $P(B, \boldsymbol{z})$ in Abhängigkeit der Spaltenreihenfolge von A bei der QR-Zerlegung

Bevor die jeweilige Basismatrix des Parallelepipeds über eine QR-Zerlegung bestimmt wird, müssen also erst die Spalten der betrachteten Matrix nach der Länge der Seiten der Lösungsmenge sortiert werden. Die vom Parallelepiped beschriebene Menge entspricht dann in einem gedrehten karthesischen Koordinatensystem, in welchem die erste Achse parallel zur längsten Seite der Lösungsmenge ist, der Intervallhülle der exakten Lösungsmenge. Da das Koordinatensystem in jedem Iterationsschritt erneut gedreht wird, ist bei Verwendung von Parallelepipeden für die Vorwärtssubstitution zur Lösung des betrachteten Dreieckssystems gemäß Satz 5.1 die Überschätzung in der Regel deutlich geringer als bei einfacher Vorwärtssubstitution.

Zusammenfassend ergibt sich somit eine Methode zur Lösung von Dreieckssystemen mit Punktsystemmatrix und intervallwertiger rechter Seite mittels Parallelepipeden auf dem Rechner, welche in Algorithmus 5.3 angegeben wird (die Matrizen $L^{(i)}$ entsprechen dabei der Definition in Satz 5.1).

Eingabe : Untere Dreiecksmatrix L mit Bandbreite m und rechte Seite $\boldsymbol{b}$
Ausgabe : Einschließung $\hat{\boldsymbol{x}}$ der Intervallhülle $\boldsymbol{x}$ der exakten Lösungsmenge
Bestimme $\boldsymbol{y}^{(1)} = (\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_m)^T$ mit Algorithmus 5.2
Setze $B^{(1)} = I$ und $z^{(1)} = y^{(1)}$
for *i=1* **to** *n-m* **do**
  Berechne $\boldsymbol{A}^{(i)} = [L^{(i)} B^{(i)}]$
  Berechne $M^{(i)} = \mathrm{mid}\,(\boldsymbol{A}^{(i)})$
  Sortiere Spalten von $M^{(i)}$ absteigend nach $||M_{\bullet i}||_2 \cdot \mathrm{diam}(\boldsymbol{x}_i)$
  Berechne QR-Zerlegung $QR \approx M^{(i)}$
  Setze $B^{(i+1)} = Q$
  Bestimme $[(B^{(i+1)})^{-1}]$ nach Satz 5.2
  Berechne $\boldsymbol{z}^{(i+1)} = ([(B^{(i+1)})^{-1}]\boldsymbol{A}^{(i)})\boldsymbol{z}^{(i)} + [(B^{(i+1)})^{-1}]\boldsymbol{d}^{(i)}$
  Setze $\hat{\boldsymbol{x}}_{i+m} = (B^{(i+1)}\boldsymbol{z}^{(i+1)})_m$

Algorithmus 5.3: Lösung eines unteren Dreieckssystems mit Vorwärtssubstitution für Parallelepipede

Der Aufwand des Algorithmus wird dominiert von den QR-Zerlegungen sowie den auftretenden Matrix-Matrix-Produkten. Insgesamt ergibt sich ein Aufwand von $\mathcal{O}(n \cdot 6m^3)+\mathcal{O}(m^2)$. Für größere Bandbreiten m steigt der Aufwand also stark an, die Methode ist daher nur für kleinere Bandbreiten sinnvoll, sollte dann aber durch die wesentlich geringere Überschätzung gegenüber der einfachen Vorwärtssubstitution deutlich häufiger zu einem Erfolg der Iteration (5.2) führen. Auch diese Methode lässt sich im Wesentlichen direkt auf den komplexen Fall übertragen. Als Basismatrizen der Parallelepipede werden dabei unitäre Matrizen Q aus der komplexen QR-Zerlegung von $L^{(i)}B^{(i)}$ verwendet. Die übrigen Erläuterungen und getroffenen Aussagen sind direkt übertragbar.

Möglichkeiten zur Optimierung

Bei Verwendung des modifizierten Krawczyk-Operators (5.2) sind die einzelnen Iterationsschritte im Vergleich zum dicht besetzten Fall für alle beschriebenen Methoden zur Lösung der auftretenden Dreieckssysteme (wenn auch jeweils in unterschiedlich starkem Maße) recht teuer. Es ist daher vorteilhaft, die Anzahl der Iterationsschritte und damit die Anzahl der zu lösenden Dreieckssysteme möglichst auf ein Mindestmaß zu beschränken.

Dazu kann der Verifikationsschritt folgendermaßen abgeändert werden [127]. Zunächst werden

$$\boldsymbol{t}_1 := U^{-1}L^{-1}(b - A\tilde{x})$$

und

$$\boldsymbol{t}_2 := U^{-1}L^{-1}(LU - A)\boldsymbol{t}_1$$

berechnet. Gilt dann für ein $d > 0 \in \mathbb{R}$

$$|\boldsymbol{t}_1| + (1 + d)|\boldsymbol{t}_2| < (1 + d)|\boldsymbol{t}_1|,$$

so gilt für den Defekt $\boldsymbol{x}$ der Näherungslösung $\tilde{x}$

$$\boldsymbol{x} \in (1 + d)[-|\boldsymbol{t}_1|, |\boldsymbol{t}_1|].$$

Dabei kann d gemäß

$$d := \max_i \frac{|\boldsymbol{t}_1|_i}{|\boldsymbol{t}_1|_i - |\boldsymbol{t}_2|_i}$$

berechnet werden (sofern $|\boldsymbol{t}_1| > |\boldsymbol{t}_2|$). Auf dem Rechner sind jeweils mit den bereits besprochenen Methoden Einschließungen, welche die Rundungsfehler berücksichtigen, zu berechnen.

Der Aufwand entspricht so im Wesentlichen stets zwei herkömmlichen Iterationsschritten. Die auf diese Art berechnete Einschließung ist allerdings etwas breiter.

Ein weiterer aufwändiger Teil der Berechnung mittels des modifizierten Krawczyk-Operators (5.2) ist die Bestimmung einer Einschließung von $LU - A$. Auf dem Rechner muss hier für eine verlässliche Einschließung zwei mal das dünn besetzte Matrix-Matrix-Produkt LU (jeweils einmal mit Rundung nach oben und nach unten) bestimmt werden.

Auch hier ist eine Vereinfachung möglich, durch welche nur ein Matrix-Matrix-Produkt erforderlich ist.

Dazu wird folgender Satz nach Oishi und Rump [107] verwendet.

Satz 5.3. *Es seien $A, L, U \in \mathbb{R}^{n\times n}$ und $LU \approx A$, wobei die LU-Zerlegung mit einer Variante der Gauß-Elimination berechnet wurde. Falls $n \cdot eps < 1$, so gilt mit $e = (1, \ldots, 1)^T$ und $\mathrm{diag}(|U|) = (|u_{11}|, \ldots, |u_{nn}|)^T$, auch bei Unterlauf,*

$$|LU - A| \leq \gamma_n \cdot |L| \cdot |U| + \frac{ne + \mathrm{diag}(|U|)}{1 - n \cdot eps} \cdot e^T \cdot eta.$$

Für einen Beweis siehe [107]. Diese Fehlerschranke gilt im Wesentlichen auch im Komplexen, es müssen nur einige der Konstanten wie γ_n passend angehoben werden, um die zusätzlich nötigen Gleitkommaoperationen zu berücksichtigen. Siehe dazu auch die Anmerkungen zu Fehlerschranken für komplexe Gleitkommaoperationen in Higham [58, Abschnitt 3.6].

Die Laufzeit wird bei dieser Methode von dem Produkt $|L| \cdot |U|$ dominiert. Je nach Besetztheit von L und U wird der Aufwand also in etwa halbiert. Die resultierende Einschließung ist im Allgemeinen allerdings breiter als bei direkter Berechnung, was in der Regel wiederum zu breiteren Einschließungen der Lösung und in manchen Fällen zu einer nicht erfolgreichen Verifikation führt.

5.1.2. Lösungsansatz basierend auf einer normweisen Fehlerabschätzung

Wie im letzten Abschnitt gesehen ist die Anwendung des Krawczyk-Operators zur Lösung dünn besetzter linearer Gleichungssysteme im Allgemeinen problematisch. Die besprochenen Methoden zum Lösen der auftretenden Dreieckssysteme sind entweder auf eine sehr spezifische Klasse von Matrizen beschränkt (Substitution mit und ohne Parallelepipede) oder in der Regel sehr aufwändig (implizite Berechnung der Inversen, Substitution mit Parallelepipeden für große Bandbreiten). Auch die Berechnung der Iterationsmatrix $LU - A$ kann, je nach Besetztheit des Systems, sehr aufwändig werden, so dass die Methode verglichen mit der näherungsweisen Lösung eines Punktsystems über eine LU-Zerlegung oft um deutlich mehr als eine Größenordnung langsamer ist und in vielen Fällen zu keinem verifizierten Ergebnis führt.

In diesem Abschnitt wird daher ein alternativer Ansatz nach Rump [126,127] beschrieben, welcher eine verifizierte normweise Abschätzung des Fehlers einer zuvor berechneten Näherungslösung bestimmt. Der Algorithmus ist insbesondere für symmetrisch positiv definite Matrizen gut anwendbar und unter geeigneten Bedingungen auch deutlich schneller als der im letzten Abschnitt besprochene Algorithmus. Je nach Kondition der Matrix kann die Laufzeit aber deutlich ansteigen. Auch gibt es für nicht symmetrisch positiv definite Matrizen starke Einschränkungen in der Anwendbarkeit. Dennoch ist dies im Allgemeinen die beste derzeit bekannte Methode zur verifizierten Lösung dünn besetzter Gleichungssysteme. Die folgenden Ausführungen lassen sich in der Regel direkt auf komplexe Systeme übertragen, ansonsten werden nötige Anpassungen explizit erwähnt.

Es sei $\hat{x} := A^{-1}b$ die exakte Lösung eines linearen Gleichungssystems $Ax = b$, $A \in \mathbb{R}^{n\times n}$, $b, x \in \mathbb{R}^n$ und $\tilde{x}$ eine entsprechende Näherungslösung. Dann gilt

$$||\hat{x} - \tilde{x}||_\infty \leq ||\hat{x} - \tilde{x}||_2 = ||A^{-1}(b - A\tilde{x})||_2 \leq ||A^{-1}||_2 \cdot ||b - A\tilde{x}||_2 = \frac{||b - A\tilde{x}||_2}{\sigma_n(A)}.$$

Hierbei ist $\sigma_n(A)$ der kleinste Singulärwert von A. Der Fehler der Näherungslösung $\tilde{x}$ eines linearen Gleichungssystems lässt sich also normweise verifiziert abschätzen, sofern eine untere Schranke für den kleinsten Singulärwert von A bestimmt werden kann.

Es sei nun A symmetrisch positiv definit. Dann sind die Singulärwerte von A mit den Eigenwerten von A identisch. Komplexe Matrizen werden als hermitesch positiv definit vorausgesetzt, die Singulärwerte sind dann positive reelle Zahlen und obige sowie folgende Aussagen direkt übertragbar. Auf der Maschine kann eine näherungsweise Cholesky-Zerlegung $A \approx \tilde{L}\tilde{L}^T$ bestimmt werden. Mit dieser lässt sich eine Näherung an den kleinsten Eigenwert (und damit den kleinsten Singulärwert) von A z.B. über inverse Iteration berechnen. Lemma 5.1 zeigt, wie dann eine verifizierte Unterschranke von $\sigma_n(A)$ bestimmt werden kann.

Lemma 5.1. *Es sei $\lambda \in \mathbb{R}$, $\lambda > 0$. Die näherungsweise Cholesky-Zerlegung $A - \lambda I \approx \tilde{L}\tilde{L}^T$ auf der Maschine sei erfolgreich durchführbar und es sei $D := \tilde{L}\tilde{L}^T - (A - \lambda I)$. Ist dann $\lambda > ||D||_2$, so gilt:*

$$\sigma_n(A) \geq \lambda - ||D||_2$$

Beweis: Da die näherungsweise Cholesky-Zerlegung erfolgreich durchgeführt wurde, ist $\tilde{L}\tilde{L}^T$ symmetrisch mit $\tilde{l}_{ii} \geq 0$ für $i = 1, \ldots, n$. Damit ist $A - \lambda I + D$ symmetrisch positiv semidefinit. Damit ist auch $A - (\lambda - ||D||_2)I$ symmetrisch positiv semidefinit. Es ist

$$\sigma_n(A - (\lambda - ||D||_2)I) = \sigma_n(A) - \lambda + ||D||_2 \geq 0$$

und damit folgt

$$\lambda - ||D||_2 \leq \sigma_n(A).$$

□

Insgesamt ergibt sich damit die in Satz 5.4 angegebene Methode zur verifizierten Lösung linearer Gleichungssysteme mit symmetrisch positiv definiter Systemmatrix.

Satz 5.4. *Es sei $A \in \mathbb{R}^{n\times n}$, $b, \tilde{x} \in \mathbb{R}^n$, $\tilde{x}$ Näherungslösung von $Ax = b$, $\lambda \in \mathbb{R}$, $\lambda > 0$, sowie $\tilde{L} \in \mathbb{R}^{n\times n}$ ein näherungsweiser Cholesky-Faktor von $A - \lambda I$. Mit $D := \tilde{L}\tilde{L}^T - (A - \lambda I)$ folgt dann, falls*

$$||D||_2 \leq \sqrt{||D||_1 ||D||_\infty} < \lambda$$

gilt, dass A regulär ist,

$$\sigma_n(A) \geq \lambda - \sqrt{||D||_1 ||D||_\infty}$$

und

$$A^{-1}b \in \tilde{x} + [-\Delta, \Delta]e$$

mit

$$\Delta := \frac{||b - A\tilde{x}||_2}{\lambda - \sqrt{||D||_1||D||_\infty}}$$

und $e = (1, \ldots, 1)^T$.

Die meisten Aussagen des Satzes folgen aus den bisherigen Erläuterungen. Für einen vollständigen Beweis und weitere Details siehe Rump [126].

Für die Cholesky-Zerlegung kann eine beliebige Bibliotheksroutine verwendet werden, insbesondere also auch eine für dünn besetzte Matrizen optimierte Variante. Es ist eine Cholesky-Zerlegung von A zur Bestimmung von $\tilde{x}$ und λ nötig, sowie eine Cholesky-Zerlegung von $A - \lambda I$. Die sonstigen Berechnungen können jeweils am Rechner durch entsprechende Rundungen sicher durchgeführt werden.

Wesentlich für den Gesamtaufwand ist neben den Cholesky-Zerlegungen die Bestimmung einer oberen Schranke für $||D||_2$. Da $||D||_2 \leq \sqrt{||D||_1||D||_\infty}$ kann diese mit einer elementweisen sicheren Abschätzung von $|D|$ bestimmt werden. Eine Möglichkeit zur Bestimmung auf der Maschine ist die direkte Berechnung von $|D|$ mit entsprechender Rundungsumschaltung. Diese ist allerdings in der Regel sehr aufwändig im Vergleich zu den restlichen Berechnungen. Eine alternative Möglichkeit nach Rump [131] gibt der folgende Satz an.

Satz 5.5. *Es sei* $A \in \mathbb{F}^{n\times n}$ *symmetrisch. Es sei* $\hat{L}$ *der symbolische Cholesky-Faktor von* A *(d.h.* $\hat{l}_{ij} = 0$*, falls* $l_{ij} = 0$ *und* $\hat{l}_{ij} = 1$ *sonst) und* $\tilde{L} \in \mathbb{F}^{n\times n}$ *der zugehörige näherungsweise Cholesky-Faktor. Für* $1 \leq i, j \leq n$ *definiere*

$$s(i,j) := |\{k \in \mathbb{N} \mid 1 \leq k < \min(i,j), \hat{l}_{ki}\hat{l}_{kj} \neq 0\}|$$

(die Betragsstriche stehen hier für die Anzahl der Elemente der angegebenen Menge) und

$$\alpha_{ij} := \begin{cases} \gamma_{s(i,j)+2} & \text{, falls } s(i,j) \neq 0 \\ 0 & \text{sonst.} \end{cases}$$

Definiere weiterhin

$$c_j := ((1 - \alpha_{jj})^{-1} a_{jj})^{\frac{1}{2}}$$

und

$$M := 3(2n + \max_\nu(a_{\nu\nu})).$$

Falls $\alpha_{jj} < 1$ *für alle* j*, so gilt für* $D \in \mathbb{F}^{n\times n}$ *mit*

$$d_{ij} := \alpha_{ij} c_i c_j + M \cdot eta$$

die Abschätzung

$$D \geq |\tilde{L}\tilde{L}^T - A|.$$

Diese Abschätzung gilt auch im Komplexen. Da A dann als hermitesch vorausgesetzt wird und zuvor ein näherungsweiser Cholesky-Faktor berechnet werden konnte, sind die Diagonalelemente von A reell und positiv. Die angegebene Definition von M ist also auch

direkt übertragbar. Für einen Beweis von Satz 5.5 siehe [131]. Die Berechnung nach Satz 5.5 ist deutlich weniger aufwändig als die direkte Berechnung von D, liefert allerdings ein weniger genaues Ergebnis, so dass evtl. die Bedingung von Satz 5.4 nicht verifiziert werden kann, obwohl das exakte D sie erfüllt. Es ist daher in der Praxis sinnvoll, zunächst die Abschätzung nach Satz 5.5 vorzunehmen und danach, falls die Bedingung von Satz 5.4 nicht erfüllt werden kann, D direkt zu berechnen.

Da bei der in diesem Abschnitt beschriebenen Methode eine normweise Abschätzung des Fehlers erfolgt, also der maximale elementweise Fehler abgeschätzt wird, kann es, insbesondere bei betragskleinen Lösungskomponenten, zu recht breiten Einschließungen kommen. Um die Qualität der Ergebniseinschließung zu verbessern, gibt es zwei Möglichkeiten.

Zum einen kann versucht werden, eine bessere Näherungslösung zu bestimmen, um so $||b - A\tilde{x}||_2$ möglichst klein zu halten. Auf dem Rechner eignet sich dazu die Bestimmung der Näherungslösung in einer Art Staggered-Format, also in Form einer Summe

$$\tilde{x} = \tilde{x}_1 + \tilde{x}_2 + \ldots + \tilde{x}_s \tag{5.10}$$

von s Vektoren, wobei $\tilde{x}_i$ als eine Näherungslösung des Punktsystems

$$Ay = b - \sum_{k=1}^{i-1} A\tilde{x}_k$$

bestimmt wird. Es ist bekannt, dass sich bei ausreichend guter Kondition $\sum_i \tilde{x}_i$ dem exakten Ergebnis annähert. Bei der späteren Implementierung wird bei der Berechnung zusätzlich Gebrauch vom langen Akkumulator gemacht, wodurch die Genauigkeit weiter gesteigert wird. Allerdings ist die Verbesserung der Näherungslösung bei Intervallsystemen nur bei sehr schmalen Intervalleinträgen sinnvoll, da ansonsten der Durchmesser der auftretenden Intervalle die Größe von $||\boldsymbol{b} - \boldsymbol{A}\tilde{x}||_2$ dominiert.

Die zweite Möglichkeit ist zu versuchen, über eine geeignete Skalierung die Kondition von A zu verbessern. Dadurch wird in der Regel $\sigma_n(A)$ größer, wodurch Δ aus Satz 5.4 kleiner wird. Weiterhin erleichtert dies die Bestimmung einer verifizierten Unterschranke für $\sigma_n(A)$, wodurch auch die Abschätzung nach Satz 5.5 eher zum Erfolg führt.

Für die Skalierung werden Diagonalmatrizen $S, T \in \mathbb{R}^{n\times n}$ gesucht, so dass möglichst

$$\mathrm{cond}(SAT) < \mathrm{cond}(A).$$

Für symmetrisch positiv definite Matrizen hat van der Sluis [136] gezeigt, dass hierbei die Wahl $S = T$ mit

$$s_{ii} = \sqrt{a_{ii}}$$

optimal ist. Für allgemeine Matrizen kann zunächst S so bestimmt werden, dass $||SA||_\infty \approx 1$, und anschließend T so, dass $||SAT||_1 \approx 1$. Dieses Vorgehen wird dann einige Male wiederholt, wodurch sich in der Regel Spalten- und Zeilennorm von SAT dem Wert 1 annähern. Näheres zu Matrixskalierungen und alternativen Methoden findet sich z.B. bei Bradley [26].

Auf der Maschine wird bei der Skalierung so vorgegangen, dass die Einträge der Diagonalmatrizen S und T zur nächsten Zweierpotenz gerundet werden. Die Berechnung von SAT ist somit (falls kein Unterlauf auftritt) exakt und das Ausgangsproblem wird nicht durch Rundungsfehler verfälscht.

Mit den bisherigen Überlegungen kann nun ein entsprechender Algorithmus für die verifizierte Lösung dünn besetzter linearer Gleichungssysteme mit symmetrisch positiv definiter (Punkt-)Systemmatrix formuliert werden. Dieser ist als Algorithmus 5.4 gegeben.

Bei nicht symmetrisch positiv definiten Systemen kann statt $Ax = b$ das System $A^T Ax = A^T b$ gelöst werden. Nachteil hierbei ist, dass

$$\mathrm{cond}(A^T A) = (\mathrm{cond}(A))^2.$$

Dieser Effekt kann für viele Systeme durch Skalierung der Systemmatrix gemindert werden. Trotzdem wird die Klasse der so lösbaren Systeme eingeschränkt. Eine Alternative wäre, die LU-Zerlegung $LU = A$ zu bestimmen und untere Schranken für $\sigma_n(L)$ und $\sigma_n(U)$ zu berechnen, denn es gilt

$$\sigma_n(A) \geq \sigma_n(L)\sigma_n(U).$$

Allerdings wandert die Kondition von A bei einer LU-Zerlegung im Wesentlichen in den Faktor U [126], so dass L zwar gut konditioniert ist, U aber eine ähnliche Kondition wie A aufweist und sich somit das gleiche Problem wie zuvor ergibt.

Die Anpassung für Intervallsysteme hingegen ist vergleichsweise einfach. Im Prinzip wird die besprochene Methode für das zugehörige Mittelpunktssystem angewendet und der Radius der Elemente von $\boldsymbol{A}$ wird als Störung interpretiert. Der Radius wird also bei der Verifizierung der Schranke für den kleinsten Singulärwert mit berücksichtigt, so dass sie bei erfolgreicher Verifizierung für alle Punktmatrizen $A \in \boldsymbol{A}$ gilt. Bei der Bestimmung von Δ aus Satz 5.4 wird das Residuum intervallmäßig berechnet, anschließend die Norm bestimmt und im Nenner die 2-Norm des Radius von $\boldsymbol{A}$ abgezogen.

Die beschriebene Methode ist für dünn besetzte Punktsysteme und Intervallsysteme mit engen Intervalleinträgen gut geeignet, insbesondere für symmetrisch positiv definite Systeme. Führt die Abschätzung von $||D||_2$ mittels Satz 5.5 zum Erfolg, so wird der Aufwand von zwei Cholesky-Zerlegungen dominiert, für welche hocheffiziente Bibliotheksroutinen zur Verfügung stehen. Der Aufwand für die Methode liegt dann innerhalb einer Größenordnung des Aufwands für die Bestimmung einer einfachen Gleitkommalösung mittels Cholesky-Zerlegung. Für Punktsysteme können darüber hinaus bei Berechnung des Residuums gemäß (5.10) sehr genaue verifizierte Einschließungen bestimmt werden. Bei Systemen mit breiteren Intervalleinträgen oder bei einer einfachen Berechnung der Näherungslösung können die berechneten Einschließungen allerdings aufgrund der normweisen Abschätzung des Defekts sehr breit werden. Darüber hinaus ist die Anwendbarkeit für nicht symmetrisch positiv definite Matrizen im Allgemeinen auf gut konditionierte Matrizen beschränkt.

Eingabe : Systemmatrix A positiv definit, rechte Seite b
Ausgabe : Einschließung $\hat{\boldsymbol{x}}$ der Intervallhülle $\boldsymbol{x}$ der exakten Lösungsmenge
Bestimme Matrix S für van der Sluis Skalierung
Setze $A = SAS$ und $b = Sb$
Bestimme näherungsweise Cholesky-Zerlegung $\tilde{R}\tilde{R}^T \approx A$
Bestimme Näherungslösung $\tilde{x}$ aus Cholesky-Zerlegung
Bestimme Näherung $\lambda \approx \sigma_n(A)$ über inverse Iteration mit $\tilde{R}$ und $\tilde{R}^T$
Setze $f = 0.9$
repeat
  Setze $\lambda = f \cdot \lambda$
  Bestimme näherungsweise Cholesky-Zerlegung $\tilde{L}\tilde{L}^T \approx A - \lambda I$
  Setze $f = f - 0.1$
until $\lambda \leq 0$ *oder Cholesky-Zerlegung erfolgreich*
if $\lambda \leq 0$ **then**
  Verifizierung nicht erfolgreich
Bestimme $\tau \geq ||\tilde{L}\tilde{L}^T - A - \lambda I||_2$ mittels Satz 5.5
if $\lambda \leq \tau$ **then**
  Bestimme $\tau \geq ||\tilde{L}\tilde{L}^T - A - \lambda I||_2$ mittels direkter Berechnung
if $\lambda > \tau$ **then**
  Bestimme $\Delta = ||b - A\tilde{x}||_2/(\lambda - \tau)$
  Verifizierte Lösung ist $\hat{x} = \tilde{x} + [-\Delta, \Delta] \cdot (1, \ldots, 1)^T$
else
  Verifizierung nicht erfolgreich

Algorithmus 5.4: Verifizierte Lösung eines dünn besetzten symmetrisch positiv definiten linearen Gleichungssystems mittels normweiser Abschätzung des Defekts

5.2. Implementierung

In diesem Abschnitt wird die Implementierung von Lösern basierend auf dem in 5.1 besprochenen theoretischen Hintergrund beschrieben. Die beiden grundsätzlichen Methoden werden dabei als einzelne Löser in einem gemeinsamen Softwarepaket implementiert. Dadurch können einige Schnittstellen und Hilfsfunktionen von beiden Lösern gemeinsam genutzt werden, was zu einer kompakteren Implementierung führt.

Im Folgenden werden die beiden Löser getrennt beschrieben. Dabei wird genauer auf die konkrete Implementierung der vorgestellten Algorithmen und wichtige Details bei der Realisierung auf dem Rechner eingegangen. Die Implementierungen nutzen Templates, um den Quelltext möglichst schlank und leicht wartbar zu halten, aber trotzdem Löser für alle C-XSC Grunddatentypen zu realisieren. Weitere Schwerpunkte sind die Verwendung der dünn besetzten C-XSC Datentypen aus Abschnitt 3.3 sowie die Parallelisierung einiger Teile der Löser mittels OpenMP. Ebenso wird auf die Benutzung optimierter Routinen aus numerischen Bibliotheken wie BLAS und LAPACK und vor allem UMFPACK und CHOLMOD zur Bestimmung der dünn besetzten LU- bzw. Cholesky-Zerlegung eingegangen. Vor der Beschreibung der eigentlichen Löser wird zunächst ein Interface zu UMFPACK und CHOLMOD beschrieben.

5.2.1. Schnittstelle zu UMFPACK und CHOLMOD

Bereits in Abschnitt 2.4.4 wurde die SuiteSparse-Bibliothek, eine Sammlung von hochoptimierten Routinen für dünn besetzte Matrizen, kurz beschrieben. Diese enthält Routinen für eine effiziente Berechnung der LU-Zerlegung (UMFPACK) und der Cholesky-Zerlegung (CHOLMOD) einer dünn besetzten Matrix. Die Zerlegungen werden durch geeignete Permutationen der Matrix so berechnet, dass möglichst geringer Fill-in entsteht, so dass die Faktoren L und U bzw. der Cholesky-Faktor L möglichst dünn besetzt sind. Die Routinen aus SuiteSparse zählen zu den effizientesten ihrer Art und werden in der Praxis vielfach verwendet, unter anderem auch in Matlab.

Für die Benutzung mit C-XSC Datentypen wird jeweils eine Schnittstelle zu UMFPACK und CHOLMOD benötigt. Durch die Verwendung des *Compressed Column Storage*-Formats für die dünn besetzten Datentypen in C-XSC (siehe Abschnitt 3.3) sind solche Schnittstellen recht einfach zu erstellen, da auch UMFPACK und CHOLMOD diese Datenstruktur verwenden.

Zunächst wird die Schnittstelle zu UMFPACK betrachtet. Listing 5.1 zeigt einen Auszug aus der Funktion zur Durchführung einer LU-Zerlegung einer reellen dünn besetzten C-XSC Punktmatrix mit Hilfe von UMFPACK.

```
1  void lu_decomp(const srmatrix& A, srmatrix& L, srmatrix& U, intvector& p,
2                 intvector& q, int& Err) {
3
4    //...
5
6    const std::vector<int>&  pv   = A.column_pointers();
7    const std::vector<int>&  indv = A.row_indices();
8    const std::vector<real>& xv   = A.values();
```

```
9
10    int*     Ap = const_cast<int*>(&pv[0]);
11    int*     Ai = const_cast<int*>(&indv[0]);
12    double* Ax = const_cast<double*>(reinterpret_cast<const double*>(&xv[0]));
13
14    //...
15
16    void *Symbolic, *Numeric;
17
18    //Symbolic analysis
19    umfpack_di_symbolic(n,n,Ap,Ai,Ax,&Symbolic,control,info);
20
21    if(info[UMFPACK_STATUS] != UMFPACK_OK) {
22      Err = 1;
23      return;
24    }
25
26    //Numeric decomposition
27    umfpack_di_numeric(Ap,Ai,Ax,Symbolic,&Numeric,control,info);
28
29    if(info[UMFPACK_STATUS] != UMFPACK_OK) {
30      Err = 2;
31      return;
32    }
33
34    //...
35
36  }
```

Listing 5.1: Auszug aus der UMFPACK-Schnittstelle: Cast, symbolische und numerische Zerlegung

Die Parameter der Funktion haben folgende Bedeutung:

- `A`: Referenz auf die zu zerlegende dünn besetzte Matrix.
- `L`: Referenz auf eine dünn besetzte Matrix. Nach Ablauf der Funktion enthält sie den Faktor L der LU-Zerlegung.
- `U`: Referenz auf eine dünn besetzte Matrix. Nach Ablauf der Funktion enthält sie den Faktor U der LU-Zerlegung.
- `p`: Referenz auf einen `intvector`. Enthält nach Ablauf der Funktion Informationen für die Zeilenpermutation von A, die vor der Zerlegung zur Minimierung des Fill-ins von UMFPACK durchgeführt wurde.
- `q`: Referenz auf einen `intvector`. Enthält nach Ablauf der Funktion Informationen für die Spaltenpermutation von A, die vor der Zerlegung zur Minimierung des Fill-ins von UMFPACK durchgeführt wurde.
- `Err`: Ein Fehlercode. Ist `Err` gleich 0, so konnte die LU-Zerlegung erfolgreich durchgeführt werden. Ansonsten ist während der Zerlegung ein Fehler aufgetreten.

Im Auszug aus Listing 5.1 ist zu Beginn die Übertragung der als C-XSC-Datentyp gespeicherten dünn besetzten Matrix in eine UMFPACK-Datenstruktur zu sehen. Da die verwendete Datenstruktur identisch ist, können die im Speicher vorliegenden Daten des C-XSC Datentyps durch entsprechende Casts der als STL-Vektoren vorliegenden Daten in einfache eindimensionale Arrays direkt verwendet werden. Dadurch müssen die Daten nicht extra kopiert werden und es wird kein zusätzlicher Speicherplatz benötigt.

Anschließend wird die symbolische und numerische Zerlegung mittels UMFPACK durchgeführt. Dabei wird jeweils geprüft, ob ein Fehler aufgetreten ist und gegebenenfalls ein Fehlercode gesetzt und die Funktion beendet. Konnte die numerische Zerlegung erfolgreich berechnet werden, so muss das Ergebnis in geeignete C-XSC Datentypen umgewandelt werden. Listing 5.2 zeigt das entsprechende Vorgehen.

```
1   //...
2
3   //Get nnz(L) and nnz(U)
4   int lnz = static_cast<int>(info[UMFPACK_LNZ]);
5   int unz = static_cast<int>(info[UMFPACK_UNZ]);
6
7   //Read out data for L,U,p,q,R...
8   p = intvector(n);
9   q = intvector(n);
10
11  int* Lp    = new int[n+1];
12  int* Li    = new int[lnz];
13  double* Lx = new double[lnz];
14  int* Up    = new int[n+1];
15  int* Ui    = new int[unz];
16  double* Ux = new double[unz];
17  int* pi    = static_cast<int*> (&p[1]);
18  int* qi    = static cast<int*> (&q[1]);
19
20  umfpack_di_get_numeric(Lp,Li,Lx,Up,Ui,Ux,pi,qi,NULL,NULL,NULL,Numeric);
21
22  //Copy L
23  L = srmatrix(n,n,lnz,Lp,Li,reinterpret_cast<real*>(Lx),compressed_row);
24
25  delete[] Lp;
26  delete[] Li;
27  delete[] Lx;
28
29  //Copy U
30  U = srmatrix(n,n,unz,Up,Ui,reinterpret_cast<real*>(Ux),
31               compressed_column);
32
33  delete[] Up;
34  delete[] Ui;
35  delete[] Ux;
36
37  //Delete numeric data
38  umfpack_di_free_numeric(&Numeric);
39
```

```
40    Err = 0;
41  }
```

Listing 5.2: Auszug aus UMFPACK-Schnittstelle: Umwandlung des Ergebnisses in einen C-XSC Datentyp

Zunächst wird die Zahl der Nicht-Null-Einträge von L und U ausgelesen. Danach wird Speicher für L und U in Form von einfachen Arrays angefordert (dies ist hier nötig, da zum einen L im *Compressed Row Storage*-Format vorliegt und zum anderen die Reihenfolge der Spalteneinträge der Ausgabe evtl. nicht sortiert ist). Für die Permutationsvektoren wird der C-XSC Datentyp `intvector` verwendet. Durch passend gecastete Zeiger kann UMFPACK direkt in diese Vektoren schreiben. Anschließend werden mittels Konstruktoren des Datentyps `srmatrix` die Dreiecksmatrizen L und U als C-XSC Datentyp angelegt. Der Speicher für die zum Auslesen des UMFPACK-Ergebnisses nötigen eindimensionalen Arrays kann anschließend wieder freigegeben werden. Die Version für komplexe Matrizen ist im Wesentlichen analog zur hier vorgestellten Version implementiert.

Anzumerken ist noch, dass UMFPACK bei Verwendung des Datentyps `int` für die Indizes auch auf 64-Bit Systemen nicht mehr als 4 GB Speicher ansprechen kann. Für eine Ausnutzung des kompletten verfügbaren Speichers müssen der Datentyp `long` für die Indizierung genutzt und die für diesen Datentyp angepassten Versionen der UMFPACK-Funktionen aufgerufen werden. C-XSC - und aus Kompatibilitätsgründen somit auch die dünn besetzten C-XSC Datentypen - unterstützen aktuell allerdings nur den `int`-Datentyp. Es muss also, wie in der oben beschriebenen Schnittstelle geschehen, mit der 32 Bit Version von UMFPACK gearbeitet werden. Wie in Abschnitt 6.2.1 beschrieben, wird für künftige Versionen von C-XSC eine Wahlmöglichkeit für den Indizierungsdatentyp in Betracht gezogen. Sollte dies realisiert werden, kann auch die UMFPACK-Schnittstelle leicht angepasst und somit der gesamte Hauptspeicher genutzt werden. Die im Folgenden beschriebene CHOLMOD-Schnittstelle unterliegt nicht diesen Beschränkungen, da CHOLMOD unabhängig vom Indizierungsdatentyp auf 64-Bit Systemen stets 8 Byte lange Zeiger verwendet.

Das Vorgehen für das CHOLMOD-Interface ist sehr ähnlich zum UMFPACK-Interface, CHOLMOD verwendet nur etwas andere Datenstrukturen (wobei auch hier für die Speicherung der eigentlichen dünn besetzten Matrix das *Compressed Column Storage*-Format verwendet wird) und Funktionsnamen. Listing 5.3 zeigt die komplette Funktion zur Berechnung der Cholesky-Zerlegung für reelle Matrizen mittels CHOLMOD.

```
1  void chol(srmatrix& A, srmatrix& L, intvector& p, int& err) {
2    ::cholmod_common cm;
3
4    std::vector<int>&  Ap  = A.column_pointers();
5    std::vector<int>&  Ai  = A.row_indices();
6    std::vector<real>& Ax  = A.values();
7
8    cholmod_start(&cm);
9    cm.print = 0;
10
11   ::cholmod_sparse *A_chol =
```

```
12          cholmod_allocate_sparse(ColLen(A), RowLen(A),
13                                  RowLen(A)+0.5*(A.get_nnz()-RowLen(A)),
14                                  true, true, -1, CHOLMOD_REAL, &cm);
15
16    //copy lower triangular part of A
17    int nnz = 0;
18    for(int j=0 ; j<RowLen(A) ; j++) {
19      ((int*)(A_chol->p))[j] = nnz;
20      for(int i=Ap[j] ; i<Ap[j+1] ; i++) {
21        if(Ai[i] >= j) {
22          ((int*)(A_chol->i))[nnz] = Ai[i];
23          ((double*)(A_chol->x))[nnz] = _double(Ax[i]);
24          nnz++;
25        }
26      }
27    }
28    ((int*)(A_chol->p))[RowLen(A)] = nnz;
29
30    ::cholmod_factor* factor =
31              (::cholmod_factor*) cholmod_analyze(A_chol,&cm);
32
33    if(cm.status != 0) {
34      err = 1;
35      cholmod_free_sparse(&A_chol,&cm);
36      cholmod_free_factor(&factor,&cm);
37      cholmod_finish(&cm);
38      return;
39    }
40
41    if(!cholmod_factorize(A_chol,factor,&cm)) {
42      err = 2;
43      cholmod_free_sparse(&A_chol,&cm);
44      cholmod_free_factor(&factor,&cm);
45      cholmod_finish(&cm);
46      return;
47    }
48
49    if(cm.status == CHOLMOD_NOT_POSDEF) {
50      err = 3;
51      cholmod_free_sparse(&A_chol,&cm);
52      cholmod_free_factor(&factor,&cm);
53      cholmod_finish(&cm);
54      return;
55    }
56
57    if(!(factor->is_ll)) {
58      cholmod_change_factor(CHOLMOD_REAL, 1, 0, 1, 1, factor, &cm);
59    }
60
61    if(cm.status == CHOLMOD_NOT_POSDEF) {
62      err = 4;
63      cholmod_free_sparse(&A_chol,&cm);
```

```
64      cholmod_free_factor(&factor,&cm);
65      cholmod_finish(&cm);
66      return;
67    }
68
69    Resize(p, A_chol->nrow);
70    int *perm_int = (int*)(factor->Perm);
71    for(int i=1 ; i<=A_chol->nrow ; i++)
72      p[i] = perm_int[i-1];
73
74
75    cholmod_free_sparse(&A_chol,&cm);
76
77    ::cholmod_sparse* L_chol =
78          (::cholmod_sparse*) cholmod_factor_to_sparse(factor,&cm);
79
80
81    L = srmatrix(L_chol->nrow, L_chol->ncol,
82                 ((int*)(L_chol->p))[L_chol->ncol], (int*)L_chol->p,
83                 (int*)L_chol->i, reinterpret_cast<real*>(L_chol->x),
84                 compressed_column);
85
86    cholmod_free_sparse(&L_chol,&cm);
87    cholmod_free_factor(&factor,&cm);
88    cholmod_finish(&cm);
89
90    err = 0;
91  }
```

Listing 5.3: Cholesky-Zerlegung einer reellen Matrix mittels CHOLMOD

Die Parameter der Funktion haben folgende Bedeutung:

- `A`: Referenz auf die zu zerlegende, dünn besetzte Matrix.
- `L`: Nach erfolgreichem Ablauf der Funktion eine Referenz auf den berechneten Cholesky-Faktor.
- `p`: Permutationsvektor der zum Erreichen eines geringen Fill-ins von CHOLMOD berechneten symmetrischen Zeilen- und Spaltenpermutation von A.
- `Err`: Ein Fehlercode. Ist `Err` nach Ablauf der Funktion ungleich 0, so ist während der Cholesky-Zerlegung ein Fehler aufgetreten.

Anders als bei der UMFPACK-Schnittstelle wird für die Zerlegung hier nicht mit den Originaldaten gearbeitet, da innerhalb der CHOLMOD-Datenstruktur für dünn besetzte Matrizen zusätzliche Informationen gespeichert werden, die sonst nicht mehr richtig freigegeben werden könnten. Allerdings wird für die Zerlegung nur das untere Dreieck der Matrix `A` benötigt, es muss also nur etwa die Hälfte der Matrix umkopiert werden. Anschließend werden die CHOLMOD-Funktionen zur symbolischen und numerischen Cholesky-Zerlegung aufgerufen. Sollten dabei Fehler auftreten, so wird der Fehlercode entsprechend gesetzt und alle bis dahin belegten Ressourcen werden freigegeben.

Zum Abschluss werden die Daten aus den CHOLMOD-Objekten, welche die Ergebnisse der symbolischen und numerischen Zerlegung enthalten, ausgelesen und in den C-XSC Variablen für den Permutationsvektor und den eigentlichen Cholesky-Faktor gespeichert. Ein einfacher Cast würde hier nicht ausreichen, da die CHOLMOD-Objekte am Ende der Funktion freigegeben werden müssen und so auch die zugehörigen Daten gelöscht werden. Zeiger, die auf diese Daten verweisen, würden dann also auf bereits freigegebenen Speicher zeigen. Die der hier beschriebenen Funktion entsprechende Version für komplexe Matrizen ist wiederum analog aufgebaut.

5.2.2. Löser basierend auf dem Krawczyk-Operator

In diesem Abschnitt wird die Implementierung eines Lösers für dünn besetzte lineare Gleichungssysteme basierend auf dem Krawczyk-Operator (im Folgenden auch als dünn besetzter Krawczyk-Löser bezeichnet) beschrieben. Wie in Abschnitt 5.1 gesehen, erfordert dabei die Lösung der auftretenden Dreieckssysteme besondere Aufmerksamkeit. Alle drei in 5.1 vorgestellten Algorithmen zur Lösung dieser Dreieckssysteme wurden für den hier beschriebenen Löser implementiert und können vom Benutzer ausgewählt werden. Der eigentliche Kern des Lösers wird von der Wahl der Methode zur Lösung der Dreieckssysteme aber nicht beeinflusst, weshalb er zunächst gesondert beschrieben wird. Details zur Implementierung der Routinen zur Lösung der Dreieckssysteme folgen im Anschluss. Dabei wird jeweils auf Maßnahmen zur Parallelisierung oder allgemeinen Beschleunigung der Berechnungen eingegangen.

Implementierung der zentralen Funktionalität

Analog zu den in Kapitel 4 beschriebenen Lösern können die wesentlichen Optionen für den Aufruf des dünn besetzten Krawczyk-Lösers über eine `struct` gesetzt werden, welche beim Aufruf des Lösers als Parameter übergeben werden kann. Auch hier werden Standard-Werte verwendet, falls nicht explizit eine eigene Konfiguration übergeben wird. Die Konfigurations-`struct` für den dünn besetzten Krawczyk-Löser wird in Listing 5.4 angegeben.

```
1  enum triSysMethod { FORWARD_BACKWARD, IMPLICIT_INVERSE, BANDED };
2
3  struct slssconfig {
4    int    precResidual;              // Dot product precision for
5                                      // computation of residual
6    bool   msg;                       // Status message output
7    int    threads;                   // Number of threads to use for OpenMP
8    int    maxIterVer;                // Maximum number of iterations during
9                                      // the verification step
10   real   epsVer;                    // Epsilon for the Epsilon inflation
11                                     // during verification
12   triSysMethod  triMethod;          // Solving method for triangular
13                                     //systems
14   int    precIterMat;               // Dot product precision for
15                                     // computation of [LU-A]
```

```
16    bool  forceSuiteSparse;            // Force usage of SuiteSparse for
17                                       //  banded systems
18    bool  forceHighPrecIterMatrix;     // Force high precision computation of
19                                       // [LU-A]
20    bool  fastDecompErrorComputation;  // Fast computation of [LU-A]
21    bool  fastVerification;            // Use faster verification method
22    bool  highPrecApprxSolution;       // High precision approximate solution
23    int   apprxSolutionStagPrec;       // Precision for high precision
24                                       // approximate solution
25    bool  storeBase;                   // For banded systems: Store base
26                                       // matrices?
27    bool  useCholesky;                 // Use cholesky decomposition
28
29    slssconfig() : precResidual(2), precIterMat(1), msg(false), threads(-1),
30                   epsVer(0.1), maxIterVer(5), triMethod(FORWARD_BACKWARD),
31                   forceSuiteSparse(false), fastVerification(true),
32                   highPrecApprxSolution(true), storeBase(false),
33                   apprxSolutionStagPrec(3), forceHighPrecIterMatrix(false),
34                   fastDecompErrorComputation(true), useCholesky(true)  {}
35 };
```

Listing 5.4: Konfigurations-`struct` für den dünn besetzten Krawczyk-Löser

Die einzelnen Attribute der `struct` haben dabei folgende Bedeutung:

- `precResidual`: Falls kein hochgenaues Residuum berechnet werden soll (s.u.), gibt `precResidual` an, mit welcher Präzision das Matrix-Vektor-Produkt bei der Berechnung von $\boldsymbol{z} = b - A\tilde{x}$ bestimmt wird. Die Näherungslösung selbst wird dabei über einfache Gleitkommalösung der Dreieckssysteme aus der LU-Zerlegung bestimmt (hierbei handelt es sich also um Punkt-Dreieckssysteme, d.h. die speziellen Algorithmen zur Lösung von Dreieckssystemen mit intervallwertiger rechter Seite werden hier nicht benötigt). Der Standardwert ist hierbei 2, also doppelte *double*-Präzision.

- `msg`: Gibt an, ob während der Ausführung des Lösers Statusmeldungen auf `cout` ausgegeben werden sollen. Die Standardeinstellung ist `false`.

- `threads`: Gibt die Anzahl der von OpenMP zu verwendenden Threads an. Bei einem Wert kleiner oder gleich Null wird dabei die Standardeinstellung von OpenMP (z.B. aus der entsprechenden Umgebungsvariablen) verwendet. Falls SuiteSparse mit BLAS-Unterstützung kompiliert wurde, kann diese Einstellung je nach verwendeter BLAS Bibliothek auch Einfluss auf die Performance von SuiteSparse (also auf die LU- bzw. Cholesky-Zerlegung von A) haben. Standardmäßig wird für `threads` der Wert -1 verwendet.

- `maxIterVer`: Die maximale Anzahl an durchzuführenden Verifikationsschritten. In der Regel sollte die Standardeinstellung 4 ausreichend sein. Je nach System und Einstellung der anderen Optionen (insbesondere `fastDecompErrorComputation`) können allerdings auch mehr Verifikationsschritte nötig sein. Dabei ist zu bedenken, das die Verifikationsschritte für dünn besetzte Systeme in der Regel recht

aufwändig sind. Diese Einstellung hat keinen Effekt, falls die schnelle Verifikation (Attribut `fastVerification`) aktiviert ist.

- `epsVer`: Epsilon für die Epsilon-Aufblähung (siehe Abschnitt 4.1.1) in jedem Verifikationsschritt. Die Standardeinstellung ist 0.1. Diese Einstellung hat ebenfalls keinen Effekt, falls die schnelle Verifikation aktiviert ist.

- `triMethod`: Mit diesem Attribut wird die Methode zur Lösung der Dreieckssysteme während der Verifizierung ausgewählt. Dazu werden die möglichen Werte des Enumerations-Typs `triSysMethod` verwendet (siehe Listing 5.4). Diese sind:

  1. `FORWARD_BACKWARD`: Einfache Vorwärts- bzw. Rückwärtssubstitution.
  2. `IMPLICIT_INVERSE`: Implizite Berechnung der Inversen nach Algorithmus 5.2.
  3. `BANDED`: Vorwärts- und Rückwärtssubstitution für Bandsysteme mittels Parallelepipeden nach Algorithmus 5.3.

  Standardmäßig wird einfache Vorwärts- und Rückwärtssubstitution verwendet.

- `precIterMat`: Präzision für die Berechnung der Iterationsmatrix $[LU - A]$. Die Standardeinstellung ist 1, also Gleitkommarechnung im *double*-Format. Eine höhere Präzision kann zu einem genaueren Ergebnis führen und bei manchen Gleichungssystemen auch eine erfolgreiche Verifikation erst ermöglichen. Allerdings steigt der Aufwand dadurch deutlich an. Diese Einstellung hat keinen Effekt, falls `fastDecompErrorComputation` aktiviert wurde. Weitere Details folgen im Verlauf dieses Kapitels.

- `fastDecompErrorComputation`: Aktiviert die Verwendung der schnelleren, aber ungenaueren Berechnung von $[LU - A]$ gemäß Satz 5.3. Standardmäßig ist die schnelle Berechnung deaktiviert.

- `forceSuiteSparse`: Erzwingt die Verwendung von SuiteSparse für die Dreieckszerlegung von A, falls `triSysMethod` auf `BANDED` gesetzt wurde. Normalerweise wird hierfür eine eigene, genauere Routine verwendet (diese wird im Verlauf dieses Abschnitts noch näher erläutert), welche die Bandbreite von A bei der Zerlegung erhält. Bei Verwendung von SuiteSparse kann die Bandbreite durch Zeilen- und Spalten-Permutation zur Reduzierung des Fill-ins bzw. zur Pivotisierung ansteigen. Allerdings ist SuiteSparse in der Regel deutlich schneller, wobei dieser Vorteil bei einer Verbreiterung der Bandbreite durch die dann aufwändigere Lösung der Dreieckssysteme wieder zunichte gemacht wird. Die Standardeinstellung ist daher `false`.

- `forceHighPrecIterMatrix`: Falls nicht die Lösungsmethode mit Parallelepipeden für die auftretenden Dreieckssysteme verwendet wird, kann mit dieser Option trotzdem die hochgenaue Matrixzerlegung verwendet werden. Da diese aber von einem Bandsystem mit dicht besetzten Bändern ausgeht, sollte diese Option auch nur für entsprechende System verwendet werden. Standardmäßig ist sie deaktiviert.

- `fastVerification`: Aktiviert die in Abschnitt 5.1 erläuterte schnelle Verifikation. Diese begrenzt die Anzahl der notwendigen Verifikationsschritte stets auf zwei. Allerdings erhält man so in der Regel breitere Einschließungen. Die Standardeinstellung ist daher `false`.

- `highPrecApprxSolution`: Aktiviert die hochgenaue Berechnung der Näherungslösung und des Residuums in Form einer Staggered-artigen Darstellung

$$\tilde{x} = \tilde{x}_1 + \ldots + \tilde{x}_s.$$

  Dabei entspricht s dem in `approxSolutionStagPrec` gesetzten Wert. Details zu dieser Berechnung folgen im Verlauf dieses Abschnitts. Diese Einstellung ist standardmäßig aktiviert.

- `apprxSolutionStagPrec`: Falls `highPrecApprxSolution` aktiviert ist, gibt dieses Attribut den Wert von s an. Standardmäßig wird der Wert 3 verwendet.

- `storeBase`: Falls für die Lösung der Dreieckssysteme die Option `BANDED` gewählt wurde, werden bei Aktivierung dieser Option die Basismatrizen für die berechneten Parallelepipede gespeichert. Da diese in jedem Verifikationsschritt identisch sind, wird so die Verifikation ab dem zweiten Schritt deutlich beschleunigt. Allerdings erfordert dies einen höheren Speicherbedarf. Standardmäßig ist diese Option daher deaktiviert. Eine genauere Erläuterung folgt im weiteren Verlauf dieses Abschnitts.

- `useCholesky`: Gibt an, ob statt einer LU- eine Cholesky-Zerlegung verwendet werden soll. Die Standardeinstellung ist `false`. Im Gegensatz zum normweisen Löser wird hier nicht verifiziert, ob die Systemmatrix wirklich symmetrisch positiv definit ist, sondern nur, ob sie regulär ist. Hat `useCholesky` den Wert `true` und der Löser berechnet erfolgreich eine Einschließung, so bedeutet dies nur, dass die näherungsweise Cholesky-Zerlegung für diese Matrix erfolgreich durchgeführt werden konnte und von einer für die Verifikation ausreichenden Güte war.

Analog zu den in Kapitel 4 beschriebenen Lösern wurde auch der dünn besetzte Krawczyk-Löser intern unter intensiver Benutzung von Templates realisiert. Aufgrund des Aufbaus und der Hierarchie der C-XSC Datentypen ist ein Aufruf von Template-Funktionen direkt durch den Benutzer nicht sinnvoll, da eine große Anzahl von Template-Parametern nötig ist. Der Benutzer ruft daher für jeden Datentyp passende, einfache Funktionen auf, welche als Startpunkt fungieren und ihrerseits die als Template-Funktion implementierte Hauptroutine des Lösers passend ausprägen und aufrufen. Listing 5.5 zeigt den Kopf dieser Template-Funktion.

```
1 template<typename TSysMat, typename TRhs, typename TSolution,
2          typename TMidMat, typename TMidRhs, typename TC,
3          typename TRhsVec, typename TVec, typename TIVec,
4          typename TElement, typename TDot, typename TIDot,
5          typename TSIMat>
6 void slssmain ( const TSysMat  Ao, const TRhs  bo, TSolution&  x,
```

```
7                    int& Err, slssconfig cfg );
```

Listing 5.5: Kopf der als Template-Funktion implementierten Hauptroutine des dünn besetzten Krawczyk-Lösers

Die verwendeten Template-Parameter haben dabei folgende Bedeutung:

- `TSysMat`: Typ der Systemmatrix. Die ist einer der dünn besetzten Matrixtypen, also `srmatrix`, `simatrix`, `scmatrix` oder `scimatrix`.
- `TRhs`: Typ der rechten Seite. Da auch mehrere rechte Seiten unterstützt werden und die rechten Seiten in der Regel voll besetzt sind, muss es sich um einen dicht besetzten Matrixtyp handeln, also `rmatrix`, `imatrix`, `cmatrix` oder `cimatrix`.
- `TSolution`: Typ der Lösung. Hierbei muss es sich um einen zu `TRhs` passenden dicht besetzten Intervallmatrixtyp handeln, also `imatrix` oder `cimatrix`.
- `TMidMat`: Typ der Mittelpunkts-Systemmatrix, also `srmatrix` oder `scmatrix`.
- `TMidRhs`: Typ der Mittelpunktsmatrix der rechten Seite, also `rmatrix` oder `cmatrix`.
- `TC`: Typ der Iterationsmatrix $[LU - A]$, also entweder `simatrix` oder `scimatrix`.
- `TRhsVec`: Typ eines passenden Vektortyps für eine rechte Seite. Möglich sind also `rvector`, `ivector`, `cvector` oder `civector`.
- `TVec`: Typ für einen allgemeinen dicht besetzten Punktvektor, passend zum Typ der Systemmatrix. Möglich sind also `rvector` oder `cvector`.
- `TIVec`: Typ für einen allgemeinen dicht besetzten Intervallvektor, passend zum Typ der Systemmatrix. Möglich sind also `ivector` oder `civector`.
- `TElement`: Typ für ein einzelnes Element der Systemmatrix, also `real`, `interval`, `complex` oder `cinterval`.
- `TDot`: Zum Typ der Systemmatrix passender Akkumulator-Datentyp für Punktberechnungen, also `dotprecision` oder `cdotprecision`.
- `TIDot`: Zum Typ der Systemmatrix passender Akkumulator-Datentyp für Intervallberechnungen, also `idotprecision` oder `cidotprecision`.
- `TSIMat`: Typ für eine zum Typ der Systemmatrix passende, dünn besetzte Intervallmatrix. Möglich sind also `simatrix` oder `scimatrix`.

Die Parameter der Funktion haben folgende Bedeutung:

- `Ao`: Eine Referenz auf die Systemmatrix. Diese wird hier als `Ao` bezeichnet, da später die für weitere Berechnungen entsprechend der Dreieckszerlegung permutierte Systemmatrix mit `A` bezeichnet wird.

- `bo`: Eine Referenz auf die rechte Seite des Systems. Der Grund für die Benennung ist analog zu `Ao`.
- `x`: Der Lösungsvektor mit der berechneten verifizierten Einschließung der exakten Lösung.
- `Err`: Ein Fehlercode. Die Fehlercodes entsprechen im Wesentlichen den Fehlercodes im dicht besetzten Fall, siehe Abschnitt 4.1.2 für Details.
- `cfg`: Ein Objekt der bereits beschriebenen Konfigurations-`struct` entsprechend Listing 5.4.

Wie beim Löser für allgemeine dicht besetzte Systeme werden auch hier unter- und überbestimmte Gleichungssysteme unterstützt. Die Implementierung erfolgt analog, d.h. die vom Benutzer aufgerufene Startfunktion ruft eine Template-Funktion `slssstart` mit den gleichen Template-Parametern wie `slssmain` auf, welche das System gemäß der in Abschnitt 4.1.1 beschriebenen Methode zur Lösung unter- und überbestimmter Systeme gegebenenfalls umstellt und dann die Hauptroutine des Lösers aufruft.

Als Beispiel für eine Startfunktion, welche die Funktion `slssstart` entsprechend ausprägt, wird in Listing 5.6 die Startfunktion für ein komplexes Punktsystem mit einer rechten Seite (die rechte Seite ist also vom Type `cvector`) angegeben.

```
1  void  slss ( const scmatrix& A, const cvector& b, civector& x, int& err,
2               slssconfig cfg ) {
3    cmatrix B(VecLen(b),1);
4    B[Col(1)] = b;
5    cimatrix X;
6    slssstart<scmatrix,cmatrix,cimatrix,scmatrix,cmatrix,scimatrix,cvector,
7              cvector,civector,complex,cdotprecision,cidotprecision,
8              scimatrix>
9              (A,B,X,err,cfg);
10   x = X[Col(1)];
11 }
```

Listing 5.6: Beispiel für eine Startfunktion des dünn besetzten Krawczyk-Lösers

Da die Hauptroutine des Lösers stets eine Matrix als rechte Seite erwartet, werden bei Aufruf des Lösers mit Vektoren als rechter Seite temporär entsprechende Matrizen mit einer Spalte angelegt, mit denen der Löser aufgerufen wird. Für Startfunktionen mit Matrizen als rechter Seite wird die Funktion `slss` hingegen direkt aufgerufen. Die verwendeten Template-Parameter unterscheiden sich dabei nicht. Die Parameter der Startfunktion haben die gleiche Bedeutung wie die der Hauptroutine.

Für Intervallsysteme müssen zunächst $\mathrm{mid}(\boldsymbol{A})$ und $\mathrm{mid}(\boldsymbol{b})$ berechnet werden, bei Punktsystemen hingegen ist diese Berechnung nicht erforderlich. Um für Punktsysteme nicht unnötig Rechenzeit und Speicherplatz zu verschwenden, wird daher wie folgt vorgegangen: Es wird mit Referenzen auf Punktmatrizen gearbeitet, welche für Intervallsysteme auf das berechnete Mittelpunktssystem verweisen, für Punktsysteme aber auf die original Systemmatrix und rechte Seite. Für Einzelheiten zur Implementierung siehe die Erläuterung in Abschnitt 4.1.2.

Einer der ersten Schritte im dünn besetzten Krawczyk-Löser ist die Berechnung der LU- bzw. Cholesky-Zerlegung der Systemmatrix A bzw. für Intervallsysteme der Mittelpunktsmatrix $\operatorname{mid}(\boldsymbol{A})$. Hierzu werden im Allgemeinen die Routinen aus SuiteSparse (siehe Abschnitt 2.4.4), also UMFPACK für die LU-Zerlegung bzw. CHOLMOD für die Cholesky-Zerlegung, mittels der in Abschnitt 5.2.1 beschriebenen Schnittstellen verwendet. Die gesuchte Zerlegung lässt sich also mit einem einfachen Funktionsaufruf berechnen. Sowohl UMFPACK als auch CHOLMOD können BLAS-Routinen zur Beschleunigung der Berechnung nutzen. Bei Benutzung geeigneter BLAS-Bibliotheken wird somit auch ein Teil der Berechnungen ohne weitere besondere Vorkehrungen parallel durchgeführt.

Für Bandsysteme kann alternativ eine hochgenaue LU-Zerlegung mittels des langen Akkumulators verwendet werden. Hierbei wird Crouts Algorithmus für die LU-Zerlegung genutzt. Bei Berechnung mit dem Akkumulator kann so neben L und U auch $[LU - A]$ mit geringem Mehraufwand direkt mitberechnet werden. Die Zerlegung wird außerdem so vorgenommen, dass die untere bzw. obere Bandbreite von L und U der jeweiligen Bandbreite der Systemmatrix A entspricht. Für Details zur Implementierung siehe [83].

Die entsprechende Funktion speichert die Bandmatrizen als dicht besetzte Matrizen, wobei jedes Band der Originalmatrix einer Spalte der übergebenen Matrix entspricht. Vor Aufruf der Funktion wird die dünn besetzte Systemmatrix daher in eine passende dicht besetzte Matrix konvertiert. Nach Abschluss der Funktion werden die als dicht besetzte Matrizen gespeicherten Faktoren L und U wiederum in dünn besetzte Matrizen konvertiert.

Analog dazu steht auch eine Funktion für die Cholesky-Zerlegung bereit. Diese wird für symmetrisch positiv definite Bandsysteme eingesetzt und spart etwas Rechenzeit gegenüber der LU-Zerlegung. Ansonsten entspricht diese Funktion der Funktion zur LU-Zerlegung, d.h. auch hier wird die Matrix $[LU - A]$ (mit $U = L^T$) bereits während der Zerlegung bestimmt.

Die Berechnung selbst ist aufgrund der Nutzung des langen Akkumulators deutlich aufwändiger als eine Zerlegung mittels SuiteSparse. Allerdings bietet diese Methode auch einige Vorteile. Eine explizite Berechnung der Matrix $[LU - A]$ ist nicht mehr nötig. Ebenso wächst die Bandbreite von L und U nicht an, wodurch im Verifikationsschritt bei Verwendung von Parallelepipeden einiges an Aufwand gespart werden kann. Weiterhin sind die berechneten Faktoren L und U in der Regel deutlich genauer als die mit SuiteSparse berechneten Faktoren, wodurch in manchen Fällen die Verifikation schneller zum Abschluss gebracht und somit weiterer Aufwand gespart werden kann. Für Bandmatrizen wird daher standardmäßig diese angepasste Methode verwendet.

Wird die Zerlegung der Systemmatrix hingegen mit SuiteSparse durchgeführt, so muss anschließend $[LU - A]$ explizit berechnet werden. Dazu ist es erforderlich, unabhängig vom Typ der Systemmatrix, eine Einschließung des Produktes LU zu bestimmen, wobei L und U stets Punktmatrizen sind. Analog zum Vorgehen in Abschnitt 3.2.1 kann eine solche Einschließung über eine Manipulation des Rundungsmodus erreicht werden. LU wird also einmal mit Rundung nach unten und einmal mit Rundung nach oben berechnet, um eine verlässliche Einschließung zu erhalten.

Je nach der Besetztheitsstruktur von L und U kann diese Berechnung sehr aufwändig sein. In der Regel wird das Produkt LU deutlich dichter besetzt sein als A. Eine Parallelisierung an dieser Stelle ist daher sinnvoll. Da es sich um dünn besetzte Matrizen handelt, ist eine Parallelisierung allerdings nicht so einfach wie im dicht besetzten Fall. Insbesondere ist eine gleichmäßige Aufteilung des Produktes in Blöcke fester Größe, die auf die verfügbaren Threads aufgeteilt werden, oftmals nicht optimal, da die Besetztheitsstruktur und somit der nötige Rechenaufwand sehr ungleichmäßig sein kann. Um dieses Problem zu umgehen, wird hier mit dem dynamischen Scheduling von OpenMP gearbeitet. Listing 5.7 zeigt Ausschnitte aus einer der Funktionen zur Berechnung von $LU - A$.

```
1  template<class TF, class TPoint>
2  void LUMinusAMain(const TF& L, const TF& U, const TF& A,
3                    TF& D_inf, TF& D_sup) {
4      int n = RowLen(L);
5      D_inf = TF(n,n);
6      D_sup = TF(n,n);
7
8      #ifdef _OPENMP
9        std::vector<int>& Di_ind = D_inf.row_indices();
10       std::vector<int>& Di_p   = D_inf.column_pointers();
11       std::vector<TPoint>& Di_x  = D_inf.values();
12       std::vector<int>& Ds_ind = D_sup.row_indices();
13       std::vector<int>& Ds_p   = D_sup.column_pointers();
14       std::vector<TPoint>& Ds_x  = D_sup.values();
15
16       const int chunk = determine_chunk_size(n,
17                                   omp_get_num_threads(),
18                                   U.column_pointers());
19       std::vector<TF> D_p((n/chunk)+1);
20       std::vector<int> startpos;
21
22       #pragma omp parallel default(shared)
23       {
24
25         fesetround(FE_UPWARD);
26
27         #pragma omp for schedule(dynamic,1)
28         for(int i=0 ; i<n/chunk ; i++) {
29           LUMinusAChunk(L,U,A,D_p[i],i*chunk+1,(i+1)*chunk);
30           D_p[i].dropzeros();
31         }
32
33         #pragma omp barrier
34
35         #pragma omp single
36         {
37           if(n%chunk != 0) {
38             LUMinusAChunk(L,U,A,D_p[n/chunk],(n/chunk)*chunk+1,n);
39             D_p[n/chunk].dropzeros();
40           }
```

```
41
42                  startpos = std::vector<int>(D_p.size());
43                  int nnzsum=0;
44                  for(unsigned int i=0 ; i<D_p.size() ; i++) {
45                    startpos[i] = nnzsum;
46                    nnzsum += D_p[i].get_nnz();
47                  }
48
49                  Ds_ind = std::vector<int>(nnzsum);
50                  Ds_x = std::vector<TPoint>(nnzsum);
51                  Ds_p[0] = 0;
52                  Ds_p[n] = nnzsum;
53                }
54
55  //...
```

Listing 5.7: Ausschnitt aus der Template-Funktion zur Berechnung von $LU - A$ für Punktmatrizen

Hierbei handelt es sich um eine Template-Funktion, welche für die Fälle verwendet wird, in denen A eine Punktmatrix ist. Der Template-Parameter `TF` steht für den Typ der dünn besetzten Punktmatrizen (also `srmatrix` oder `scmatrix`), `TPoint` steht für den Typ eines Elements der Matrizen (`real` oder `complex`). Die Parameter der Funktion haben folgende Bedeutung:

- `L`, `U` und `A`: Die Matrizen mit denen die Berechnung $[LU - A]$ durchgeführt werden soll.
- `D_inf` und `D_sup`: Berechnete Unter- und Oberschranke von $LU - A$.

Die Grundidee bei der Berechnung ist, die Ergebnismatrix in vertikale Streifen der Länge c einzuteilen, die unabhängig voneinander berechnet werden können. Die Berechnung dieser Teile erfolgt in einer parallelisierten for-Schleife. Im i-ten Schleifendurchlauf für $i = 1, \ldots, c$ wird also

$$LU_{1:n,(i-1)c+1:ic} - A_{1:n,(i-1)c+1:ic}$$

berechnet. Statt mit Teilmatrizen über den `operator()` zu arbeiten (siehe Abschnitt 3.3), wird für die eigentliche Berechnung jedes vertikalen Blocks der Ergebnismatrix die Funktion `LUMinusAChunk` aufgerufen. Diese berechnet den vertikalen Block von $LU - A$, welcher durch zwei Funktionsparameter beschrieben wird. Das Vorgehen bei der Berechnung entspricht dabei den Algorithmen aus Abschnitt 3.3. Hauptgrund für eine Vermeidung der Nutzung von Teilmatrizen mittels des C-XSC Operators ist hierbei nicht die Laufzeit, sondern der dadurch ansteigende Speicherbedarf. Da bei Ausschneiden einer Teilmatrix aus einer größeren Matrix wie in Abschnitt 3.3 beschrieben stets eine Kopie der Teilmatrix erstellt wird, kann es bei Verwendung sehr vieler Threads schnell zu Speicherengpässen kommen. Durch die direkte Berechnung mittels der Funktion `LUMinusAChunk` wird dies vermieden.

Die Ergebnismatrix für jeden Schleifendurchlauf wird jeweils in einem STL-Vektor zwischengespeichert. Da die Teilmatrizen bei der Berechnung von $LU - A$ oftmals explizit

gespeicherte Nullen enthalten können, werden diese nach der Berechnung über die Methode `dropzeros` entfernt, um Speicher (und im weiteren Verlauf auch Rechenaufwand) zu sparen.

Ist nun die Matrix A ungleichmäßig besetzt, so kann sich der Aufwand für die einzelnen Schleifendurchläufe stark unterscheiden. Um dies abzufangen, wird für die Parallelisierung das dynamische Scheduling von OpenMP verwendet. Hierdurch werden die einzelnen Schleifendurchläufe dynamisch auf die verfügbaren Threads verteilt, d.h. der nächste abzuarbeitende Durchlauf wird stets dem nächsten verfügbaren Thread zugeordnet, so dass z.B. ein Thread länger an einem Schleifendurchlauf arbeiten kann, während die anderen Threads sich um die restlichen Durchläufe kümmern.

Nach Abschluss der for-Schleife muss zunächst, falls n nicht ohne Rest durch c teilbar ist, ein letzter Teil der Ergebnismatrix berechnet werden. Danach müssen die Ergebnismatrizen für die einzelnen Teilberechnungen zu der Gesamtergebnismatrix zusammengefasst werden. Um eine Parallelisierung zu ermöglichen, wird zunächst die Zahl der Nicht-Nullen der Ergebnismatrix von einem einzelnen Thread in einem `single` Block bestimmt. Das eigentliche Umkopieren kann dann parallel ausgeführt werden.

Für die Wahl der Blockgröße ist entscheidend, den Aufwand pro Schleifendurchlauf so hoch zu halten, dass der Overhead durch die Parallelisierung nicht zu sehr ins Gewicht fällt. Andererseits sollte aber nicht einem Thread ein zu großer Anteil der Berechnung zugeteilt werden. Die Blockgröße wird daher für ein Produkt LU nach folgender Heuristik bestimmt:

1. Bestimme $s = \mathrm{nnz}(U)/(10 \cdot t)$, wobei t die Anzahl der verfügbaren Threads ist. Bei gleicher Verteilung der Nicht-Nullen soll jeder Thread für zehn gleich große Blöcke von U die Berechnung durchführen.

2. Stelle fest, was die kleinste Anzahl c von Spalten von U ist, die mindestens s Nicht-Nullen enthält.

3. Ist $c < 50$, so setze $c = 50$.

Listing 5.8 zeigt die Implementierung. Die Funktion bekommt die Dimension n der Matrix, die Anzahl der Threads sowie das Array mit Indizes der Spaltenanfänge von U übergeben.

```
1  int determine_chunk_size(int n, int threads,
2                           const std::vector<int>& p) {
3      if(threads == 1) return n;
4
5      int nnz = p[n];
6      int size = nnz / (10*threads);
7      int chunk = n;
8      int cs = 0;
9      int tmp = 0;
10
11     for(int i=0 ; i<n ; i++) {
12         cs += p[i+1]-p[i];
13         tmp++;
```

```
14            if(cs >= size) {
15                if(tmp < chunk)
16                    chunk = tmp;
17                tmp = 0;
18                cs = 0;
19            }
20        }
21
22        if(chunk < 50)
23            chunk = 50;
24
25        return chunk;
26    }
```

Listing 5.8: Funktion zur Berechnung der Blockgröße für parallele dünn besetzte Matrixprodukte

Dieses Vorgehen sorgt dafür, dass die Arbeitslast in der Regel gut auf die einzelnen Threads verteilt ist. Die Begrenzung auf ein Minimum von $c = 50$ verhindert, dass der Aufwand für die Verwaltung der Threads zu großen Einfluss auf die Laufzeit hat. Wie die Testergebnisse in Abschnitt 5.3 zeigen, sorgt die verwendete Heuristik in der Regel für eine effiziente Parallelisierung.

Zu beachten ist, dass das beschriebene Vorgehen zur Berechnung von $LU - A$ hier zwei mal durchgeführt werden muss, einmal mit Rundung nach oben und einmal mit Rundung nach unten. Für Intervallmatrizen wird eine leicht angepasste Version verwendet, die notwendigen Anpassungen ändern aber nichts Wesentliches am beschriebenen Vorgehen.

Für die schnelle Berechnung von $LU - A$ gemäß Satz 5.3 muss nur einmal das Produkt $|L||U|$ mit Rundung zur nächsten Gleitkommazahl berechnet werden. Für diesen Fall wird daher eine eigene Version der angegebenen Funktion verwendet, welche die Berechnung entsprechend anpasst und nur einmal durchführt.

Ein ähnliches Vorgehen erfolgt im Verifikationsschritt. Hier muss in jedem Schritt vor der Lösung der Dreieckssysteme zunächst die rechte Seite von

$$LU\boldsymbol{y}_{i+1} = \boldsymbol{z} + [LU - A]\boldsymbol{y}_i$$

berechnet werden. Je nach Besetztheit von $[LU - A]$ kann diese Operation recht aufwändig sein. Daher wird eine Parallelisierung ähnlich zum Vorgehen bei der Berechnung der Iterationsmatrix durchgeführt. Auch hier wird der zu berechnende Ausdruck auf Teile der Blockgröße c aufgespalten, welche über eine for-Schleife mit dynamischem Scheduling auf die verfügbaren Threads verteilt wird. Da es sich bei den Vektoren um dicht besetzte Vektoren handelt, kann das Ergebnis direkt in einem passenden Vektorobjekt gespeichert werden. Falls n nicht ohne Rest durch c teilbar ist, wird der verbleibende Teil der Berechnung wiederum von einem einzelnen Thread durchgeführt.

Ein letzter wesentlicher Punkt - neben der noch folgenden Beschreibung der Implementierung der Lösungsmethoden für die auftretenden Dreieckssysteme - ist die Berechnung der Näherungslösung und des Residuums. Hierbei gibt es zwei Möglichkeiten, einmal die einfache Berechnung der Näherungslösung mit anschließender Bestimmung des Residuums unter direkter Verwendung der C-XSC Operatoren und einmal die Berechnung der

Näherungslösung in einer Staggered-artigen Darstellung mit hochgenauer Berechnung des Residuums über den langen Akkumulator. Listing 5.9 zeigt zunächst die einfache Berechnungsmethode.

```
1 TVec xapp(n);
2 TVec h(bm[Col(s)]);
3 lowtrisolve(L,h,d);
4 uptrisolve(U,d,xapp);
5 xx = xapp;
6 opdotprec = cfg.precResidual;
7 zz = b - A*xx;
```

Listing 5.9: Einfache Berechnung von Näherungslösung und Residuum

Wie im dicht besetzten Fall werden auch beim hier beschriebenen Löser mehrere rechte Seiten unterstützt. Dazu läuft eine for-Schleife mit Schleifenvariable `s` über alle rechte Seiten. In der zweiten Zeile wird also die `s`-te rechte Seite verwendet. Danach werden die beiden Dreieckssysteme mit normaler Gleitkommarechnung gelöst. Das Residuum wird schließlich direkt über C-XSC Operatoren berechnet, wobei die Präzision für die Berechnung des Matrix-Vektor-Produktes gemäß der gewählten Konfiguration gesetzt wird (standardmäßig wird 2, also doppelte `double`-Präzision genutzt). Für die genauere Berechnung wird gemäß Listing 5.10 vorgegangen.

```
1  int prec = (cfg.apprxSolutionStagPrec > 0) ? cfg.apprxSolutionStagPrec : 1;
2  TVec xapp(n);
3  TMidRhs xappstag(n,prec);
4  //...
5  TVec h(n);
6  d = bm[Col(s)];
7
8  for(int p=1 ; p<prec ; p++) {
9    lowtrisolve(L,d,h);
10   uptrisolve(U,h,xapp);
11   xappstag[Col(p)] = xapp;
12   residual(d,bm[Col(s)],Am,xappstag,p);
13 }
14
15 lowtrisolve(L,d,h);
16 uptrisolve(U,h,xapp);
17 xappstag[Col(prec)] = xapp;
18 residual(zz,b,A,xappstag,prec);
```

Listing 5.10: Hochgenaue Berechnung von Näherungslösung und Residuum

Die Näherungslösung wird hier als (nicht ausgewertete) Summe von Vektoren bestimmt. Diese werden als Spalten einer dicht besetzten Matrix (`xappstag`) gespeichert. Hierzu werden, je nach in der Konfiguration gewählter Präzision, die Dreieckssysteme mehrfach gelöst, wobei in jedem Durchlauf die neue rechte Seite das Residuum der bisher bestimmten Näherungslösung ist. Die Funktion `residual` berechnet hierzu mit Hilfe des langen Akkumulators am Ende des p-ten Durchlaufs den Ausdruck $b - \sum_{i=1}^{p} A\tilde{x}_i$, wobei $\tilde{x}_i$ für die bisher bestimmten Näherungslösungen steht. Durch Verwendung des langen Akkumulators wird das Residuum bis auf eine Rundung genau bestimmt.

Listing 5.11 zeigt zur Verdeutlichung die einfachste Version der Funktion `residual` für reelle Punktdatentypen. Die entsprechenden Versionen für die anderen vorkommenden Datentypen ergeben sich jeweils im Wesentlichen durch Anpassungen der verwendeten Datentypen.

```
1  void residual(rvector& def, const rvector& b, const srmatrix& A,
2                const rmatrix& xapp, int cols) {
3    int n = VecLen(b);
4    int lb = Lb(xapp,COL);
5
6    std::vector<dotprecision> res;
7    res.reserve(n);
8    #ifdef _OPENMP
9    std::vector<omp_lock_t> locks(n);
10   #endif
11
12   const std::vector<int>& cp   = A.column_pointers();
13   const std::vector<int>& ri   = A.row_indices();
14   const std::vector<real>& val = A.values();
15
16   for(int i=1 ; i<=n ; i++) {
17     res.push_back(dotprecision(b[i]));
18     #ifdef _OPENMP
19     omp_init_lock(&(locks[i-1]));
20     #endif
21   }
22
23   #ifdef _OPENMP
24   int t = omp_get_max_threads();
25   #pragma omp parallel for schedule(static,n/t)
26   #endif
27   for(int j=0 ; j<n ; j++) {
28     for(int k=cp[j] ; k<cp[j+1] ; k++) {
29       #ifdef _OPENMP
30       omp_set_lock(&(locks[ri[k]]));
31       #endif
32       for(int c=1 ; c<=cols ; c++) {
33           accumulate(res[ri[k]], -val[k], xapp[j+1][c-1+lb]);
34       }
35       #ifdef _OPENMP
36       omp_unset_lock(&(locks[ri[k]]));
37       #endif
38     }
39   }
40
41   #ifdef _OPENMP
42   #pragma omp parallel for
43   #endif
44   for(int i=1 ; i<=n ; i++)
```

```
45      def[i] = rnd(res[i-1],RND_UP);
```

Listing 5.11: Funktion zur Berechnung des Residuums bei Staggered-artiger Näherungslösung

Die Parameter dieser Funktion haben folgende Bedeutung:

- `def`: Das nach Ablauf der Funktion berechnete Residuum.
- `A` und `b`: Systemmatrix und rechte Seite des zu lösenden Systems.
- `xapp`: Dicht besetzte Matrix, deren Spalten die Summanden der berechneten Näherungslösung sind.
- `cols`: Anzahl der bisher berechneten Summanden der Näherungslösung, also die Anzahl der bereits belegten Spalten von `xapp`.

Da es sich bei `A` um eine dünn besetzte Matrix im CCS-Format handelt, ist es sinnvoll, bei der Berechnung `A` spaltenweise zu durchlaufen. Daher wird zunächst für jede Komponente des Ergebnisvektors ein Akkumulator angelegt und mit b_i initialisiert. Danach wird `A` spaltenweise durchlaufen und die Beiträge zum jeweiligen Element des Ergebnisvektors werden, unter Berücksichtigung aller Summanden der Näherungslösung (also der belegten Spalten von `xapp`), zum zugehörigen Akkumulator hinzu addiert.

Die einzelnen Durchläufe der for-Schleife über eine Spalte von `A` sind unabhängig. Eine direkte Parallelisierung der Schleife wäre aber in der Regel wenig effektiv, da meist sehr viele Spalten der Matrix nur sehr wenige Elemente enthalten und der Verwaltungsaufwand für die Parallelisierung somit überwiegen würde. Ein besserer Ansatz ist daher, die äußere for-Schleife zu parallelisieren.

Um zu verhindern, dass dabei parallele Zugriffe auf den gleichen Akkumulator aus dem STL-Vektor `res` erfolgen, wird der jeweils aktuell von einem Thread verwendete Akkumulator mit einem OpenMP-`lock` (also einem Mutex) blockiert, bis die jeweilige Berechnung abgeschlossen ist. Durch die dünne Besetztheit von `A` und die statische Aufteilung der Schleifeniterationen auf n/t gleich große Blöcke (t ist die Anzahl der verwendeten Threads) ist der Laufzeitverlust durch diese Form der Synchronisierung in der Regel wesentlich geringer als der durch die Parallelisierung der äußeren Schleife erreichte Performance-Gewinn.

Zum Abschluss werden die in den Akkumulatoren gespeicherten (exakten) Ergebnisse gerundet und im Vektor `def` gespeichert. Auch diese Schleife kann parallelisiert werden. Neben der hier gezeigten Version sind auch für die abschließende Residuumsberechnung jeweils Versionen implementiert, bei denen am Ende eine Einschließung des Residuums berechnet wird. Der Vektor `def` ist dann also ein Intervallvektor.

Bevor nun genauer auf die Implementierung der Methoden zur Lösung der auftretenden Dreieckssysteme eingegangen wird, sollen noch kurz einige Punkte angesprochen werden, die im allgemeinen dicht besetzten Löser aus Abschnitt 4.1.2 implementiert wurden, deren Implementierung für den dünn besetzten Krawczyk-Löser aber nicht sinnvoll ist, obwohl er im Kern den gleichen Algorithmus verwendet:

- **Inneneinschließungen**: Im dicht besetzten Fall ist die Berechnung einer Inneneinschließung der Lösungsmenge vergleichsweise günstig, da hierzu ein leicht umgestellter Verifikationsschritt mit einer Innenrundung von $\boldsymbol{z}$ durchgeführt wird. Im dünn besetzten Fall ist aber zum einen ein solcher Schritt deutlich teurer, zum anderen ist eine sinnvolle Innenrundung nur mit hohem Aufwand möglich, da dann die Methoden zur Lösung der auftretenden Dreieckssysteme entsprechend angepasst werden müssten. Eine entsprechende Implementierung erscheint daher nicht sinnvoll.

- **Nachiteration**: Mit der Nachiteration kann im dicht besetzten Fall durch eine Fortführung der Verifikationsiteration nach einer erfolgreichen Einschließung ins Innere teilweise noch eine leichte Verbesserung des Ergebnisses erzielt werden. Aufgrund der hohen Kosten eines Verifikationsschrittes im dünn besetzten Fall und der ohnehin in der Regel sehr geringen Auswirkungen der Nachiteration wird auf eine Implementierung im dünn besetzten Krawczyk-Löser verzichtet.

- **Zweiter Teil mit Inverser doppelter Länge**: Der dicht besetzte Löser kann einen zweiten Algorithmus-Teil mit einer Inversen $R_1 + R_2$ durchführen, welcher auch bei sehr schlecht konditionierten Systemen noch zu einer Lösung führen kann. Im dünn besetzten Fall müsste eine LU-Zerlegung der Form $L_1U_1 + L_2U_2$ bestimmt und der Rest des Algorithmus angepasst werden. Aufgrund der beschränkten Anwendbarkeit des dünn besetzten Krawczyk-Lösers erscheint der dafür nötige Aufwand nicht gerechtfertigt.

- **Matrix Modus**: Im dicht besetzten Fall steht für Systeme mit mehreren rechten Seiten der sogenannte Matrix Modus zur Verfügung, welcher die rechten Seiten nicht spaltenweise durchläuft, sondern als Matrix behandelt und dadurch eine wesentliche Beschleunigung durch deutlich verbesserte Cache-Ausnutzung erzielt. Im dünn besetzten Fall macht dieser Modus keinen Sinn, da zum einen Cache-Effekte durch die Natur der verwendeten dünn besetzten Datenstrukturen eine wesentlich geringere Rolle spielen, zum anderen in der Regel auch nur wenige rechte Seiten verwendet werden sollten, da rechte Seite und Ergebnis als dicht besetzt gespeichert werden.

Lösung der auftretenden Dreieckssysteme

Im Folgenden wird die Implementierung der drei in Abschnitt 5.1 beschriebenen Methoden zur Lösung der im dünn besetzten Krawczyk-Löser während des Verifikationsschrittes auftretenden Dreieckssysteme mit Punkt-Systemmatrix und intervallwertiger rechter Seite beschrieben. Die Implementierung ist jeweils unabhängig vom Rest des Lösers, der bisher in diesem Abschnitt beschrieben wurde, d.h. der einzige Unterschied ist, dass entsprechend der gewählten Lösungsmethode eine passende Funktion für die Lösung der Dreieckssysteme aufgerufen wird. Es werden im Folgenden wiederum nur untere Dreieckssysteme betrachtet. Die entsprechende Implementierung für obere Dreieckssys-

teme erfolgt analog. Weiterhin werden zumeist Versionen für reelle Systeme angegeben, Versionen für komplexe Systeme sind jeweils mit wenigen Anpassungen möglich.

Zunächst wird der einfachste Fall, also die Lösung mittels einfacher Vorwärtssubstitution, betrachtet. Diese wurde als Template-Funktion implementiert, bei der der Typ der rechten Seite und des Lösungsvektors Template-Parameter sind. Dabei wurde für beide mögliche Typen von Systemmatrizen, also `srmatrix` und `scmatrix`, jeweils eigene Versionen erstellt, da im komplexen Fall einige kleine Anpassungen nötig sind, welche die Erstellung eines Gesamt-Templates erschwert hätten. Vor allem muss im Komplexen die Berechnung des Produktes direkt implementiert werden, da der Multiplikationsoperator für den Datentyp `complex` den Akkumulator verwendet und somit die Rechenzeit drastisch erhöhen würde. Listing 5.12 zeigt den Quellcode für den komplexen Fall.

```
1  template<typename TRhs, typename Tx>
2  void lowtrisolve(const scmatrix& L, const TRhs& b, Tx& x) {
3      int n = RowLen(L);
4      const vector<int>& ind      = L.row_indices();
5      const vector<int>& p        = L.column_pointers();
6      const vector<complex>& val  = L.values();
7
8      x = b;
9      int lb = Lb(x);
10
11     for(int j=0 ; j<n ; j++) {
12       x[j+lb] /= val[p[j]];
13
14       #ifdef _OPENMP
15       int d = p[j+1]-p[j];
16       int t = (d/50 > omp_get_max_threads()) ?
17                          omp_get_max_threads() : d/50;
18       #pragma omp parallel for if(d >= 100) num_threads(t)
19       #endif
20       for(int k=p[j]+1 ; k<p[j+1] ; k++) {
21         //x[ind[k]+lb] -= val[k] * x[j+lb];
22         SetRe(x[ind[k]+lb], Re(x[ind[k]+lb]) -
23                             (Re(val[k]) * Re(x[j+lb]) -
24                              Im(val[k]) * Im(x[j+lb])) );
25         SetIm(x[ind[k]+lb], Im(x[ind[k]+lb]) -
26                             (Im(val[k]) * Re(x[j+lb]) +
27                              Re(val[k]) * Im(x[j+lb])) );
28       }
29     }
30 }
```

Listing 5.12: Template-Funktion zur Lösung eines komplexen unteren Dreieckssystems mit Vorwärtssubstitution

Der Template-Parameter `TRhs` gibt den Typ der rechten Seite, der Template-Parameter `Tx` den Typ des Lösungsvektors an. Es kann sich also jeweils um einen der reellen C-XSC Vektortypen handeln. Die Parameter der Funktion stehen für die Systemmatrix `L`, die rechte Seite `b` und den Lösungsvektor `x`.

Es handelt sich im Wesentlichen um eine direkte Implementierung von Algorithmus

5.1. Diese Implementierung ist durch Nutzung der überladenen Operatoren von C-XSC sowohl für intervallwertige rechte Seiten als auch für Punktvektoren geeignet und wird jeweils für die entsprechenden Datentypen ausgeprägt. Die Ausprägungen dieser Funktion werden also auch dann verwendet, wenn ein Dreieckssystem mit einfacher Gleitkommarechnung gelöst werden soll, z.B. bei der Berechnung der Näherungslösung.

Die innere for-Schleife, welche über die Einträge der aktuell betrachteten Spalte von `L` läuft, kann parallelisiert werden. Um Overhead durch die Parallelisierung bei dünn besetzten Spalten von `L` zu vermeiden, wird pro 50 Nicht-Nullen in der jeweiligen Spalte ein Thread (falls verfügbar) verwendet und eine Parallelisierung unter Verwendung der `if`-Direktive von OpenMP nur dann durchgeführt, wenn die Anzahl der Nicht-Nullen in der aktuellen Zeile größer oder gleich 100 ist.

Als nächstes wird die Implementierung der Methode der impliziten Berechnung der Inversen gemäß Algorithmus 5.2 betrachtet. Da die Funktion nur für intervallwertige rechte Seiten benötigt wird, ist die Implementierung nicht Template-basiert. Es wird jeweils eine eigene Version für den reellen und den komplexen Fall verwendet (eine Template-Version würde hier wiederum aufgrund einiger kleinerer Unterschiede, wie u.a. der Umgehung des langsamen Multiplikationsoperators der Klasse `complex`, zu viel Aufwand ohne echten Nutzen bedeuten). Im Folgenden wird die reelle Version betrachtet. Listing 5.13 zeigt zunächst den Anfang der entsprechenden Funktion `lowtrisolve_inv`.

```
1  void lowtrisolve_inv(const cxsc::srmatrix& L, const cxsc::ivector& bo,
2                       cxsc::ivector& x) {
3      int n = RowLen(L);
4
5      rvector xapp(n);
6      ivector b(n);
7      real abssup, absinf;
8
9      const vector<int>& ind   = L.row_indices();
10     const vector<int>& p     = L.column_pointers();
11     const vector<real>& val  = L.values();
12
13     vector<sivector> rhs;
14
15     lowtrisolve(L,mid(bo),xapp);
16     b = bo - L*xapp;
17
18     for(int i=0 ; i<n ; i++) {
19       abssup = abs(Sup(b[i+1]));
20       absinf = abs(Inf(b[i+1]));
21       if(abssup > absinf)
22         SetInf(b[i+1], -abssup);
23       else if(absinf > abssup)
24         SetSup(b[i+1], absinf);
25
26       rhs.push_back(sivector(n));
27       rhs[i][i+1] = 1.0;
28     }
29
```

```
30      //...
```

Listing 5.13: Funktion zur Lösung eines unteren Dreieckssystems mit impliziter Berechnung der Inversen, Teil 1

Die der Funktion übergebenen Parameter entsprechen wieder der Systemmatrix `L`, der rechten Seite `bo` und dem Lösungsvektor `x`. Zu Beginn wird zunächst das Mittelpunktssystem gelöst und das Ergebnis von der rechten Seite abgezogen. Der so erhaltene Intervallvektor $\boldsymbol{b}$ ist bis auf Rundungsfehler symmetrisch. Danach wird er durch eine leichte Aufblähung der Form

$$\boldsymbol{b}_i = [-|b_i|, |b_i|]$$

symmetrisiert. Weiterhin wird ein STL-Vektor mit n dünn besetzten Vektoren angelegt, welche im Laufe der Berechnungen Einschließungen der Spalten von L^{-1} speichern. Initialisiert wird dieser zunächst mit der Einheitsmatrix. Listing 5.14 zeigt nun den zweiten Teil der Funktion mit der eigentlichen Berechnung.

```
1       //...
2
3       for(int j=0 ; j<n ; j++) {
4         fesetround(FE_UPWARD);
5         rhs[j] /= abs(val[p[j]]);
6         fesetround(FE_TONEAREST);
7         x[j+1] = xapp[j+1] + abs(rhs[j]) * b;
8
9         #ifdef _OPENMP
10        #pragma omp parallel
11        {
12        #endif
13
14          #ifdef _OPENMP
15          #pragma omp single
16          #endif
17          rhs[j] /= val[p[j]];
18
19          #ifdef _OPENMP
20          #pragma omp for schedule(dynamic,1)
21          #endif
22          for(int k=p[j]+1 ; k<p[j+1] ; k++)
23            rhs[ind[k]] += (-val[k]) * rhs[j];
24
25        #ifdef _OPENMP
26        }
27        #endif
28
29        Resize(rhs[j]);
30      }
31
32      rhs.clear();
33  }
```

Listing 5.14: Funktion zur Lösung eines unteren Dreieckssystems mit impliziter Berechnung der Inversen, Teil 2

Die Matrix `L` wird spaltenweise durchlaufen und die Matrix L^{-1} dabei zeilenweise aufgebaut. Zu Beginn jedes Schleifendurchlaufs wird

$$\texttt{rhs[j]}/l_{jj}$$

berechnet, also eine Einschließung für die j-te Zeile von L^{-1}. Deren Betrag wird dann mit der symmetrisierten rechten Seite multipliziert und zur zuvor berechneten Näherungslösung des Mittelpunktssystems hinzu addiert. Daraus ergibt sich die j-te Komponente des Ergebnisvektors.

Anschließend müssen obere und untere Schranke der nächsten Zeile von L^{-1} berechnet werden. Die beiden hier auftretenden for-Schleifen können parallelisiert werden. Dabei wird eine dynamische Verteilung der Schleifeniterationen verwendet, da die einzelnen Durchläufe in der Regel einen sehr unterschiedlichen Aufwand aufweisen. Durch den zumeist hohen Aufwand dieser Berechnungen (L^{-1} ist in der Regel dicht besetzt) ist diese Parallelisierung oftmals recht effektiv. Nach Abschluss dieser Berechnungen werden schließlich noch die Daten zur j-ten Zeile von L^{-1}, also der entsprechende dünn besetzte Vektor aus `rhs` gelöscht, da sie im weiteren Verlauf nicht mehr benötigt werden.

Als letztes folgt nun die Erläuterung der Implementierung der Vorwärtssubstitution mittels Parallelepipeden gemäß Algorithmus 5.3. Dazu sind zunächst Routinen für die QR-Zerlegung und die Berechnung der Inversen der (nahezu) orthogonalen bzw. unitären Basismatrix nötig. Listing 5.15 zeigt die Routine zur Berechnung der QR-Zerlegung einer reellen Matrix mit LAPACK-Unterstützung.

```
1  inline void QR( rmatrix& A, rmatrix& Q, int& err) {
2  #ifdef CXSC_USE_LAPACK
3    Q = transp(A);
4    int m = ColLen(A);
5    int n = RowLen(A);
6    int lda = m;
7    int info;
8    int lwork = 256*n;
9    double* tau = new double[n];
10   double* work = new double[lwork];
11   double* DA = Q.to_blas_array();
12
13   dgeqrf_(&m, &n, DA, &lda, tau, work, &lwork, &info);
14
15   if(info != 0) {
16     err = 1;
17     delete[] work;
18     delete[] tau;
19     return;
20   }
21
22   dorgqr_(&m, &n , &n, DA, &lda, tau, work, &lwork, &info);
23
24   if(info != 0) {
25     err = 1;
26     delete[] work;
27     delete[] tau;
```

```
28        return;
29     }
30
31     Q = transp(Q);
32
33     delete[] work;
34     delete[] tau;
35
36     err = 0;
37
38  #else
39     //... C-XSC Implementierung
40  }
```

Listing 5.15: Funktion zur näherungsweisen Berechnung der QR-Zerlegung (LAPACK-Version)

Es wird nur die Berechnung mittels LAPACK gezeigt. Die alternative C-XSC Implementierung berechnet Q mittels Householder-Matrizen. Sie entspricht der Version aus [83], ist jedoch mit OpenMP parallelisiert.

Die hier gezeigte Version verwendet zunächst die LAPACK-Funktion `dgetqrf` zur eigentlichen Berechnung der QR-Zerlegung. Dazu wird `Q` mit der Transponierten von `A` (da LAPACK von spaltenorientiert gespeicherten Matrizen ausgeht) vorbelegt. Die Funktion `dgetqrf` arbeitet dann direkt mit den Daten von `Q` und überschreibt die Matrix. LAPACK speichert die Daten aus der QR-Zerlegung zunächst in einem internen Format. Um die tatsächliche Matrix `Q` zu erhalten, muss daher noch die Funktion `dorgqr` aufgerufen werden. Abschließend muss `Q` transponiert werden, um für die weiteren Berechnungen als eine zeilenorientiert gespeicherte Matrix zur Verfügung zu stehen.

Trotz des zusätzlichen Aufwands für die Transponierung ist die LAPACK-Version auch für kleine Matrizen (also kleine Bandbreiten des Dreieckssystems) meist deutlich schneller als die direkte C-XSC Implementierung.

Um die Inverse der nahezu orthogonalen bzw. unitären Matrix `Q` zu berechnen, wird die Funktion `QR_inv` verwendet. Hierbei handelt es sich um eine an die Pascal-XSC-Version aus [83] angelehnte Implementierung, welche im Prinzip direkt die Methode aus Satz 5.2 umsetzt. Ähnliches gilt für die Funktion `QR_sorted`, welche wie in Abschnitt 5.1 besprochen die Spalten von `A` vor der QR-Zerlegung entsprechend der Länge der Ränder der vom betrachteten Parallelepiped eingeschlossenen Menge sortiert und erst anschließend die Funktion `QR` zur Durchführung der eigentlichen QR-Zerlegung aufruft. Unterschiede zur Implementierung aus [53] bestehen vor allem in einer wesentlich sparsameren Verwendung des langen Akkumulators, wodurch die Berechnungen etwas beschleunigt werden. Aufgrund der trotzdem geringen Unterschiede zu den Implementierungen aus [83] wird hier auf eine weitere Besprechung dieser Funktionen verzichtet.

Mit Hilfe der Funktionen `QR`, `QR_sorted` und `QR_inv` kann nun die eigentliche Funktion zur Lösung eines unteren Dreieckssystems mit Hilfe von Parallelepipeden formuliert werden. Diese wird wieder als Template-Funktion implementiert, welche von Startfunktionen für die relevanten Datentypen ausgeprägt wird. Listing 5.16 zeigt zunächst den Kopf und die Initialisierungen in dieser Template-Funktion.

```
1  template<typename TMat, typename TVec, typename TIVec, typename TSVec,
2           typename TSIVec, typename TDenseMat, typename TDenseIMat,
3           typename TPoint, typename TInterval>
4  void lowtrisolve_band_main( TMat& A, TMat& At, TIVec& b, TIVec& x,
5                              int l, TDenseIMat& Basis, bool baseStore) {
6    int n = RowLen(A);
7    TIVec xtmp(l);
8
9    x = 0.0;
10   xtmp = 0.0;
11
12   if(l == 0) {
13     for(int i=1 ; i<=n ; i++)
14       x[i] = b[i] / A(i,i);
15     return;
16   } else if (l == 1) {
17     lowtrisolve(A,b,x);
18     return;
19   }
20
21   //Starting values: Compute x[1] to x[l] by forward substitution
22   TIVec xstart(l);
23   lowtrisolve_inv(A(1,l,1,l),b(1,l),xstart);
24   x(1,l) = xstart;
25   xtmp = xstart;
26
27   //Compute remaining x by forward substitution using coordinate
28   //transformation
29   TDenseMat B0(l,l), B1(l,l);
30   TDenseIMat B1_inv(l,l);
31   TSIVec bi(l);
32   int err=0;
33
34   B0 = Id(B0);
35
36   //...
37 }
```

Listing 5.16: Funktion zur Lösung eines unteren Dreiecks-Bandsystems: Kopf und Initialisierung

Die Template-Parameter haben dabei folgende Bedeutung:

- `TMat`: Typ der Systemmatrix, also `srmatrix` oder `scmatrix`.
- `TVec`: Typ eines passenden dicht besetzten Punktvektors, also `rvector` oder `cvector`.
- `TIVec`: Typ eines passenden dicht besetzten Intervallvektors, also `ivector` oder `civector`.
- `TSVec`: Typ eines passenden dünn besetzten Punktvektors, also `srvector` oder `scvector`.

- `TSIVec`: Typ eines passenden dünn besetzten Intervallvektors, also `sivector` oder `scivector`.
- `TDenseMat`: Typ für eine passende dicht besetzte Punktmatrix, als `rmatrix` oder `cmatrix`.
- `TDenseIMat`: Typ für eine passende dicht besetzte Intervallmatrix, also `imatrix` oder `cimatrix`.
- `TPoint`: Typ für ein Punkt-Element, also `real` oder `complex`.
- `TInterval`: Typ für ein intervallwertiges Element, also `interval` oder `cinterval`.

Die Parameter der Funktion haben folgende Bedeutung:

- `A`: Die Systemmatrix.
- `At`: Die Transponierte der Systemmatrix. Diese wird an mehreren Stellen gebraucht und muss so nur einmal vorab bestimmt werden.
- `b`: Die rechte Seite des Systems.
- `x`: Nach erfolgreichem Ablauf der Funktion eine Einschließung der exakten Lösung.
- `l`: Die untere Bandbreite des Systems.
- `Basis`: Eine dicht besetzte Matrix zur Speicherung der Basismatrizen. Diese kann optional aktiviert werden, eine genauere Beschreibung folgt im Laufe dieses Abschnitts.
- `baseStore`: Gibt an, ob die Speicherung der Basismatrizen aktiviert ist oder nicht.

Zu Beginn der Funktion wird zunächst abgefragt, ob die Bandbreite des Systems 1 oder 0 ist. Wie in Abschnitt 5.1 gesehen, kann in diesen beiden Fällen das System stets mit einfacher Vorwärtssubstitution gelöst werden. Ist die Bandbreite größer, so müssen zunächst die Startwerte berechnet werden (bei Bandbreite m also die ersten m Komponenten der Lösung). Um zu vermeiden, dass es dabei bereits zu Überschätzungen kommt, wird hierzu die Methode mit Berechnung der impliziten Inversen verwendet. Danach folgen Initialisierungen der für die folgenden Berechnungen nötigen Variablen. Die eigentliche Berechnung zeigt Listing 5.17.

```
1   //...
2
3   for(int i=l+1 ; i<=n ; i++) {
4     TDenseMat Ai_mid(l,l);
5     TDenseIMat Ai(l,l);
6
7     //Aufbau der Matrix Ai...
8     TSVec row(At[Col(i)]);
9     vector<int>& rp = row.row_indices();
```

```
10      vector<TPoint>& rx = row.values();
11
12      Ai = 0.0;
13      for(int k=1 ; k<l ; k++)
14        Ai[k][k+1] = 1.0;
15      for(unsigned int j=0 ; j<rp.size()-1 ; j++) {
16        Ai[l][rp[j]+1-(i-(l+1))] = TInterval(-rx[j]) / rx[rp.size()-1];
17      }
18
19      //Berechnung...
20      bi[l] = b[i] / rx[rp.size()-1];
21
22      Ai = Ai * B0;
23      Ai_mid = Inf(Ai) + 0.5 * (Sup(Ai) - Inf(Ai));
24
25      QR_sorted(Ai_mid, xtmp, B1);
26      QR_inv(B1, B1_inv, err);
27
28      if (err != 0) return;
29
30      xtmp = (B1_inv * Ai) * xtmp + B1_inv * bi;
31      x[i] = B1[l] * xtmp;
32      B0 = B1;
33    }
34
35  }
```

Listing 5.17: Funktion zur Lösung eines unteren Dreiecks-Bandsystems: Vorwärtssubstitution mit Parallelepipeden

Dieser Quellcode ist im Wesentlichen eine direkte Umsetzung von Algorithmus 5.3. Die dort als $L^{(i)}$ bezeichneten Matrizen entsprechen hier der Variablen `Ai`, welche zu Beginn aufgebaut werden muss. Dazu wird die i-te Spalte der Transponierten der Systemmatrix ausgelesen (dies ist aufgrund der spaltenorientierten Datenstruktur deutlich günstiger, als die i-te Zeile direkt aus der Systemmatrix zu übernehmen) und für die Berechnung der letzten Zeile von `Ai` verwendet. Daraufhin folgt die eigentliche Berechnung unter Benutzung der bereits beschriebenen Funktionen `QR_sorted` und `QR_inv`. Zu beachten ist, dass bei aktivierter BLAS-Unterstützung die Matrix-Matrix-Produkte, welche neben der QR-Zerlegung den größten Teil des Aufwands ausmachen, hier mittels BLAS ausgeführt werden.

Allgemein machen die Bestimmung der Basismatrix, ihrer Inversen sowie der Matrix-Matrix-Produkte $(B^{(i+1)})^{-1}B^{(i)}A^{(i)}$ den bei weitem größten Teil des Aufwands dieser Funktion aus. Diese Matrizen sind im Allgemeinen während der Verifikation bei jeder Lösung der Dreieckssysteme gleich, denn die Sortierung der Spalten vor der QR-Zerlegung ändert sich in den meisten Fällen nicht, da sich die rechte Seite bei der Benutzung des Krawczyk-Operators in der Regel nur wenig verändert. Durch eine Speicherung der Matrizen $B^{(i)}$, $[(B^{(i)})^{-1}]$ und $[(B^{(i)})^{-1}A^{(i)}B^{(i)}]$ bei der ersten Lösung des Dreieckssystems können diese also für alle weiteren Verifikationsschritte wiederverwendet werden, wodurch die folgenden Berechnungen der Lösungen der Dreieckssysteme deutlich be-

schleunigt werden. Dieses Vorgehen wird bei Aktivierung der Option `storeBase` in der Konfiguration verwendet.

Dazu wird im Löser eine dicht besetzte Intervallmatrix der Dimension $3m \times m(n-m)$ angelegt, wobei m die Bandbreite des Dreieckssystems angibt. Diese Matrix wird der Funktion zum Lösen des Dreieckssystems übergeben (Parameter `Basis`). Beim ersten Aufruf der Funktion werden die in jedem Iterationsschritt auftretenden Matrizen in dieser Matrix gespeichert und in allen folgenden Aufrufen für die Berechnungen wiederverwendet. Für den Löser bedeutet dies, dass die Verifikation ab dem zweiten Iterationsschritt deutlich beschleunigt wird. Die Berechnung aus Listing 5.17 vereinfacht sich dann wie in Listing 5.18 zu sehen.

```
1   //...
2
3   const vector<int>& p = A.column_pointers();
4   const vector<TPoint>& rx = A.values();
5
6   for(int i=l+1 ; i<=n ; i++) {
7     bi[l] = b[i] / rx[p[i-1]];
8
9     xtmp = Basis(1,l,(i-l-1)*l+1,(i-l)*l) * xtmp +
10           Basis(l+1,2*l,(i-l-1)*l+1,(i-l)*l) * bi;
11    TDenseIMat B = Basis(2*l+1,3*l,(i-l-1)*l+1,(i-l)*l);
12    x[i] = B[Ub(B,ROW)] * xtmp;
13  }
14
15  //...
```

Listing 5.18: Funktion zur Lösung eines unteren Dreiecks-Bandsystems: Vorgehen bei gespeicherten Basismatrizen

Der Aufwand wird hier also nur noch von drei Matrix-Vektor-Produkten pro Iterationsschritt dominiert. Nachteil dieser Methode ist allerdings ein (je nach Bandbreite deutlich) erhöhter Speicherbedarf. Haben sowohl L als auch U eine Bandbreite von m, so sind zwei dicht besetzte Intervallmatrizen der Dimension $3m \times m(n-m)$ zu speichern. Die Speicherung der Basismatrizen ist also nur für kleine Bandbreiten geeignet. Da aber auch die Methode insgesamt nur für kleinere Bandbreiten sinnvoll ist, da sonst der Aufwand durch die QR-Zerlegung und die Matrix-Matrix-Produkte zu stark ansteigt und es oftmals auch zu deutlichen Überschätzungen kommt, kann die Methode der Speicherung der Basismatrizen für viele praktisch sinnvolle Anwendungen der Parallelepiped-Methode eine gute Möglichkeit zur Beschleunigung des Lösers sein.

Die Beschleunigung ist auch insofern nützlich, als dass eine Parallelisierung dieser Methode nur in Teilbereichen, z.B. bei der QR-Zerlegung und den Matrix-Matrix-Produkten, möglich ist. Übliche Parallelisierungsalgorithmen für Dreieckssysteme, z.B. basierend auf einer Fan-In-Methode, scheinen hier nicht sinnvoll übertragbar zu sein.

5.2.3. Löser basierend auf einer normweisen Abschätzung des Defekts

In diesem Abschnitt wird die Implementierung eines verifizierenden Lösers für dünn besetzte Systeme basierend auf den in Abschnitt 5.1.2 dargelegten theoretischen Überlegungen beschrieben. Es handelt sich also um eine Umsetzung von Algorithmus 5.4, wobei auch Intervallsysteme und komplexe Systeme sowie nicht symmetrisch positiv definite Systeme mittels der dort angesprochenen Modifizierungen unterstützt werden. Ebenso werden unter- und überbestimmte Systeme unterstützt. Der hier besprochene Löser wird im Folgenden auch kurz als normweiser Löser bezeichnet.

Auch der normweise Löser bietet diverse Konfigurationsoptionen, die wiederum über eine eigene `struct` gesetzt werden können. Die entsprechende Konfigurations-`struct` wird in Listing 5.19 angegeben.

```
1  struct slssnconfig {
2    bool   msg;                      //Status message output
3    int    threads;                  //Number of threads to use for OpenMP
4    int    precDecompError;          //Dot product precision for computation
5                                     //of decomposition error
6    int    maxIterPower;             //Maximum number of iterations for
7                                     //inverse power iteration
8    bool   highPrecApprxSolution;    //Compute high precision approximate
9                                     //solution
10   int    apprxSolutionStagPrec;    //Precision for computation of
11                                    //approximate solution
12   bool   scaleMatrix;              //Apply scaling of system matrix
13   bool   spd;                      //A is symmatrix positive definite
14
15
16   slssnconfig() : msg(false), threads(-1), precDecompError(1),
17                   maxIterPower(4), apprxSolutionStagPrec(3),
18                   highPrecApprxSolution(true), scaleMatrix(true),
19                   spd(false) {}
20 };
```

Listing 5.19: Konfigurations-`struct` für normweisen Löser

Die einzelnen Optionen haben folgende Bedeutung:

- `msg`: Gibt wie bei den anderen Lösern an, ob zur Laufzeit Statusmeldungen ausgegeben werden sollen. Standardmäßig ist diese Option deaktiviert.

- `threads`: Gibt die Anzahl der zu verwendenden OpenMP-Threads an. Bei Werten kleiner als 1 wird die Standardeinstellung von OpenMP, z.B. aus der entsprechenden Umgebungsvariable, verwendet. Als Standardwert wird -1 verwendet.

- `precDecompError`: Falls die schnelle Abschätzung des Fehlers der Cholesky Zerlegung der verschobenen Matrix $A - \lambda I$ nach Satz 5.5 nicht erfolgreich ist und eine direkte Berechnung durchgeführt werden muss, kann mit diesem Attribut die zu verwendende Genauigkeit eingestellt werden. Standardmäßig wird 1, also reine

Gleitkommarechnung, verwendet. Bei größerer Präzision wird die berechnete Abschätzung besser, was bei manchen Systemen erst zu einer erfolgreichen Verifikation führen kann. Allerdings steigt auch der Aufwand für die Berechnung. Details hierzu und zu einer Parallelisierung der Berechnung folgen im Verlauf dieses Abschnitts.

- `maxIterPower`: Die maximale Anzahl an Iterationsschritten für die inverse Iteration zur Bestimmung einer Näherung für den kleinsten Singulärwert von A bzw. $A^T A$. Standardmäßig werden vier Iterationsschritte durchgeführt.

- `highPrecApprxSolution`: Gibt an, ob eine hochgenaue Näherungslösung mittels des langen Akkumulators berechnet werden soll (Wert `true`) oder ob die Näherungslösung stattdessen mittels der C-XSC Operatoren und dem DotK-Algorithmus berechnet werden soll (Wert `false`). In ersterem Fall wird mehr Speicher benötigt, dafür ist die berechnete Näherungslösung genauer. Dieser Fall ist die Standardeinstellung.

- `apprxSolutionStagPrec`: Um bei Punktsystemen und engen Intervallsystemen eine gute Einschließung zu erhalten, muss die Näherungslösung möglichst genau bestimmt werden. Wie beim dünn besetzten Krawczyk-Löser wird dazu eine Staggered-artige Näherungslösung bestimmt, deren Länge mit diesem Attribut gesetzt werden kann. Standardmäßig wird der Wert 3 verwendet, d.h. die Näherungslösung wird als nicht ausgewertete Summe aus drei Vektoren berechnet und gespeichert.

- `scaleMatrix`: Gibt an, ob die Systemmatrix zu Beginn mittels van-der-Sluis Skalierung bzw. mittels Binormalisierung skaliert werden soll. Da hierdurch in der Regel die Kondition der Matrix sinkt und somit die Wahrscheinlichkeit einer erfolgreichen Verifikation erhöht wird, sollte diese Option normalerweise immer aktiviert sein. Dies ist standardmäßig auch der Fall.

- `spd`: Gibt an, ob es sich um ein symmetrisch positives System handelt oder nicht. Standardmäßig wird nicht von einem symmetrisch positiv definiten System ausgegangen. Hat `spd` den Wert `true` und der Löser berechnet erfolgreich eine Einschließung, so ist auch die positive Definitheit des Systems verifiziert.

Wie in Abschnitt 5.2.2 wurde die Hauptfunktion des Lösers als Template-Funktion implementiert, welche von Startfunktionen für die jeweiligen Datentypen ausgeprägt und aufgerufen wird. Für den Benutzer sichtbar sind wieder nur die als einfache Funktionen implementierten Startfunktionen. Die Template-Nutzung ist also wieder vor dem Benutzer verborgen, da erneut sehr viele Template-Parameter nötig sind, was einen korrekten direkten Aufruf der Template-Funktion durch den Benutzer erschweren und die Verwendung des Lösers zu stark verkomplizieren würde. Im Unterschied zum Löser aus Abschnitt 5.2.2 gibt es hier allerdings zwei verschiedene Hauptfunktionen, einmal für symmetrisch positiv definite Matrizen (`SparseLinSolveMain`) und einmal für den allgemeinen Fall (`SparseLinSolveGeneralMain`). Die Template-Parameter und die Köpfe der beiden Funktionen sind in Listing 5.20 zu sehen.

```
1  template<typename TMat, typename TRhs, typename TSolution,
2           typename TPointVec, typename TDot>
3  void SparseLinSolveMain(TMat& Ao, TRhs& bo, TSolution& xx,
4                          real& lambda, real& tau, srmatrix& S,
5                          TRhs& xappstag, TMat* rA, int& err,
6                          slssnconfig cfg);
7
8  template<typename TMat, typename TRhs, typename TSolution,
9           typename TPointVec, typename TIVec, typename TDot>
10 void SparseLinSolveGeneralMain(TMat& Ao, TRhs& bo,
11                          TSolution& xx, real& lambda, real& tau,
12                          srmatrix& T, TRhs& xappstag, TMat* rA,
13                          int& err, slssnconfig cfg);
```

Listing 5.20: Kopf der Hauptfunktionen des normweisen Lösers

Sowohl die Template- als auch die Funktionsparameter haben hier im Wesentlichen die gleiche Bedeutung. Die Template-Parameter stehen für:

- `TMat`: Typ der Systemmatrix bzw. bei Intervallsystemen der Mittelpunktsmatrix, also `srmatrix` oder `scmatrix`.
- `TRhs`: Typ der rechten Seiten des Systems bzw. bei Intervallsystemen des Mittelpunktsystems, also (da auch hier mehrere rechte Seiten unterstützt werden) `rmatrix` oder `cmatrix`.
- `TSolution`: Typ der berechneten Einschließung der Lösungsmenge, also `imatrix` oder `cimatrix`.
- `TPointVec`: Typ eines zum System passenden dicht besetzten Punktvektors, also `rvector` oder `cvector`.
- `TIVec`: Typ eines zum System passenden dicht besetzten Intervallvektors, also `ivector` oder `civector`.
- `TDot`: Zum System bzw. bei Intervallmatrizen zum Mittelpunktssystem passender langer Akkumulator-Typ, also `dotprecision` oder `cdotprecision`.

Die Funktionsparameter haben folgende Bedeutung:

- `Ao`: Die Systemmatrix bzw. bei Intervallsystemen die entsprechende Mittelpunktsmatrix. Der Radius von Intervallsystemen wird getrennt übergeben (s.u.). Ebenso werden einige der für Intervallsysteme notwendigen Anpassungen in der Startfunktion vorgenommen. Dies wird im weiteren Verlauf noch näher erläutert.
- `bo`: Die rechte Seite bzw. bei Intervallsystemen der Mittelpunkt der rechten Seite.
- `xx`: Der Lösungsvektor. Enthält nach Ablauf der Funktion die (nicht skalierte) Einschließung der aufsummierten Näherungslösung. Die weiteren Berechnungen zur Bestimmung der finalen Einschließung des Gesamtsystems werden in der Startfunktion vorgenommen (s.u.).

- `lambda`: Nach erfolgreichem Ablauf der Funktion die verifizierte untere Schranke für den kleinsten Singulärwert des Mittelpunktssystems.
- `tau`: Nach erfolgreichem Ablauf der Funktion die für das (Mittelpunkts-)System bestimmte obere Schranke für $||D||_2$ aus Satz 5.4.
- `S` bzw. `T`: Die verwendete Skalierungsmatrix für die Spaltenskalierung der Systemmatrix. Die Lösung muss in der Startfunktion noch mit dieser Matrix skaliert werden.
- `xappstag`: Die berechnete Näherungslösung in Form einer dicht besetzten Matrix, deren Spalten die zur Bestimmung der Näherungslösung zu summierenden Vektoren sind.
- `rA`: Bei Intervallsystemen der elementweise Radius der Systemmatrix. Dieser wird übergeben, um, falls nötig, Skalierungen und Permutationen vorzunehmen. Weitere Berechnungen erfolgen erst in der Startfunktion. Wird als Zeiger übergeben, wobei für Punktmatrizen der Wert `NULL` verwendet wird. Die entsprechenden Berechnungen werden dann nicht ausgeführt.
- `err`: Ein Fehlercode. Bei einem Wert ungleich 0 konnte keine verifizierte Einschließung der Lösung bestimmt werden.
- `cfg`: Eine Instanz der Konfigurations-`struct` gemäß Listing 5.19.

Vor Aufruf der eigentlichen Startfunktionen erfolgt noch die Prüfung, ob es sich um ein quadratisches System handelt. Bei über- und unterbestimmten Systemen wird eine Umformung in ein quadratisches System gemäß dem Vorgehen aus Abschnitt 4.1.1 durchgeführt. Danach wird die eigentliche Startfunktion aufgerufen, welche die Template-Hauptfunktion für die einzelnen Datentypen ausprägt. Bei Intervallsystemen werden die wesentlichen Anpassungen bei der Berechnung in diesen Startfunktionen ausgeführt. Zur Verdeutlichung zeigt Listing 5.21 als Beispiel Auszüge aus der Startfunktion für reelle Intervallsysteme.

```
1  void slssn_start(simatrix& A, imatrix& b, imatrix& xx, int& err,
2                   slssnconfig cfg) {
3    //...
4
5    srmatrix mA = mid(A);
6    rmatrix mb = mid(b);
7    srmatrix rA = 0.5*diam(A);
8
9    if(cfg.spd) {
10     SparseLinSolveMain<srmatrix,rmatrix,imatrix,rvector,dotprecision>
11                       (mA,mb,xx,lambda,tau,S,xappstag,&rA,err,cfg);
12   } else {
13     SparseLinSolveGeneralMain<srmatrix,rmatrix,imatrix,rvector,
14                               ivector,dotprecision>
15                       (mA,mb,xx,lambda,tau,S,xappstag,&rA,err,cfg);
```

```
16    }
17
18    tau = (tau + sqrt(Norm1(rA) * Norm00(rA))) / (1-3*Epsilon);
19
20    if(err!=0 || tau >= lambda) {
21      //...
22      return;
23    }
24
25    //...
26
27    xappstag = S*xappstag;
28    for(int r=1 ; r<=rhs ; r++) {
29      residual(d,b[Col(r)],A,xappstag(1,n,(r-1)*prec+1,r*prec),prec);
30      def[Col(r)] = d;
31    }
32
33    rvector nrm(rhs);
34    rvector delta(rhs);
35    for(int r=1 ; r<=rhs ; r++) {
36      nrm[r] = Norm2(absmax(def[Col(r)]));
37      delta[r] = ( ( 1.0 / (lambda - tau) ) * nrm[r] ) / (1-3*Epsilon);
38    }
39
40    //...
41
42    y = 1.0;
43    for(int r=1 ; r<=rhs ; r++)
44      xx[Col(r)] = S * xx[Col(r)] + interval(-delta[r],delta[r]) * y;
45
46    //...
47  }
```

Listing 5.21: Auszüge aus der Startfunktion des normweisen Lösers für reelle Intervallsysteme

Da es sich hier um die Startfunktion für ein reelles Intervallsystem handelt, wird zu Beginn zunächst das zugehörige Mittelpunktssystem und der Radius der Systemmatrix berechnet. Anschließend wird, abhängig davon, ob das System symmetrisch positiv definit ist oder nicht, die jeweilige Hauptfunktion des Lösers ausgeprägt und aufgerufen. Nach Ablauf der Funktion ist `tau` eine obere Schranke für $||D||_2$ aus Satz 5.4. Für Intervallsysteme muss der Radius der Systemmatrix hier zusätzlich berücksichtigt werden und die berechnete Schranke entsprechend angehoben werden. Gilt dann immer noch, dass der so berechnete Wert `tau` größer ist als die berechnete Unterschranke des kleinsten Singulärwertes `lambda` der Mittelpunktsmatrix, so ist `lambda` eine untere Schranke für den kleinsten Singulärwert aller in `A` enthaltenen Punktmatrizen.

Danach folgt die Berechnung des Residuums. Hier wird, wie schon in Abschnitt 5.2.2, die überladene Funktion `residual` verwendet. Die als Summe von Vektoren gespeicherte Näherungslösung und das Residuum werden also mit Hilfe des langen Akkumulators exakt berechnet und anschließend gerundet, so dass das Residuum bis auf eine Run-

dung genau bestimmt ist. Zu beachten ist, dass die Näherungslösung noch mit der zur Spaltenskalierung verwendeten Matrix `S` multipliziert werden muss.

Anschließend wird mit Hilfe des Residuums Δ aus Satz 5.4 bestimmt. Die hierbei nötigen Normberechnungen werden mit Hilfe der Skalarproduktalgorithmen von C-XSC so vorgenommen, dass ein sicheres Ergebnis (also eine obere Schranke der tatsächlichen Norm) berechnet wird. Damit kann dann abschließend die endgültige Einschließung bestimmt werden, indem das so berechnete `delta` mit der von der Hauptfunktion bestimmten (und noch zu skalierenden) Näherungslösung `xx` verrechnet wird.

Bei allen Berechnungen muss beachtet werden, dass Systeme mit mehreren rechten Seiten möglich sind. Die Berechnung des Residuums, von `delta` und der abschließenden Lösungseinschließung werden daher über eine for-Schleife nacheinander für alle Spalten der rechten Seite durchgeführt. In der Hauptfunktion des Lösers werden zuvor Näherungslösungen für alle rechte Seiten, ebenfalls nacheinander über eine for-Schleife, bestimmt.

Als nächstes soll nun näher auf die beiden Hauptfunktionen des Lösers eingegangen werden. Dazu wird der Ablauf der Version für symmetrisch positiv definite Matrizen schrittweise erläutert. Sofern in der allgemeinen Version Unterschiede bestehen, werden diese an der jeweilig passenden Stelle explizit beschrieben. Zu Beginn wird zunächst eine Skalierung der Matrix vorgenommen. Listing 5.22 zeigt den entsprechenden Teil des Quellcodes zur Durchführung einer van der Sluis Skalierung.

```
1   S = Id(srmatrix(n,n));
2   if(cfg.scaleMatrix) {
3     if(cfg.msg) cout << "Scaling matrix..." << endl;
4     vector<real>& S_val = S.values();
5     for(int i=1 ; i<=n ; i++){
6       if(!checkPosDef(Ao(i,i))) err = NotPosDef;
7       S_val[i-1] =
8         ::pow( 2.0, static_cast<int>(_double(-0.5*q_log2(abs(Ao(i,i))))) );
9     }
10
11    A = S*Ao*S;
12    b = S*bo;
13
14    //Check for underflow during scaling
15    if(fetestexcept(FE_UNDERFLOW) & FE_UNDERFLOW) {
16      if(cfg.msg) cout << " Underflow during scaling, no scaling used"
17                       << endl;
18      S = Id(srmatrix(n,n));
19      A = Ao;
20      b = bo;
21      feclearexcept(FE_UNDERFLOW);
22    }
23
24    if(rA != NULL) {
25      fesetround(FE_UPWARD);
26      *rA = S * (*rA) * S;
27      fesetround(FE_TONEAREST);
28    }
```

```
29
30    } else {
31      A = Ao;
32      b = bo;
33    }
```

Listing 5.22: Vorgehen bei der van der Sluis Skalierung

Allgemein wird bei der van der Sluis Skalierung mit einer Diagonalmatrix S mit den Einträgen $s_{ii} := \sqrt{a_{ii}}$ skaliert. Um Rundungsfehler zu vermeiden, werden die berechneten Werte zur nächsten Zweierpotenz gerundet. Die Skalierung kann so trotz Gleitkommarechnung exakt erfolgen, das zu lösende Problem wird also durch die Skalierung nicht verändert. Es muss allerdings überprüft werden, ob es bei der Skalierung zu einem Unterlauf kommt, da dann Informationen des nicht skalierten Systems verloren gehen. Dazu kann mittels der Funktion `fetestexcept` aus der C Standardbibliothek überprüft werden, ob das Unterlauf-Flag des Prozessors gesetzt wurde. Ist dies der Fall, so wird keine Skalierung vorgenommen und das Flag zurückgesetzt, um zukünftige Unterläufe feststellen zu können.

Bei Intervallsystemen muss zusätzlich die Radiusmatrix skaliert werden. Hierzu wird der Rundungsmodus auf Rundung nach oben geschaltet. Dadurch wird auch bei Unterlauf ein für die weiteren Berechnungen sicheres Ergebnis erzielt. Bei Punktsystemen zeigt `rA` auf `NULL`, die Berechnung wird dann nicht ausgeführt. Für nicht symmetrisch positiv definite Systeme wird analog vorgegangen, einziger Unterschied ist, wie in Abschnitt 5.1 beschrieben, die Verwendung einer Binormalisierung statt der van der Sluis Skalierung.

Nach der Skalierung wird zunächst eine Cholesky- bzw. LU-Zerlegung der Systemmatrix berechnet, mit welcher später die Näherungslösung sowie die Näherung für den kleinsten Singulärwert von A bzw. im allgemeinen Fall von A^TA berechnet werden können. Für die Zerlegungen wird die in Abschnitt 5.2.1 beschriebene SuiteSparse Schnittstelle verwendet. Die eigentliche Berechnung wird also von CHOLMOD bzw. UMFPACK vorgenommen und ist bei Verwendung einer entsprechend optimierten BLAS-Bibliothek somit auch bereits parallelisiert.

Nach der Zerlegung der Systemmatrix erfolgt die Berechnung der Näherung des kleinsten Singulärwertes von A bzw. A^TA mittels inverser Iteration. Listing 5.23 zeigt den entsprechenden Quellcode.

```
1   y = 1;
2   y = y / Norm2(y);
3   for( int k=1 ; k<=cfg.maxIterPower ; k++ ) {
4     x = y / Norm2(y);
5     lowtrisolve(L, x, h);
6     uptrisolve(U, h, y);
7     lambda = Norm2(x) / Norm2(y) ;
8   }
```

Listing 5.23: Berechnung einer Näherung des kleinsten Eigenwertes mit inverser Iteration

Danach erfolgt die Berechnung der Näherungslösung über die gleitkommamäßige Lösung der Dreieckssysteme. Hierbei wird grundsätzlich eine Staggered-artige Lösung bestimmt, deren Länge über die Konfigurations-`struct` eingestellt werden kann. Die vor-

eingestellte Länge 3 sollte in der Regel bei Punktsystemen zu einer nahezu idealen Einschließung führen.

Im allgemeinen Fall wird nun für die weiteren Berechnungen die Matrix $A^T A$ explizit benötigt. Für diese Matrix wird dann genau wie im symmetrisch positiv definiten Fall eine untere Schranke für den kleinsten Singulärwert bestimmt, deren Wurzel somit eine untere Schranke für den kleinsten Singulärwert von A darstellt. Da auf der Maschine $A^T A$ normalerweise nicht exakt berechnet werden kann, muss zusätzlich eine elementweise Abschätzung des Fehlers bestimmt werden. Bei der Verifizierung der Schranke für den kleinsten Singulärwert wird diese Fehlerabschätzung dann analog zum Radius bei der Bestimmung der Schranke für Intervallmatrizen berücksichtigt.

Dazu wird bei der Berechnung von $A^T A$ mittels Änderung des Rundungsmodus eine Intervalleinschließung bestimmt, deren Mittelpunkt für die weiteren Berechnungen verwendet wird und deren Radius als Fehlerabschätzung dient. Die Berechnung wird dabei mit Hilfe von OpenMP parallelisiert und erfolgt analog zu der bereits in Abschnitt 5.2.2 besprochenen Methode (Listing 5.7).

Der folgende Schritt im Ablauf der Hauptfunktion des Lösers ist die Bestimmung der verifizierten unteren Schranke des kleinsten Singulärwertes von A bzw. $A^T A$. Da hier kein Unterschied zwischen den beiden Versionen des Lösers besteht, wird eine gemeinsame Funktion (`boundSingMin`) für beide verwendet. Diese ist wieder als Template-Funktion implementiert, um auch den komplexen Fall direkt abzudecken. Listing 5.24 zeigt zunächst den Kopf sowie den ersten Berechnungsschritt der Funktion.

```
1  template<typename TMat>
2  bool boundSingMin(const TMat& Aorig, real& lambda, real& tau,
3                    slssnconfig& cfg) {
4    int n = RowLen(Aorig);
5    TMat A(Aorig), L(n,n), D(n,n);
6    intvector q(n);
7    real factor = 0.9;
8    bool success = false;
9    int err = 0;
10
11   while(factor > 0.0 && !success) {
12     if(cfg.msg) cout << " Trying factor=" << factor << endl;
13     real lambda_fac = factor * lambda ;
14     A = Aorig - lambda_fac*Id(A);
15     if(cfg.msg) {
16       cout << " Cholesky decomposition of A-factor*lambda*I...";
17       cout.flush();
18     }
19     chol(A,L,q,err);
20     if(err != 0) {
21       if(cfg.msg)  cout << "failed" << endl;
22       success = false;
23       factor -= 0.1;
24     } else {
25       success = true;
26       lambda = lambda_fac;
27       if(cfg.msg) cout << endl;
```

```
28      }
29    }
30
31    //...
```

Listing 5.24: Funktion zur Berechnung einer verifizierten unteren Schranke des kleinsten Singulärwertes, Teil 1: Cholesky-Zerlegung der verschobenen Matrix

Die Funktion verwendet einen Template-Parameter (`TMat`) für den Typ der betrachteten Matrix, also `srmatrix` oder `scmatrix`. Die Funktionsparameter haben folgende Bedeutung:

- `Aorig`: Die Matrix, für welche die untere Schranke des kleinsten Singulärwertes bestimmt werden soll.
- `lambda`: Zu Beginn der Funktion die zuvor berechnete Näherung für den kleinsten Singulärwert. Nach Ablauf der Funktion eine kleinere Näherung.
- `tau`: Nach Ablauf der Funktion entspricht `tau` einer oberen Schranke für $||D||_2$ aus Satz 5.4. Falls `lambda>tau`, ist also `lambda-tau` eine verifizierte untere Schranke für den kleinsten Singulärwert von `Aorig`.
- `cfg`: Eine Instanz der Konfigurations-`struct` mit der Konfiguration dieses Aufrufs des Lösers.

Als erster Schritt wird in der Funktion `boundSingMin` gemäß Lemma 5.1 die Cholesky-Zerlegung der verschobenen Matrix durchgeführt. Vor der Verschiebung wird die berechnete Näherung des kleinsten Singulärwertes mit 0.9 multipliziert. Ist die Zerlegung nicht erfolgreich, so wird als Faktor 0.8 verwendet usw., bis schließlich die Cholesky-Zerlegung erfolgreich war oder der Faktor 0 erreicht ist. In letzterem Fall war die Bestimmung der unteren Schranke des Singulärwertes erfolglos. War die Zerlegung hingegen erfolgreich, so muss die Obergrenze `tau` für $||D||_2$ bestimmt werden. Listing 5.25 zeigt zunächst die schnelle Version der Abschätzung.

```
1   //...
2
3   intvector t(n);
4   vector<int>& p = L.column_pointers();
5   for(unsigned int i=0 ; i<p.size()-1 ; i++)
6     t[i+1] = p[i+1] - p[i] - 1;
7
8   A = A(q,q);
9   rvector d(n);
10  real Eta = power(2,-1074);
11  real maxdiag = 0.0, da;
12  for(int i=1 ; i<=n ; i++) {
13    da = abs(A(i,i)); //Um im Komplexen den Realteil
14                      //zu erhalten (hermitesche Matrix!)
15    if(da > maxdiag) maxdiag = da;
16    real gamma = ((t[i]+2)*Epsilon / (1-(t[i]+2)*Epsilon))
17                 / (1-2*Epsilon);
```

```
18      d[i] = sqrt( (gamma/(1-gamma)) * da ) / (1-4*Epsilon);
19    }
20
21    real M = ( (3*(2*n+maxdiag)) / (1-3*Epsilon) );
22
23    tau = ( ((d*d)/(1-(2*n-1)*Epsilon)) + ((n*M*Eta)/(1-Epsilon)) )
24                    / (1-Epsilon);
25
26    //...
```

Listing 5.25: Funktion zur Berechnung einer verifizierten unteren Schranke des kleinsten Singulärwertes, Teil 2: Schnelle Abschätzung des Zerlegungsfehlers

Hierbei handelt es sich um eine direkte Umsetzung von Satz 5.5. Ist das so berechnete `tau` kleiner als `lambda`, so wurde erfolgreich eine untere Schranke für den Singulärwert von `Aorig` bestimmt. Ist dies nicht der Fall, so muss die aufwändigere, aber genauere direkte Berechnung in Listing 5.26 verwendet werden. Da der Aufwand für die schnelle Variante gegenüber dem Gesamtaufwand des Lösers vernachlässigbar ist, wird sie grundsätzlich zuerst ausgeführt und nur bei Misserfolg dieser Methode auf die direkte Berechnung zurückgegriffen.

```
1     //...
2
3     if(tau >= lambda) {
4       TMat U = transpherm(L);
5       TMat D_inf(n,n), D_sup(n,n);
6
7       opdotprec = cfg.precDecompError;
8       if(opdotprec == 1) {
9         LUMinusA(L,U,A,D_inf,D_sup);
10        AbsMax(D,D_inf,D_sup);
11      } else {
12        LUMinusAHighPrec(L,U,A,D);
13        opdotprec = 1;
14      }
15
16      real n1 = Norm1(D);
17      real n0 = Norm00(D);
18      tau = (sqrt(n1) * sqrt(n0)) / (1 - 3*Epsilon);
19    }
20
21    return tau < lambda;
22  }
```

Listing 5.26: Funktion zur Berechnung einer verifizierten unteren Schranke des kleinsten Singulärwertes, Teil 3: Genaue Abschätzung des Zerlegungsfehlers

Die direkte Berechnung verwendet die schon aus Abschnitt 5.2.2 bekannte Funktion `LUMinusA`, um eine elementweise Einschließung des Fehlers der Cholesky-Zerlegung zu bestimmen. Die Berechnung ist also parallelisiert. Als Ergebnis liefert die Funktion eine untere Schranke `D_inf` und eine obere Schranke `D_sup` des elementweisen Fehlers. Das

Betragsmaximum D der entsprechenden Intervallmatrix, also

$$\texttt{D} := |[\texttt{D\_inf}, \texttt{D\_sup}]|\,,$$

wird dann verwendet, um tau über

$$||\texttt{D}||_2 \leq \sqrt{||\texttt{D}||_1 ||\texttt{D}||_\infty}$$

zu berechnen.

Falls in der Konfiguration precDecompError nicht auf 1, also einfache Gleitkommarechnung, gesetzt wurde, wird die Funktion LUMinusAHighPrec statt LUMinusA ausgeführt. Hierbei handelt es sich um eine angepasste Version von LUMinusA, welche die Berechnung mit höherer Präzision ausführt. Die Parallelisierung erfolgt analog zum normalen Fall. Schon ein Wert von 2, also doppelte *double*-Präzision, sollte dabei zu einer deutlich besseren Abschätzung führen, die für einige Systeme erst eine Verifikation ermöglicht.

Nach Ablauf der Funktion boundSingMin werden die entscheidenden noch verbleibenden Berechnungen in der bereits besprochenen Startfunktion für das jeweilige System durchgeführt. Im allgemeinen Fall ist nur noch zu bedenken, dass mit der Wurzel der berechneten Unterschranke des Singulärwertes von $A^T A$ weiter gerechnet werden muss.

Zusammenfassend ist der in diesem Abschnitt besprochene Löser insbesondere für symmetrisch positiv definite Systeme mit schmalen Intervall- oder Punkteinträgen sehr gut geeignet. Durch die hochgenaue Berechnung der Näherungslösung sind für solche Systeme im Allgemeinen sehr enge Einschließungen der exakten Lösung zu erwarten. Der Aufwand bleibt für Matrizen mit guter Kondition vergleichsweise gering, da dann die schnelle Abschätzung von $||D||_2$ aus Listing 5.25 normalerweise zum Erfolg führt. Der Aufwand wird in diesem Fall von zwei Cholesky-Zerlegungen dominiert. Führt die schnelle Abschätzung nicht zum Ziel, so steigt der Aufwand deutlich an, da dann eine direkte Berechnung von $||D||_2$ gemäß Listing 5.26 nötig ist. Durch die Parallelisierung dieser Berechnung ist aber auch hier zu erwarten, dass die benötigte Zeit gegenüber einem nicht verifizierenden Löser innerhalb einer Größenordnung bleibt.

Für allgemeine Systeme ist die Klasse der lösbaren Systeme theoretisch auf gut konditionierte Systeme mit Konditionszahlen bis etwa 10^7 beschränkt, wobei durch die vorgenommene Skalierung auch bei schlechter konditionierten Systemen teilweise eine erfolgreiche Verifizierung durchgeführt werden kann. Führt die Methode zum Erfolg, so ist zu erwarten, dass sie trotz der nötigen Berechnung von $A^T A$ vergleichsweise schnell ist.

Abschnitt 5.3 zeigt nun, inwiefern diese Erwartungen in der Praxis zutreffen.

5.3. Tests und Zeitmessungen

In diesem Abschnitt werden die in diesem Kapitel vorgestellten Löser für die verifizierte Lösung dünn besetzter Gleichungssysteme detaillierten Tests hinsichtlich des Laufzeitverhaltens und der numerischen Qualität der berechneten Ergebnisse unterzogen. Dabei sollen die spezifischen Eigenschaften der verschiedenen Lösungsmethoden und die Auswirkungen der optionalen Modifizierungen (z.B. der schnellen Verifikation beim dünn

besetzten Krawczyk-Löser) jeweils anhand von konkreten Testfällen veranschaulicht werden. Auch wird detailliert auf die Auswirkungen der Parallelisierung auf die erzielten Laufzeiten eingegangen.

Als Testsysteme werden - neben einigen speziellen Beispielen, einfachen Bandmatrizen und schon in Abschnitt 4.1.4 verwendeten Systemen - vor allem Systeme aus konkreten praktischen Anwendungen verwendet. Diese Testsysteme stammen aus dem MatrixMarket [6] bzw. aus einer ähnlichen Sammlung von Testmatrizen der Universität von Florida, welche über die SuiteSparse-Webseite [7] zugänglich ist.

Um eine Einordnung der Ergebnisse vornehmen zu können, wird erneut Intlab für Vergleiche hinsichtlich Laufzeit und Ergebnisqualität herangezogen. Der Intlab-Löser `verifylss` verwendet für dünn besetzte Systeme im Wesentlichen die in Abschnitt 5.1.2 besprochene Methode der normweisen Abschätzung des Fehlers einer zuvor berechneten Näherungslösung. Für weitere Tests dient eine im Rahmen von [83] erstellte Implementierung eines Lösers für Bandmatrizen in Pascal-XSC basierend auf der Vorwärtssubstitution mit Parallelepipeden sowie eine direkte C-XSC Umsetzung dieses Lösers durch Carlos Hölbig [61]. Beide Löser verwenden durchgehend den langen Akkumulator, es ist also zu erwarten, dass sie deutlich langsamer sind als die Implementierung aus 5.2.2. Allerdings eignen sie sich gut als Referenz für die numerische Qualität der Ergebnisse.

Als Testsystem dient auch in diesem Abschnitt wieder ein Rechner mit zwei Intel Xeon E5520 Prozessoren mit jeweils 4 mit 2.26 GHz getakteten Kernen und 24 GB mit 1066MHz getaktetem DDR3 Hauptspeicher. Als Betriebssystem kommt OpenSUSE 11.4 in der 64-Bit Version zum Einsatz. Als C++ Compiler dient der Intel Compiler in Version 12.1, die Intel MKL in Version 10.3 wird als BLAS und LAPACK Bibliothek verwendet. Für die dünn besetzte LU- bzw. Cholesky-Zerlegung wird SuiteSparse in Version 4.0.2 genutzt. Für die Vergleichstests wird Intlab in Version 6 laufend auf Matlab 7.12.0.635 (R2011a) herangezogen. Zu der von Matlab genutzten BLAS und LAPACK Bibliothek gelten die gleichen Hinweise wie zu Beginn von Abschnitt 4.1.4.

In diesem Abschnitt werden, sofern aus Platzgründen nötig, folgende Kurzbezeichnungen für die verschiedenen Löser verwendet:

- *SparseNorm*: Löser mit verifizierter normweiser Abschätzung des Defekts gemäß Abschnitt 5.2.3.
- *KrawSubst*: Löser basierend auf dem Krawczyk-Operator mit Lösung der Dreieckssysteme mittels einfacher Vorwärts/Rückwärts-Substitution.
- *KrawBand*: Löser für Bandsysteme basierend auf dem Krawczyk-Operator mit Lösung der Dreieckssysteme mittels Vorwärts/Rückwärts-Substitution auf Basis von Parallelepipeden.
- *KrawBandStore*: Löser für Bandsysteme basierend auf dem Krawczyk-Operator mit Lösung der Dreieckssysteme mittels Vorwärts/Rückwärts-Substitution auf Basis von Parallelepipeden mit Zwischenspeicherung der Basismatrizen.
- *KrawInv*: Löser für Bandsysteme basierend auf dem Krawczyk-Operator mit Lösung der Dreieckssysteme mittels impliziter Berechnung der Inversen.

- *BandCXSC*: Alte C-XSC Version des Lösers für Bandsysteme basierend auf dem Krawczyk-Operator mit Lösung der Dreieckssysteme mittels Vorwärts/Rückwärts-Substitution auf Basis von Parallelepipeden.
- *BandPXSC*: Alte Pascal-XSC Version des Lösers für Bandsysteme basierend auf dem Krawczyk-Operator mit Lösung der Dreieckssysteme mittels Vorwärts/Rückwärts-Substitution auf Basis von Parallelepipeden.
- *DenseLSS*: Dicht besetzter Löser basierend auf dem Krawczyk-Operator aus Abschnitt 4.1.
- *Intlab*: Der Löser `verifylss` aus Intlab.

5.3.1. Generierte Bandmatrizen

Zunächst sollen einige Tests mit generierten Bandmatrizen mit geringer Bandbreite durchgeführt werden. Dies ermöglicht einen direkten Vergleich der verschiedenen Lösungsansätze, da diese Art von Matrizen für alle Löser geeignet ist. Insbesondere ist die Variante des Krawczyk-Operators, bei welcher die auftretenden Dreieckssysteme mit Hilfe von Parallelepipeden gelöst werden, anwendbar.

Alle Tests wurden mit $P = 1, 2, 4, 8$ Threads durchgeführt und mehrfach wiederholt, um eventuelle Schwankungen, verursacht z.B. durch Systemprozesse, auszugleichen. Allgemein sollte vorab betont werden, dass die Parallelisierung für die hier verwendeten Systeme mit sehr niedriger Bandbreite oft nur geringe Auswirkungen hat. Spätere Tests in Abschnitt 5.3.2 mit allgemeinen Matrizen zeigen die Vorteile der Parallelisierung wesentlich deutlicher.

Um einen ersten Überblick, vor allem über die Laufzeitunterschiede der verschiedenen Löser, zu erhalten, wird zunächst ein Test mit einer sehr gut konditionierten, symmetrisch positiv definiten Systemmatrix durchgeführt. Betrachtet wird das System mit der $n \times n$ Matrix

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

und der rechten Seite $b \in \mathbb{R}^n$ mit $b_i = 1$, $i = 1, \ldots, n$. Dieses System hat für alle hier betrachteten Dimensionen nach Schätzung von Matlab eine Kondition von etwa 13, ist also sehr gut konditioniert. Es wurden Vergleichstests für die Dimensionen $n = 1000, 5000, 10000, 100000, 1000000$ (der Löser für dicht besetzte Systeme wurde nur für $n = 1000$ und $n = 5000$, der Löser *KrawInv* nur bis einschließlich $n = 10000$ getestet)

durchgeführt. Bei allen Lösern werden jeweils die Standardeinstellungen verwendet. Die gemessenen Zeiten werden in Tabelle 5.1 gezeigt, der relative Fehler der berechneten Einschließungen in Tabelle 5.2. Aufgrund der geringen Auswirkungen der Parallelisierung werden nur die Zeiten für 1 und 2 Threads angegeben. Der relative Fehler ist jeweils unabhängig von der Anzahl der Threads und wird daher nur einmal aufgeführt.

Threads	**Löser**	$n = 1000$	$n = 5000$	$n = 10^4$	$n = 10^5$	$n = 10^6$
$P = 1$	SparseNorm	0.012	0.068	0.144	1.62	17.9
	Intlab	0.021	0.081	0.157	1.67	18.8
	KrawSubst	0.019	0.094	0.190	1.95	19.9
	KrawInv	0.33	8.82	32.12	-	-
	KrawBand	0.11	0.57	1.18	11.96	121.0
	KrawBandStore	0.063	0.31	0.64	6.36	64.6
	BandCXSC	0.11	0.56	1.12	11.07	110.6
	BandPXSC	0.45	3.31	4.33	43.31	425.1
	DenseLSS	0.85	370.4	-	-	-
$P = 2$	SparseNorm	0.010	0.057	0.126	1.41	16.0
	Intlab	0.022	0.082	0.157	1.66	18.8
	KrawSubst	0.013	0.069	0.137	1.41	15.1
	KrawInv	0.33	7.71	25.66	-	-
	KrawBand	0.11	0.56	1.14	11.68	118.4
	KrawBandStore	0.058	0.29	0.59	6.06	61.7
	BandCXSC	0.11	0.56	1.12	11.07	110.6
	BandPXSC	0.45	3.31	4.33	43.31	425.1
	DenseLSS	0.45	195.9	-	-	-

Tabelle 5.1.: Zeit in Sekunden zur verifizierten Lösung eines einfaches Bandsystems

Es zeigt sich zunächst, dass der Löser *SparseNorm* im Allgemeinen der schnellste Löser ist. Er ist auch etwas schneller als der Intlab-Löser, welcher den gleichen Algorithmus verwendet, obwohl hier vom C-XSC Löser zusätzlich ein hochgenaues Residuum berechnet wird. Der Aufwand steigt für das betrachtete Bandsystem jeweils linear mit der Dimension.

Die Verwendung von 2 Threads bringt leichte Laufzeitgewinne für den *SparseNorm*-Löser, welche durch eine leichte Beschleunigung der Cholesky-Zerlegung und der Residuumsberechnung zustande kommen. Aufgrund der Besetztheit und Struktur der Testmatrix fällt der Gewinn allerdings gering aus. Der Intlab-Löser profitiert offenbar nur

während der Cholesky-Zerlegung von weiteren Threads, entsprechend fällt der Laufzeitgewinn hier noch wesentlich geringer aus.

Der relative Fehler liegt für den C-XSC Löser *SparseNorm* durch die hochgenaue Residuumsberechnung durchgehend im Bereich 10^{-17}, die berechnete Einschließung ist also nahezu ideal. Auch Intlab erreicht eine enge Einschließung, trotz der sehr guten Kondition des Systems gehen aber bereits 2-3 Stellen Genauigkeit verloren. Dieser Effekt wird sich in späteren Tests noch deutlich verstärken.

Der Löser *KrawSubst* zeigt hier ähnliche Laufzeiten wie Intlab und der *SparseNorm* Löser und ist nur wenig langsamer. Weiterhin profitiert er etwas stärker von weiteren Threads, vor allem für größere Dimensionen, da die Berechnung $\boldsymbol{x}_{i+1} = \boldsymbol{z} + [LU - A]\boldsymbol{x}_i$ in den einzelnen Verifikationsschritten recht gut parallelisiert werden kann. für $n = 10^6$ und zwei Threads ist dieser Löser dadurch sogar insgesamt der schnellste. Die berechnete Einschließung ist dabei auch für diesen Löser durchgehend nahezu ideal.

Als nächstes werden die vier verschiedenen Löser, welche Parallelepipede für die Lösung verwenden, betrachtet. Der alte Pascal-XSC Löser ist dabei deutlich am langsamsten. Der Löser *KrawBand* ist in etwa gleich schnell wie die alte C-XSC Implementierung *BandCXSC*. Auf diesen Umstand wird in Kürze genauer eingegangen. Bei Zwischenspeicherung der Basismatrizen (Löser *KrawBandStore*) ergibt sich klar die beste Laufzeit der vier Versionen. Allerdings sind alle vier Löser durch den hohen Aufwand der Verifikationsschritte bei Verwendung von Parallelepipeden deutlich langsamer als die zuvor betrachteten Löser. Die beiden neuen Löser *KrawBand* und *KrawBandStore* gewinnen nur wenig durch eine Parallelisierung, da im Wesentlichen nur die Berechnung des Residuums und die Berechnung $\boldsymbol{x}_{i+1} = \boldsymbol{z} + [LU - A]\boldsymbol{x}_i$ in den einzelnen Verifikationsschritten parallelisiert wird. Die alten XSC-Löser verwenden keinerlei Parallelisierung.

Der relative Fehler der beiden neuen Löser liegt im Bereich 10^{-18}, ist also wiederum nahezu ideal. Die Ergebnisse bei Zwischenspeicherung der Basismatrizen sind identisch zur normalen Version. Einziger Nachteil der Zwischenspeicherung ist somit der erhöhte Speicherbedarf. Die alten XSC-Löser zeigen durch die durchgehende Verwendung des langen Akkumulators ebenfalls einen sehr geringen relativen Fehler in der Größenordnung 10^{-16}. Durch die hochgenaue Residuumsberechnung sind die neuen Löser somit trotz nicht durchgehender Verwendung des Akkumulators sogar genauer.

Zu klären ist nun noch, warum der Löser *KrawBand* trotz Optimierungen in etwa die gleichen Laufzeiten wie die alte C-XSC Version aufweist. Ursache hierfür ist, dass es bei dem betrachteten Testsystem durch die hochgenaue Berechnung der Näherungslösung und die niedrige Kondition zu massivem Unterlauf kommt. Berechnungen im Unterlaufbereich sind deutlich langsamer als normale Gleitkommaberechnungen, was hier zu der starken Verlangsamung führt. Wird der gleiche Test mit einer einfachen Berechnung der Näherungslösung über die C-XSC Operatoren mit Präzision $K = 2$ durchgeführt, so wird das Auftreten von Unterlauf stark reduziert und es ergeben sich die Zeiten in Tabelle 5.3 (hier wird durchgehend nur 1 Thread verwendet).

Beide neue Versionen sind nun also deutlich schneller als der alte C-XSC Löser. Der relative Fehler liegt dann für alle Dimensionen in etwa bei $7.7 \cdot 10^{-16}$. Er hat sich also leicht verschlechtert, liegt aber immer noch im gleichen Bereich wie der relative Fehler des alten C-XSC Lösers. Zu betonen ist, dass der hier beobachtete Zeitvorteil nahezu aus-

Löser	$n = 1000$	$n = 5000$	$n = 10^4$	$n = 10^5$	$n = 10^6$
SparseNorm	$2.09 \cdot 10^{-17}$	$2.08 \cdot 10^{-17}$	$2.08 \cdot 10^{-17}$	$2.08 \cdot 10^{-17}$	$2.08 \cdot 10^{-17}$
Intlab	$2.26 \cdot 10^{-14}$	$4.88 \cdot 10^{-14}$	$6.88 \cdot 10^{-14}$	$2.17 \cdot 10^{-13}$	$6.86 \cdot 10^{-13}$
KrawSubst	$6.95 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$
KrawInv	$6.95 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	-	-
KrawBand	$6.95 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$
KrawBandStore	$6.95 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$	$6.94 \cdot 10^{-18}$
BandCXSC	$1.38 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$
BandPXSC	$1.38 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$	$1.39 \cdot 10^{-16}$
DenseLSS	$1.52 \cdot 10^{-16}$	$1.52 \cdot 10^{-16}$	-	-	-

Tabelle 5.2.: Relativer Fehler bei verifizierter Lösung eines einfachen Bandsystems

Löser	$n = 1000$	$n = 5000$	$n = 10^4$	$n = 10^5$	$n = 10^6$
KrawBand	0.076	0.37	0.74	7.57	76.4
KrawBandStore	0.053	0.27	0.53	5.72	57.1
BandCXSC	0.11	0.56	1.12	11.07	110.6

Tabelle 5.3.: Zeitvergleich in Sekunden bei einfacher Residuumsberechnung

schließlich durch das massive Auftreten von Unterlauf bei hochgenauer Bestimmung der Näherungslösung ausgelöst wird, nicht aber durch den Mehraufwand der Bestimmung der Näherungslösung selbst. Der hier beobachtete Effekt ist auch kein allgemeines Phänomen bei Verwendung einer hochgenauen Näherungslösung, sondern tritt insbesondere für sehr gut konditionierte Systeme auf. In der Folge werden die alten XSC-Löser nicht mehr für Vergleiche herangezogen, da die bereits getroffenen Aussagen im Allgemeinen direkt auf andere Systeme übertragbar sind.

Als letztes wird der Löser *KrawInv* betrachtet. Wie zu erwarten, ist dieser von den dünn besetzten Lösern deutlich der langsamste. Allerdings erweist er sich trotzdem als deutlich schneller als der dicht besetzte Löser. Er profitiert ebenfalls leicht von zusätzlichen Threads. Die Ergebnisqualität bewegt sich in einem ähnlichen Rahmen wie bei den anderen Lösern.

Abbildung 5.3 zeigt einen Gesamtüberblick über den Speed-Up der neuen dünn besetzten Löser für das betrachtete Testsystem bei Dimension $n = 10000$. Zu beachten ist dabei, dass die vertikale Achse nur bis 1.9 reicht. Der Speed-Up ist aufgrund der Struktur der Matrix recht gering und nimmt für mehr als zwei Threads oftmals sogar ab. Allgemein zeigt sich hier, dass der Löser *KrawSubst* stärker von der Parallelisierung profitiert, während der Effekt bei den Bandlösern sehr begrenzt ist, da hier nur wenige

Teile des Lösers parallelisiert sind.

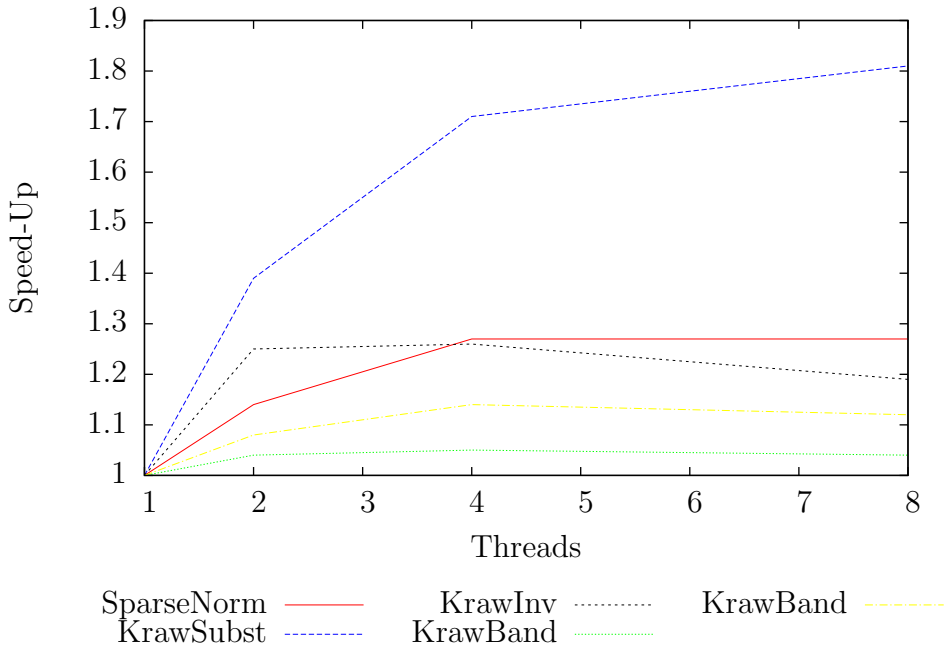


Abbildung 5.3.: Speed-Up für einfache Bandmatrix, Dimension $n = 10000$

Als nächstes wird ein deutlich schlechter konditioniertes Beispielsystem aus der Sammlung von Gregory und Karney [48] betrachtet. Als Systemmatrix wird

$$A = \begin{pmatrix} 5 & -4 & 1 & & & & & & & \\ -4 & 6 & -4 & 1 & & & & & & \\ 1 & -4 & 6 & -4 & 1 & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & 1 & -4 & 6 & -4 & 1 \\ & & & & & & & 1 & -4 & 6 & -4 \\ & & & & & & & & 1 & -4 & 5 \end{pmatrix}$$

verwendet, als rechte Seite der erste Einheitsvektor. Für die getesteten Dimensionen $n = 1000, 5000, 10000, 15000$ ergeben sich in etwa die Konditionszahlen $2.1 \cdot 10^{11}$, $1.3 \cdot 10^{14}$, $1.9 \cdot 10^{15}$ und $6.7 \cdot 10^{15}$. Für größere Dimensionen ist das System für die hier verwendeten Lösungsmethoden zu schlecht konditioniert. Verglichen werden die Löser *SparseNorm*, *Intlab*, sowie die beiden Bandlöser *KrawBand* und *KrawBandStore*. Es wird jeweils nur ein Thread verwendet. Tabelle 5.4 zeigt die gemessenen Laufzeiten, Tabelle 5.5 den relativen Fehler für dieses Testsystem.

Sowohl Intlab als auch der *SparseNorm*-Löser können nur bis $n = 10000$ eine verifizierte Einschließung berechnen. Für $n = 15000$ führt der von beiden verwendete Algo-

Löser	$n = 1000$	$n = 5000$	$n = 10000$	$n = 15000$
SparseNorm	0.007	0.042	0.089	-
Intlab	0.009	0.038	0.049	-
KrawBand	0.11	0.55	1.09	2.25
KrawBandStore	0.07	0.31	0.65	1.12

Tabelle 5.4.: Zeit in Sekunden zur verifizierten Lösung von Rumps schlechter konditioniertem Beispiel-Bandsystem

Löser	$n = 1000$	$n = 5000$	$n = 10000$	$n = 15000$
SparseNorm	$2.20 \cdot 10^{-16}$	$5.43 \cdot 10^{-9}$	$2.35 \cdot 10^{-3}$	-
Intlab	$4.16 \cdot 10^{-4}$	$2.47 \cdot 10^{4}$	$5.34 \cdot 10^{7}$	-
KrawBand	$7.23 \cdot 10^{-17}$	$1.20 \cdot 10^{-16}$	$2.00 \cdot 10^{-16}$	$3.01 \cdot 10^{-16}$
KrawBandStore	$7.23 \cdot 10^{-17}$	$1.20 \cdot 10^{-16}$	$2.00 \cdot 10^{-16}$	$3.01 \cdot 10^{-16}$

Tabelle 5.5.: Relativer Fehler bei der verifizierten Lösung von Rumps schlechter konditioniertem Beispiel-Bandsystem

rithmus nicht mehr zum Erfolg. Intlab ist in diesem Test etwas schneller, vermutlich da hier die standardmäßig von *SparseNorm* genutzte hochgenaue Näherungslösung in dreifacher Länge stärker ins Gewicht fällt. Allerdings kann der C-XSC-Löser dadurch auch einen deutlich besseren relativen Fehler der Lösungseinschließung vorweisen. Durch die hohe Kondition für $n = 10000$ ist allerdings auch die berechnete Cholesky-Zerlegung relativ ungenau, wodurch der relative Fehler trotz der hochgenauen Berechnung von Näherungslösung und Residuum in der Größenordnung 10^{-3} liegt.

Hier zeigen sich die Vorteile der Bandlöser. Zwar sind sie in diesem Test in etwa um eine Größenordnung langsamer als die anderen Löser, allerdings bestimmen sie, auch durch die hochgenaue Berechnung der Matrixzerlegung, eine deutlich bessere Einschließung. Dies gilt auch für die Dimension $n = 15000$, bei der die beiden anderen Löser keine verifizierte Einschließung berechnen konnten. Der relative Fehler bewegt sich dabei stets im Bereich 10^{-16}, ist also nahezu ideal.

Ein weiteres Beispiel für die Vorteile der Verwendung des Parallelepiped-Algorithmus ist das System aus Beispiel 5.1 (untere Dreiecksmatrix mit drei Bändern, deren Elemente alle 1 sind, sowie rechter Seite mit allen Elementen gleich $[-1, 1]$). Tabelle 5.6 zeigt die Ergebnisse des entsprechenden Tests.

Sowohl der Löser *SparseNorm* als auch der Intlab-Löser sind wieder deutlich schneller, wobei hier *SparseNorm* einen kleinen Vorteil aufweist. Die relative Breite der Einschließung ist wiederum bei *SparseNorm* deutlich besser als bei Intlab. Ursache hierfür ist, dass *SparseNorm* bereits die ursprüngliche Matrix skaliert, während Intlab die Methode für symmetrisch positiv definite Systeme direkt auf das System $A^T Ax = A^T b$ anwendet.

Löser	Zeit	Rel. Breite	$\boldsymbol{x}_1$	$\boldsymbol{x}_n$
SparseNorm	13.5	$3.87 \cdot 10^8$	$[-3.9 \cdot 10^8, 3.9 \cdot 10^8]$	$[-3.9 \cdot 10^8, 3.9 \cdot 10^8]$
Intlab	14.4	$5.07 \cdot 10^{14}$	$[-5.1 \cdot 10^{14}, 5.1 \cdot 10^{14}]$	$[-5.1 \cdot 10^{14}, 5.1 \cdot 10^{14}]$
KrawBand	42.8	$3.88 \cdot 10^{10}$	$[-1, 1]$	$[-1.1 \cdot 10^{11}, 1.1 \cdot 10^{11}]$
KrawBandStore	32.6	$3.88 \cdot 10^{10}$	$[-1, 1]$	$[-1.1 \cdot 10^{11}, 1.1 \cdot 10^{11}]$

Tabelle 5.6.: Testergebnisse für Neumaiers Beispiel-Bandsystem, Dimension $n = 10^6$

Da beide Löser eine normweise Abschätzung des Fehlers berechnen, sind hier alle Elemente des Lösungsvektors sehr breit. Die Bandlöser berechnen zwar insgesamt eine etwas breitere Einschließung als *SparseNorm*, dieses Ergebnis täuscht hier allerdings. Die Bandlöser liefern eine elementweise gute Abschätzung, wodurch z.B. das erste Element des Lösungsvektors sogar exakt bestimmt wird, während der Radius der Einschließung erst bei den letzteren Elementen zunimmt, wodurch die relative Breite insgesamt stärker erscheint. Es zeigt sich aber, dass für sehr schlecht konditionierte Bandsysteme mit geringen Bandbreiten oder bei recht breiten intervallwertigen rechten Seiten die Verwendung der Parallelepiped-Methode trotz der höheren Laufzeit eine interessante Alternative sein kann.

Zum Abschluss der Tests mit generierten Bandmatrizen kleiner Bandbreite wird nun noch ein komplexes Beispielsystem betrachtet. Als Testsystem wird dabei $A = BB^*$ verwendet (B^* ist die Hermitesche von B) mit

$$B = \begin{pmatrix} 4+6i & & & & \\ 2+i & 4+6i & & & \\ 1+i & 2+i & 4+6i & & \\ & & & \ddots & \\ & & & 1+i & 2+i & 4+6i \end{pmatrix}$$

und als rechter Seite dem Vektor mit jedem Element gleich 1. Alle weiteren Rahmenbedingungen entsprechen dem ersten Test in diesem Abschnitt. Tabelle 5.7 zeigt die dabei erreichten Laufzeiten, Tabelle 5.8 den relativen Fehler.

Der Intlab Löser bricht bei diesem Test für alle Dimensionen mit einer Fehlermeldung ab. Ursache hierfür ist, dass Intlab die Matlab-Methode `isequal` verwendet, um beim Test auf hermitesch positive Definitheit der Matrix A die Gleichheit von A und A^* zu überprüfen. Dieser Test wird auf die um die berechnete Näherung des kleinsten Singulärwerts verschobene Systemmatrix angewendet und liefert hier als Ergebnis fälschlicherweise, dass die Matrix nicht hermitesch sei. Anscheinend handelt es sich um einen Bug dieser Funktion oder es treten unbekannte Seiteneffekte auf. Auch bei Entfernung der entsprechenden Abfrage aus dem Intlab-Löser liefert dieser als Ergebnis stets `NaN`.

Ansonsten zeigt sich bei den Tests ein ähnliches Bild wie im Reellen, wobei insgesamt

Threads	Löser	$n = 1000$	$n = 5000$	$n = 10^4$	$n = 10^5$	$n = 10^6$
$P = 1$	SparseNorm	0.089	0.45	0.92	9.25	97.9
	Intlab	-	-	-	-	-
	KrawSubst	0.17	0.84	1.69	17.1	175.6
	KrawInv	0.17	0.88	1.79	-	-
	KrawBand	0.26	1.82	3.69	37.2	365.7
	KrawBandStore	0.19	0.97	1.99	20.4	199.0
	DenseLSS	3.10	331.8	-	-	-
$P = 2$	SparseNorm	0.080	0.41	0.82	8.21	84.2
	Intlab	-	-	-	-	-
	KrawSubst	0.15	0.72	1.47	14.4	143.5
	KrawInv	0.18	0.90	1.78	-	-
	KrawBand	0.25	1.69	3.39	33.9	357.7
	KrawBandStore	0.17	0.86	1.76	17.8	175.4
	DenseLSS	1.60	169.4	-	-	-

Tabelle 5.7.: Zeit in Sekunden zur verifizierten Lösung eines einfachen komplexen Bandsystems

die Zeiten durch den höheren Aufwand recht stark ansteigen. Der Löser *SparseNorm* ist klar der schnellste, gefolgt von *KrawSubst*. Es folgen die beiden Bandlöser und der Löser *KrawInv*. Die Ergebnisqualität (der relative Fehler wurde hier getrennt für Real- und Imaginärteil berechnet, die Tabelle zeigt jeweils den größeren der beiden Werte) ist auch hier für alle Löser sehr gut, es werden jeweils nahezu ideale Einschließungen berechnet. Auch für die Parallelisierung gelten ähnliche Aussagen wie im Reellen, weshalb hier auf eine nähere Betrachtung des Speed-Ups verzichtet wird.

5.3.2. Testmatrizen aus praktischen Anwendungen

Im Folgenden werden Tests mit Matrizen aus praktischen Anwendungen durchgeführt. Diese stammen entweder aus dem Matrix-Market [6] oder aus der Sammlung der Universität von Florida [7]. Es werden jeweils nur die wichtigsten Eigenschaften der Testmatrizen angesprochen, für mehr Details wird in Anhang B eine detaillierte Aufzählung aller in dieser Arbeit verwendeten dünn besetzten Testmatrizen aus diesen beiden Quellen angegeben.

Löser	$n = 1000$	$n = 5000$	$n = 10^4$	$n = 10^5$	$n = 10^6$
SparseNorm	$2.60 \cdot 10^{-18}$	$2.60 \cdot 10^{-18}$	$2.60 \cdot 10^{-18}$	$2.60 \cdot 10^{-18}$	$2.60 \cdot 10^{-18}$
Intlab	-	-	-	-	-
KrawSubst	$8.66 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$
KrawInv	$8.66 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	-	-
KrawBand	$8.66 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$
KrawBandStore	$8.66 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$	$8.67 \cdot 10^{-19}$
DenseLSS	$5.26 \cdot 10^{-18}$	$5.21 \cdot 10^{-18}$	-	-	-

Tabelle 5.8.: Relativer Fehler bei verifizierter Lösung eines einfachen komplexen Bandsystems

Symmetrisch positiv definite Systeme

Zunächst werden Tests mit reellen, symmetrisch positiv definiten Systemen durchgeführt. Die verwendeten Testmatrizen mit den jeweils wichtigsten Eigenschaften werden in Tabelle 5.9 aufgezählt.

Name	Dimension n	nnz	Bandbreite	Kondition
bcsstk10	1086	11578	37	$1.3 \cdot 10^6$
bcsstk18	11948	80519	1243	65
bcsstm05	153	153	0	$1.3 \cdot 10^6$
ecology2	999999	4995991	1000	$6.7 \cdot 10^7$
G3_circuit	1585478	7660826	947128	$2.2 \cdot 10^7$
s3dkq4m2	90449	2455670	614	$3.5 \cdot 10^{11}$
s3dkt3m2	90449	1921955	614	$6.3 \cdot 10^{11}$
thermomech_TC	102158	711558	102138	123

Tabelle 5.9.: Symmetrisch positiv definite Testmatrizen

Mit jeder dieser Matrizen wird ein entsprechendes Testsystem mit rechter Seite $b_i = 1$, $i = 1, \ldots, n$ gelöst. Dabei werden die Bandlöser *KrawBand*, *KrawBandStore*, *BandCXSC* und *BandPXSC* hier nicht mehr verwendet, da sie für die betrachteten Systeme ungeeignet sind. Für die Matrix *bcsstk10* mit Bandbreite 37 ist der Aufwand zwar gering genug für einen Test, allerdings ist auch für dieses System keine verifizierte Einschließung mehr möglich, da die Überschätzung durch die Parallelepipede mit steigender Bandbreite im Allgemeinen deutlich zunimmt. Für die Diagonalmatrix *bcsstm0* lässt sich natürlich trotzdem eine Lösung berechnen, die Ergebnisse entsprechen dann aber denen

von *KrawSubst*. Weiterhin werden die Löser *DenseLSS* und *KrawInv* nur für Systeme mit hinreichend kleiner Dimension getestet. Tabelle 5.10 zeigt die Zeiten für einen und acht Threads, Tabelle 5.11 den relativen Fehler (dieser ist wiederum unabhängig von der Anzahl der Threads und wird nur einmal angegeben).

Threads	**Matrix**	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
$P = 1$	bcsstk10	0.036	0.032	-	0.57	1.13
	bcsstk18	0.574	0.668	4.29	282.9	1115.8
	bcsstm05	0.001	0.005	0.001	0.005	0.009
	ecology2	39.2	174.8	462.1	-	-
	G3_circuit	93.3	208.5	1291.5	-	-
	s3dkq4m2	114.3	180.2	-	-	-
	s3dkt3m2	106.5	105.3	-	-	-
	thermomech_TC	3.75	9.30	22.3	-	-
$P = 8$	bcsstk10	0.023	0.032	-	0.45	0.21
	bcsstk18	0.524	0.676	1.39	80.7	194.7
	bcsstm05	0.001	0.005	0.001	0.012	0.003
	ecology2	35.8	172.1	139.8	-	-
	G3_circuit	80.7	164.7	342.8	-	-
	s3dkq4m2	28.6	176.6	-	-	-
	s3dkt3m2	25.8	103.8	-	-	-
	thermomech_TC	3.42	9.31	8.61	-	-

Tabelle 5.10.: Zeit in Sekunden zur verifizierten Lösung symmetrisch positiv definiter Punktsysteme

Zunächst fällt auf, dass der *SparseNorm*-Löser erneut nahezu durchgehend schneller ist als der Intlab-Löser. Ausnahmen sind die Matrizen *bcsstk10* und *s3dkt3m2*, für die beide in etwa die gleiche Zeit erreichen.

Die große Differenz bei der Matrix *ecology2* erklärt sich darüber, dass C-XSC hier mit der schnellen Verifikation nach Satz 5.5 erfolgreich ist, Intlab hingegen eine direkte Berechnung des Fehlers der verschobenen Cholesky-Zerlegung vornehmen muss. Ursache hierfür ist, dass C-XSC als untere Grenze für den Singulärwert zunächst $0.9\tilde{\lambda}$ verwendet ($\tilde{\lambda}$ bezeichnet die zuvor berechnete Näherung des kleinsten Singulärwertes), Intlab hingegen $0.8\tilde{\lambda}$. In diesem Fall wird hierdurch gerade die Grenze für die schnelle Verifikation überschritten, diese Wahl hat für dieses Testsystem also enormen Einfluss auf die Laufzeit.

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
bcsstk10	$6.4 \cdot 10^{-19}$	$7.6 \cdot 10^{-8}$	-	$2.1 \cdot 10^{-19}$	$5.7 \cdot 10^{-19}$
bcsstk18	$1.4 \cdot 10^{-17}$	$2.6 \cdot 10^{-7}$	$1.4 \cdot 10^{-18}$	$1.4 \cdot 10^{-18}$	$1.7 \cdot 10^{-17}$
bcsstm05	$2.2 \cdot 10^{-16}$	$5.4 \cdot 10^{-15}$	$7.5 \cdot 10^{-17}$	$7.5 \cdot 10^{-17}$	$1.8 \cdot 10^{-16}$
ecology2	$2.1 \cdot 10^{-16}$	$1.8 \cdot 10^{-6}$	$7.1 \cdot 10^{-17}$	-	-
G3_circuit	$2.4 \cdot 10^{-16}$	$3.4 \cdot 10^{3}$	$8.1 \cdot 10^{-17}$	-	-
s3dkq4m2	$7.7 \cdot 10^{-11}$	$4.0 \cdot 10^{6}$	-	-	-
s3dkt3m2	$1.5 \cdot 10^{-12}$	$5.2 \cdot 10^{6}$	-	-	-
thermomech_TC	$2.4 \cdot 10^{-16}$	$1.6 \cdot 10^{-12}$	$8.1 \cdot 10^{-17}$	-	-

Tabelle 5.11.: Relativer Fehler bei der verifizierten Lösung symmetrisch positiv definiter Punktsysteme

Auch bei einigen der anderen großen Matrizen (*G3_circuit*,*thermomech_TC*, *s3dkq4m2*) zeigen sich starke Unterschiede. Der Grund hierfür ist, dass die Bestimmung der Näherungslösung und der Näherung an den kleinsten Singulärwert bei Intlab deutlich mehr Zeit benötigt als bei C-XSC. Die genaue Ursache hierfür ist unklar, da dieses Verhalten nicht bei allen Matrizen auftritt, möglicherweise ist auch ein Bug in Intlab oder Matlab der Grund. Im Allgemeinen wäre zu erwarten, dass Intlab und C-XSC, wie es z.B. auch bei der Matrix *s3dkt3m2* der Fall ist, in etwa die gleichen Laufzeiten aufweisen. C-XSC sollte theoretisch geringfügige Laufzeitvorteile durch den Interpretations-Overhead von Matlab erhalten, welche in der Praxis durch die aufwändigere hochgenaue Bestimmung von Näherungslösung und Residuum in etwa ausgeglichen werden sollten.

In jedem Fall ist ersichtlich, dass *SparseNorm* wesentlich stärker von Multithreading profitiert als Intlab. Insbesondere bei Systemen, bei denen die schnelle Verifizierung nicht zum Erfolg führt (hier *s3dkq4m2* und *s3dktm2*), ist der Effekt der Parallelisierung sehr deutlich, da dann die aufwändige Berechnung des Fehlers der verschobenen Cholesky-Zerlegung recht effektiv parallelisiert werden kann. Intlab profitiert im Wesentlichen nur bei der Cholesky-Zerlegung von mehreren Threads, wodurch der Laufzeitgewinn im Allgemeinen recht gering ausfällt. Bei Verwendung von acht Threads ist daher der C-XSC Löser *SparseNorm* für alle getesteten Systeme (in der Regel deutlich) schneller.

Der relative Fehler ist durch die hochgenaue Berechnung von Näherungslösung und Residuum dabei durchgehend deutlich geringer als bei Intlab, welches gerade für die großen Testmatrizen *G3_circuit*, *s3dkq4m2* und *s3dkt3m2* sehr breite Einschließungen in der Größenordnung 10^3 bzw. 10^6 berechnet. Der relative Fehler von *SparseNorm* hingegen ist zumeist nahezu ideal, lediglich für die schlechter konditionierten Matrizen *s3dkq4m2* und *s3dkt3m2* fällt er etwas höher aus, kann aber durch eine Erhöhung der Genauigkeit mit vergleichsweise geringem Aufwand verbessert werden (siehe Abschnitt 5.3.3).

Überraschend häufig, in fünf von acht Testfällen, führt der Löser *KrawSubst* mit der Nutzung der einfachen Vorwärts-/Rückwärtssubstitution zum Erfolg. Er ist allerdings

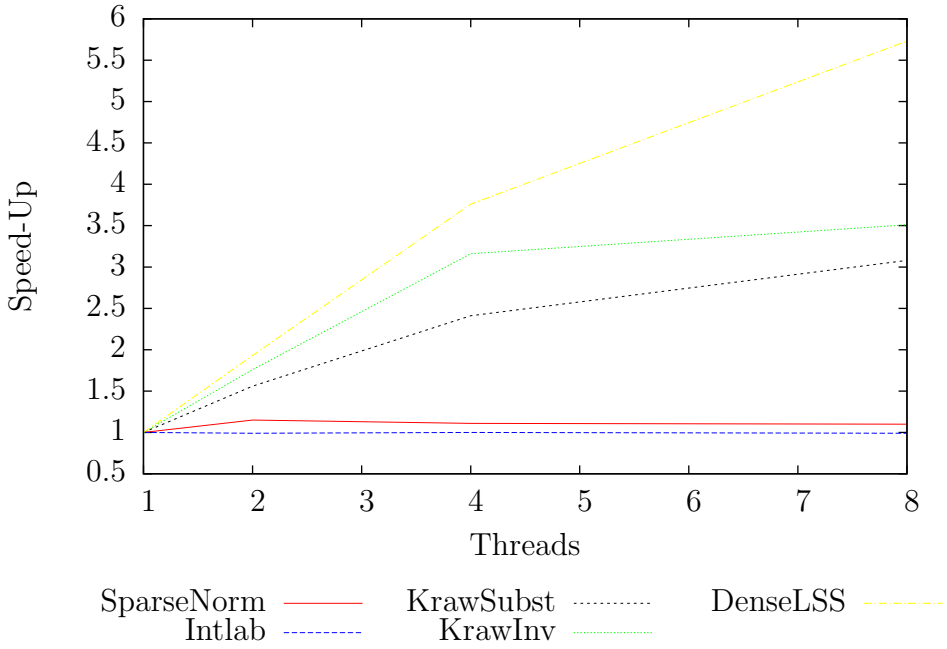


Abbildung 5.4.: Speed-Up für System mit Matrix *bcsstk18*

recht deutlich langsamer als *SparseNorm* und *Intlab*, da die Berechnung der Matrix $[LU - A]$ in der Regel sehr aufwändig ist. Andererseits lässt sich diese Berechnung wiederum sehr gut parallelisieren, wodurch der Abstand bei Benutzung von 8 Threads bereits deutlich geringer ausfällt. Der relative Fehler ist dabei durchgehend sehr gering.

Für die kleineren Matrizen *bcsstk10* und *bcsstk18* sowie die Diagonalmatrix *bcsstm05* können auch die Löser *KrawInv* und *DenseLSS* verwendet werden. Wie zu erwarten, benötigen beide deutlich mehr Zeit als die anderen Löser, profitieren aber wiederum recht stark von der Parallelisierung. Zu beachten ist auch, dass der Löser *KrawInv* auch in diesen Tests recht deutliche Laufzeitvorteile gegenüber dem allgemeinen dicht besetzten Löser aufweist. Der relative Fehler wiederum ist bei beiden nahezu ideal.

In der Folge soll der Speed-Up der verschiedenen Löser für drei exemplarische Fälle veranschaulicht werden:

- Abbildung 5.4 zeigt den Speed-Up für das System mit der kleineren Matrix *bcsstk18*, welches mit allen getesteten Lösern erfolgreich gelöst werden konnte.
- Abbildung 5.5 zeigt den Speed-Up für das System mit der großen Matrix *G3_circuit*, für welche der schnelle Verifikationsschritt zum Erfolg führt.
- Abbildung 5.6 zeigt den Speed-Up für das System mit der großen Matrix *s3dkq4m2*, für welche der schnelle Verifikationsschritt nicht zum Erfolg führt.

Abbildung 5.4 erlaubt zunächst einen allgemeinen Vergleich des Speed-Ups. Der dicht besetzte Löser zeigt hier erwartungsgemäß klar den besten Speed-Up, gefolgt vom Löser

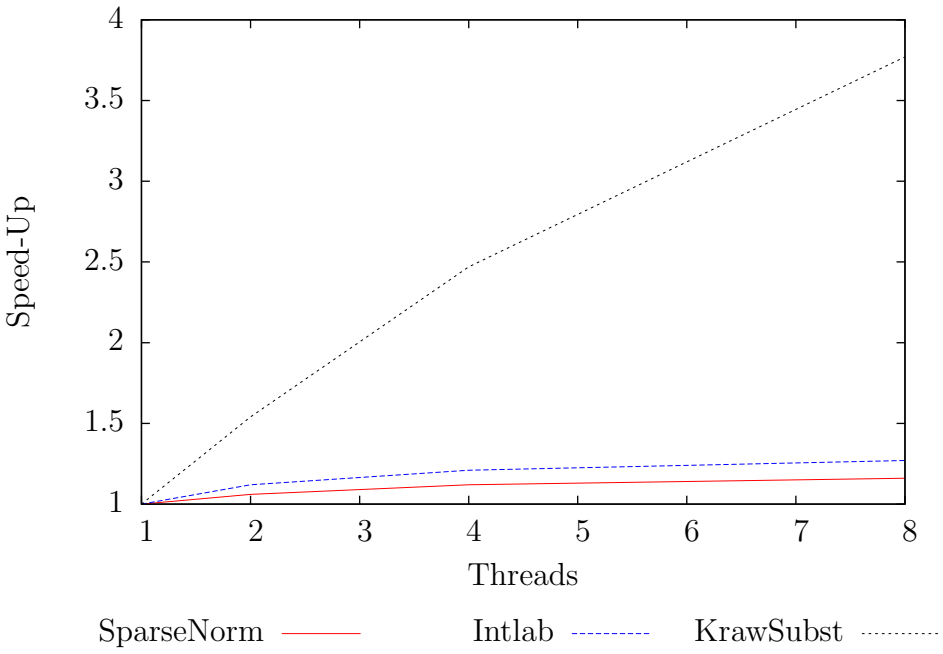


Abbildung 5.5.: Speed-Up für System mit Matrix *G3_circuit*

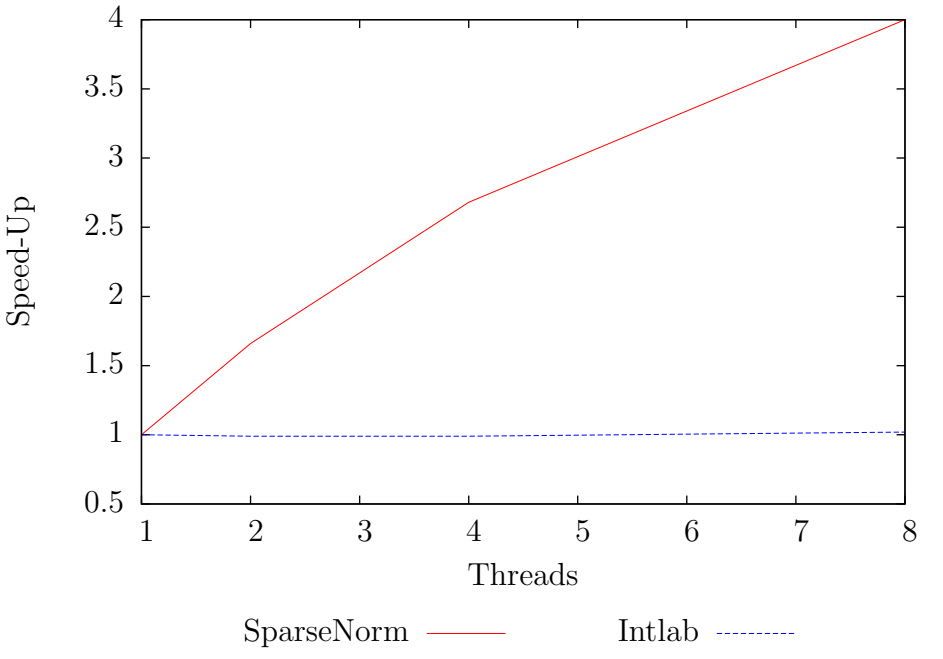


Abbildung 5.6.: Speed-Up für System mit Matrix *s3dkq4m2*

KrawInv, welcher bei Verwendung von bis zu 4 Threads ebenfalls einen guten Speed-up aufweist, der erst für weitere Threads schlechter wird. *KrawSubst* profitiert ebenfalls recht stark von der Verwendung mehrerer Threads, wobei der Speed-Up durch die Größe und Besetztheit der hier betrachteten Matrix aber noch relativ gering ausfällt. Intlab sowie *SparseNorm* können für dieses System praktisch keine Vorteile aus der Parallelisierung ziehen.

Das System *G3_circuit* zeigt einen typischen Speed-Up für große Systeme, bei denen die schnelle Verifikation zum Erfolg führt. Für *SparseNorm* sind die Vorteile dabei auf eine schnellere Cholesky-Zerlegung sowie eine leichte Beschleunigung bei der Berechnung der Näherungslösung und des Residuums beschränkt. Dieser Löser sowie *Intlab* zeigen daher wiederum nur einen leichten Speed-Up. Für *KrawSubst* hingegen steigt der Speed-Up gegenüber dem Test mit der Matrix *bcsstk18* recht deutlich an, da sich hier durch die größere Dimension der Aufwand für die Berechnungen stark erhöht.

Der Testfall aus Abbildung 5.6 schließlich zeigt eine typische Situation bei einem Scheitern der schnellen Verifikation. *Intlab* profitiert wiederum so gut wie gar nicht von der Parallelisierung. *SparseNorm* hingegen nutzt die zur Verfügung stehenden Threads für die hier nötige langsamere, aber genauere Verifikation und erreicht dadurch einen guten Speed-Up.

Als nächstes werden Tests für Intervallsysteme betrachtet. Dazu werden die gleichen Testsysteme wie zuvor verwendet, wobei die Systemmatrix und die rechte Seite um einen Faktor von 10^{-12} aufgebläht werden. Alle anderen Einstellungen und Rahmenbedingungen sind identisch zum vorherigen Test. Tabelle 5.12 zeigt die erreichten Zeiten, Tabelle 5.13 die relative Breite der berechneten Einschließungen.

Bei einigen dieser Tests trat ein Bug in Intlab auf, der den Löser mit einer Fehlermeldung abbrechen ließ. Ursache hierfür war, dass die Intlab-Funktion `mid` zur Berechnung der Mittelpunktsmatrix einer Intervallmatrix für $m \times n$ Matrizen mit $m \cdot n > 2^{31}$ einen alternativen, schnelleren Algorithmus verwendet. Dieser garantiert aber offenbar bei symmetrischen Intervallmatrizen nicht, dass auch die Mittelpunktsmatrix symmetrisch ist. Dies führt dazu, dass im Verlauf des Lösers der Test auf positive Definitheit von $\mathrm{mid}\,(A) - \tilde{\lambda} I$ mit der Fehlermeldung abbricht, die Matrix sei nicht symmetrisch. Für die hier durchgeführten Tests wurde die `mid` Funktion daher so abgeändert, dass die Mittelpunktsfunktion stehts gemäß der Formel $\underline{A} + 0.5(\overline{A} - \underline{A})$ berechnet wird, wodurch die Mittelpunktsmatrix einer symmetrischen Intervallmatrix auch stets symmetrisch ist.

Insgesamt zeigt sich erwartungsgemäß ein sehr ähnliches Bild zu den Tests mit Punktsystemen. Die Laufzeit für *KrawInv* und den Intlab-Löser ist insgesamt nur leicht angestiegen, die Verhältnisse untereinander sind aber im Wesentlichen identisch. D.h. der C-XSC Löser ist in der Regel schneller, mit Ausnahme der Systeme *bcsstk10* und *s3dkt3m2*, bei denen Intlab leichte Geschwindigkeitsvorteile hat. Bei Verwendung mehrerer Threads ist C-XSC durch die umfangreichere Parallelisierung wiederum in allen Fällen deutlich schneller.

Der Löser *KrawSubst* ist jetzt nur noch bei vier statt fünf Systemen erfolgreich. Für das System *bcsstk18* konnte keine verifizierte Einschließung mehr berechnet werden. Vermutlich wurde beim Test mit dem entsprechenden Punktsystem eine so genaue Näherungslösung berechnet, dass etliche Elemente der rechten Seite bei der Vorwärts-

Threads	**Matrix**	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
$P = 1$	bcsstk10	0.048	0.037	-	0.32	1.43
	bcsstk18	0.648	0.727	-	190.7	1450.6
	bcsstm05	0.001	0.005	0.002	0.004	0.01
	ecology2	43.1	176.4	420.6	-	-
	G3_circuit	99.3	211.5	1186.4	-	-
	s3dkq4m2	117.2	183.4	-	-	-
	s3dkt3m2	113.5	109.7	-	-	-
	thermomech_TC	4.37	9.68	19.4	-	-
$P = 8$	bcsstk10	0.036	0.037	-	0.26	0.31
	bcsstk18	0.617	0.703	-	41.6	221.6
	bcsstm05	0.001	0.005	0.001	0.007	0.004
	ecology2	39.5	174.7	137.5	-	-
	G3_circuit	83.9	168.8	312.3	-	-
	s3dkq4m2	32.8	177.4	-	-	-
	s3dkt3m2	30.1	104.6	-	-	-
	thermomech_TC	4.07	9.80	8.02	-	-

Tabelle 5.12.: Zeit in Sekunden zur verifizierten Lösung symmetrisch positiv definiter Intervallsysteme

/Rückwärtssubstitution gleich Null waren, und Überschätzungseffekte so nicht auftreten konnten. Beim hier getesteten Intervallsystem ist dies naturgemäß nicht möglich.

Interessant an den Ergebnissen ist, dass der *KrawSubst*-Löser in etlichen Tests mit Intervallsystemen schneller ist als mit den entsprechenden Punktsystemen. Ursache hierfür ist, dass für die Intervallsysteme bei der Verifikation früher eine Einschließung ins Innere erreicht wird und somit ein relativ teurer Verifikationsschritt eingespart werden kann. Insgesamt liegen die Zeiten des *KrawSubst*-Lösers aber weiterhin deutlich über denen von *SparseNorm* und *Intlab*, ebenso wie die Zeiten von *KrawInv* und *DenseLSS*. Für diese gelten analoge Aussagen zum Punktfall.

Die relative Breite der Einschließungen nimmt beim Test mit Intervallsystemen naturgemäß deutlich zu. Hier tritt auch der Nachteil der Methode der normweisen Abschätzung des Defekts bei den Lösern *SparseNorm* und *Intlab* deutlicher zu Tage, vor allem bei den schlechter konditionierten Matrizen *s3dkq4m2* und *s3dkt3m2*. Der C-XSC Löser erreicht zwar auch hier eine deutlich bessere Einschließung als Intlab. Trotzdem hat die hochgenaue Berechnung von Näherungslösung und Residuum für Intervallsysteme, auch für solche mit kleinem Radius, einen deutlich geringeren positiven Effekt auf die Brei-

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
bcsstk10	$1.4 \cdot 10^{-6}$	$1.0 \cdot 10^{3}$	-	$4.6 \cdot 10^{-11}$	$3.9 \cdot 10^{-11}$
bcsstk18	$1.5 \cdot 10^{-3}$	$2.7 \cdot 10^{3}$	-	$6.1 \cdot 10^{-12}$	$4.3 \cdot 10^{-12}$
bcsstm05	$3.4 \cdot 10^{-11}$	$1.3 \cdot 10^{-7}$	$2.0 \cdot 10^{-12}$	$2.0 \cdot 10^{-12}$	$2.0 \cdot 10^{-12}$
ecology2	$3.5 \cdot 10^{-1}$	$8.0 \cdot 10^{2}$	$6.3 \cdot 10^{-5}$	-	-
G3_circuit	$4.0 \cdot 10^{-2}$	$7.9 \cdot 10^{3}$	$2.14 \cdot 10^{-7}$	-	-
s3dkq4m2	$2.9 \cdot 10^{8}$	$1.6 \cdot 10^{10}$	-	-	-
s3dkt3m2	$5.9 \cdot 10^{8}$	$2.5 \cdot 10^{10}$	-	-	-
thermomech_TC	$6.9 \cdot 10^{-10}$	$3.6 \cdot 10^{-8}$	$2.1 \cdot 10^{-11}$	-	-

Tabelle 5.13.: Relative Breite der Einschließung bei der verifizierten Lösung symmetrisch positiv definiter Intervallsysteme

te der Einschließung. Hier zeigen sich die Vorteile des Lösers *KrawSubst*, der, sofern er anwendbar ist, durch die elementweise Fehlerabschätzung der verwendeten Methode bei Intervallsystemen eine deutlich bessere Einschließung erreicht.

Die Aussagen zum Speed-Up entsprechen im Wesentlichen denen der Tests mit Punktsystemen. Die Löser *KrawSubst* und *KrawInv* profitieren also allgemein, besonders bei größeren Systemen, recht stark von der Parallelisierung, der Löser *SparseNorm* vor allem in den Fällen, in denen die schnelle Verifikation nicht erfolgreich ist. *Intlab* hingegen kann nur sehr geringe Laufzeitgewinne durch die Parallelisierung erzielen.

Für Tests mit komplexen Systemen konnte leider im Matrix-Market und in der Sammlung der Universität Florida nur eine passende dünn besetzte, komplexe hermitesch positiv definite Matrix gefunden werden. Da ein Test mit einer Matrix $A = BB^*$ für eine allgemeine dünn besetzte komplexe Matrix B im Wesentlichen den späteren Tests für allgemeine komplexe Systeme entsprechen würde, und im letzten Abschnitt bereits ein komplexes symmetrisch positiv definites Bandsystem gelöst wurde, wird hier zunächst auf Tests mit komplexen Systemen verzichtet. Tests für allgemeine dünn besetzte komplexe Matrizen (und die einzig verfügbare hermitesch positiv definite Testmatrix) folgen im nächsten Abschnitt.

Zum Abschluss der Tests mit symmetrisch positiv definiten Systemen soll nun zur besseren Einordnung noch ein Vergleich mit einem einfachen direkten Näherungslöser für Punktsysteme vorgenommen werden. Hierzu werden alle Systeme mit Matlab über den \-Operator von Matlab gelöst. Die so erreichten Zeiten werden mit 1 gewichtet und die Zeiten der verifizierenden Löser werden entsprechend umgerechnet. Für alle Tests wurden dabei 8 Threads verwendet.

Tabelle 5.14 gibt somit den Faktor an, um den die Berechnung einer verifizierten Lösung langsamer ist als die Berechnung einer einfachen gleitkommamäßigen Näherungslösung. Zu beachten ist hierbei, dass die von Matlab berechneten Lösungen natürlich nicht verifiziert und in der Regel auch deutlich ungenauer als die von den C-XSC Lösern

berechneten Einschließungen sind.

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS	Matlab
bcsstk10	9.7	9.7	-	136.4	63.6	1.0
bcsstk18	8.3	10.7	22.0	1274.9	3075.8	1.0
bcsstm05	83.3	416.7	83.3	1000.0	250.0	1.0
ecology2	8.5	40.9	33.3	-	-	1.0
G3_circuit	5.2	10.6	22.1	-	-	1.0
s3dkq4m2	17.7	109.0	-	-	-	1.0
s3dkt3m2	24.5	98.6	-	-	-	1.0
thermomech_TC	7.2	19.6	18.1	-	-	1.0

Tabelle 5.14.: Verhältnis der Laufzeiten der verifizierenden Löser zur Laufzeit einer näherungsweisen Lösung mit Matlab

Interessant ist zunächst, dass die Laufzeiten des C-XSC Lösers *SparseNorm*, sofern die schnelle Verifikation durchführbar ist, innerhalb einer Größenordnung zu den Zeiten des einfachen Matlab-Löser bleiben (von dem hier wenig aussagekräftigen Spezialfall der sehr kleinen Diagonalmatrix *bcsstm05* abgesehen). Auch bei Anwendung des aufwändigen Verifikationsschrittes (*s3dkq4m2* und *s3dkt3m2*) bleibt der Abstand durch die effiziente Parallelisierung mit einem Faktor von 17.7 und 24.5 in einem ähnlichen Rahmen. Die von Intlab erreichten Zeiten liegen deutlich höher, insbesondere in den letztgenannten Fällen sind sie im Verhältnis bereits zwei Größenordnungen langsamer als der Matlab-Löser. Auch der *KrawSubst*-Löser erreicht durch die effektive Parallelisierung, sofern er anwendbar ist, einen vergleichsweise niedrigen zweistelligen Verlangsamungsfaktor.

Allgemeine Systeme

In diesem Abschnitt werden Tests mit allgemeinen dünn besetzten Matrizen durchgeführt. Die Testmatrizen können hier also auch nicht symmetrisch bzw. hermitesch sein und sind (bis auf eine Ausnahme) nicht positiv definit. Tabelle 5.15 zeigt eine Auflistung mit den wichtigsten Informationen zu den verwendeten Testmatrizen für den reellen Fall. Für zusätzliche Details sei wieder auf Anhang B verwiesen.

Alle Tests werden wiederum mit den Lösern *SparseNorm*, *KrawSubst* und *Intlab* durchgeführt. Für Systeme mit Dimension $n < 10000$ werden zusätzlich die Löser *KrawInv* und *DenseLSS* getestet. Ein Testsystem mit der Matrix *tub1000* ist aufgrund der geringen Bandbreite auch für die Löser *KrawBand* und *KrawBandStore* geeignet. Die zugehörigen Testergebnisse werden zur besseren Übersicht gesondert in einer eigenen Tabelle angegeben. Als rechte Seite wird bei allen Tests wiederum der Vektor $b_i = 1$, $i = 1, \ldots, n$ verwendet. Tabelle 5.16 zeigt die Laufzeiten für Testläufe mit Punktsystemen mit einem

Name	Dimension n	nnz	Bandbreite	Kondition
af23560	23560	484256	304/304	$3.5 \cdot 10^5$
cavity26	4562	138187	327/327	$1.2 \cdot 10^8$
ecology1	1000000	4996000	1000/1000	$7.6 \cdot 10^{17}$
lns_3937	3937	25407	3202/3201	$1.0 \cdot 10^{17}$
mahindas	1258	7682	1241/934	$1.0 \cdot 10^{13}$
Poisson3Db	85623	2374949	85563/85563	$1.7 \cdot 10^5$
stomach	213360	3021648	20021/19983	80
tub1000	1000	3996	2/2	$2.3 \cdot 10^6$

Tabelle 5.15.: Allgemeine dünn besetzte Testmatrizen

und acht Threads. Der relative Fehler (welcher wiederum unabhängig von der Anzahl der Threads ist) wird in Tabelle 5.17 angegeben.

Im Vergleich zwischen *SparseNorm* und *Intlab* zeigt sich zunächst wieder, dass der C-XSC Löser in nahezu allen Fällen auch bei Nutzung von nur einem Thread (zum Teil deutlich) schneller ist als *Intlab*. Einzige Ausnahme ist das sehr kleine Testsystem mit der Matrix *tub1000*, bei dem *Intlab* leichte Vorteile hat. Für die Matrix *cavity26* erklärt sich der Vorteil dadurch, dass *SparseNorm* mit der schnellen Verifikation erfolgreich ist, *Intlab* hingegen nicht. Dies ist durch die anfängliche Skalierung der Systemmatrix im C-XSC Löser zu erklären, welche *Intlab* nicht vornimmt.

In den anderen Fällen (*af23560*, *Poisson3Db* und *stomach*) hingegen sind beide Löser mit der schnellen Verifikation erfolgreich. Die großen Laufzeitunterschiede werden hier, wie im positiv definiten Fall, durch die, vermutlich durch einen Bug in Intlab oder Matlab verursachte, ungewöhnlich zeitintensive Berechnung der Näherungslösung und der Näherung des kleinsten Singulärwertes verursacht. Theoretisch wäre auch hier eigentlich zu erwarten, dass beide Löser bei Nutzung eines Threads in etwa auf einem Niveau liegen.

Durch die Vorab-Skalierung der Systemmatrix ist der *SparseNorm*-Löser auch in der Lage, für Systeme mit den Matrizen *lns_3937* und *mahindas* erfolgreich verifizierte Lösungseinschließungen zu berechnen. *Intlab* kann hier aufgrund der relativ hohen Kondition dieser Matrizen keine Einschließung berechnen. Das System mit der Matrix *ecology1* (Konditionszahl 10^{17}) hingegen kann von keinem der dünn besetzten Löser verifiziert gelöst werden.

Bei Verwendung von acht Threads zeigt *SparseNorm* ein ähnliches Verhalten wie bei symmetrisch positiv definiten Matrizen. Ist die schnelle Verifikation erfolgreich, so ist der Geschwindigkeitsgewinn relativ gering. Durch Parallelisierungen in LU- und Cholesky-Zerlegung sowie bei der Berechnung von Näherungslösung und Residuum kann, je nach Struktur der Systemmatrix, trotzdem etwas Laufzeit gewonnen werden (z.B. bei *Poisson3Db*). Andernfalls kann durch eine Parallelisierung der aufwändigen Berechnung des

Threads	**Matrix**	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
$P = 1$	af23560	7.37	10.5	-	-	-
	cavity26	0.76	3.9	-	60.8	64.5
	ecology1	-	-	-	-	-
	lns_3937	1.21	-	-	19.1	42.4
	mahindas	0.049	-	0.032	0.26	1.65
	Poisson3Db	194.4	564.8	2037.6	-	-
	stomach	97.4	297.9	802.6	-	-
	tub1000	0.017	0.012	0.018	0.30	0.87
$P = 8$	af23560	6.36	9.09	-	-	-
	cavity26	0.61	3.78	-	17.9	10.3
	ecology1	-	-	-	-	-
	lns_3937	0.49	-	-	6.58	7.67
	mahindas	0.047	-	0.027	0.29	0.32
	Poisson3Db	95.6	267.9	349.1	-	-
	stomach	70.4	236.9	168.5	-	-
	tub1000	0.015	0.012	0.011	0.31	0.18

Tabelle 5.16.: Zeit in Sekunden zur verifizierten Lösung allgemeiner Punktsysteme

Fehlers der verschobenen Cholesky-Zerlegung ein größerer Gewinn erzielt werden (dieser Schritt wird hier nur für das System mit der Matrix *lns_3937* durchgeführt).

Intlab profitiert wiederum im Wesentlichen durch Parallelisierungen bei der Cholesky-Zerlegung und dem näherungsweisen Lösen der Punkt-Dreieckssysteme im Algorithmus. Letzteres scheint insbesondere bei *Poisson3Db* für den vergleichsweise großen Laufzeitgewinn verantwortlich zu sein.

Der relative Fehler von *SparseNorm* ist, sofern eine Lösung berechnet werden kann, durch die hochgenaue Berechnung von Näherungslösung und Residuum in allen Fällen nahezu ideal. Intlab hingegen liefert deutlich breitere Einschließungen, da hier diese Möglichkeit nicht besteht.

Der Löser *KrawSubst* ist in vier von acht Testfällen erfolgreich. Wie zuvor ist die Laufzeit für größere Systeme durch die Berechnung von $[LU - A]$ vergleichsweise hoch. Durch die Parallelisierung dieser Berechnung wird allerdings auch ein guter Speed-Up erzielt, wodurch der Abstand zu den anderen Lösern bei Verwendung von acht Threads deutlich geringer ist. Der relative Fehler ist auch hier, wie zu erwarten, sehr gut und liegt in der Größenordnung 10^{-17}.

Die Löser *KrawInv* und *DenseLSS* sind, sofern sie aufgrund der Dimension des Systems

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
af23560	$2.2 \cdot 10^{-16}$	$1.7 \cdot 10^{-4}$	-	-	-
cavity26	$2.4 \cdot 10^{-16}$	$1.8 \cdot 10^{1}$	-	$8.0 \cdot 10^{-17}$	$4.9 \cdot 10^{-16}$
ecology1	-	-	-	-	-
lns_3937	$2.2 \cdot 10^{-16}$	-	-	$7.1 \cdot 10^{-17}$	$5.6 \cdot 10^{-16}$
mahindas	$2.1 \cdot 10^{-16}$	-	$5.5 \cdot 10^{-17}$	$5.5 \cdot 10^{-17}$	$6.3 \cdot 10^{-16}$
Poisson3Db	$2.3 \cdot 10^{-16}$	$1.0 \cdot 10^{-5}$	$7.8 \cdot 10^{-17}$	-	-
stomach	$1.9 \cdot 10^{-16}$	$1.1 \cdot 10^{-9}$	$6.4 \cdot 10^{-17}$	-	-
tub1000	$2.2 \cdot 10^{-16}$	$7.7 \cdot 10^{-2}$	$7.3 \cdot 10^{-17}$	$7.3 \cdot 10^{-17}$	$1.8 \cdot 10^{-16}$

Tabelle 5.17.: Relativer Fehler bei der verifizierten Lösung allgemeiner Punktsysteme

anwendbar sind, erwartungsgemäß deutlich langsamer, profitieren aber auch vergleichsweise stark von der Parallelisierung. Der relative Fehler ist auch hier bei beiden nahezu ideal.

Für das System mit der Matrix *tub1000* können auch die beiden Bandlöser *KrawBand* und *KrawBandStore* verwendet werden. Tabelle 5.18 zeigt die dabei erzielten Testergebnisse. Wie zuvor sind die Bandlöser um bis zu eine Größenordnung langsamer als die anderen dünn besetzten Löser und profitieren nur wenig von mehreren Threads. Die Ergebnisqualität liegt aber auf einem ähnlich hohen Niveau wie bei den anderen C-XSC Lösern.

Matrix	KrawBand	KrawBandStore
Zeit, $P = 1$	0.12	0.075
Zeit, $P = 8$	0.11	0.068
Relativer Fehler	$7.3 \cdot 10^{-17}$	$7.3 \cdot 10^{-17}$

Tabelle 5.18.: Testergebnisse für Bandlöser bei Punktsystem mit Matrix *tub1000*

Zur Veranschaulichung des Speed-Ups werden folgende drei Testfälle näher betrachtet:

- *lns_3937* als Beispiel für ein System bei dem der langsame Verifikationsschritt durchgeführt werden muss und als Vergleich zu *KrawInv* und *DenseLSS* (Abbildung 5.7).
- *af23560* für den Vergleich für ein größeres System zwischen *Intlab* und *SparseNorm* (Abbildung 5.8).
- *stomach* für den Vergleich von *SparseNorm*, *Intlab* und *KrawSubst* für ein größeres System (Abbildung 5.9).

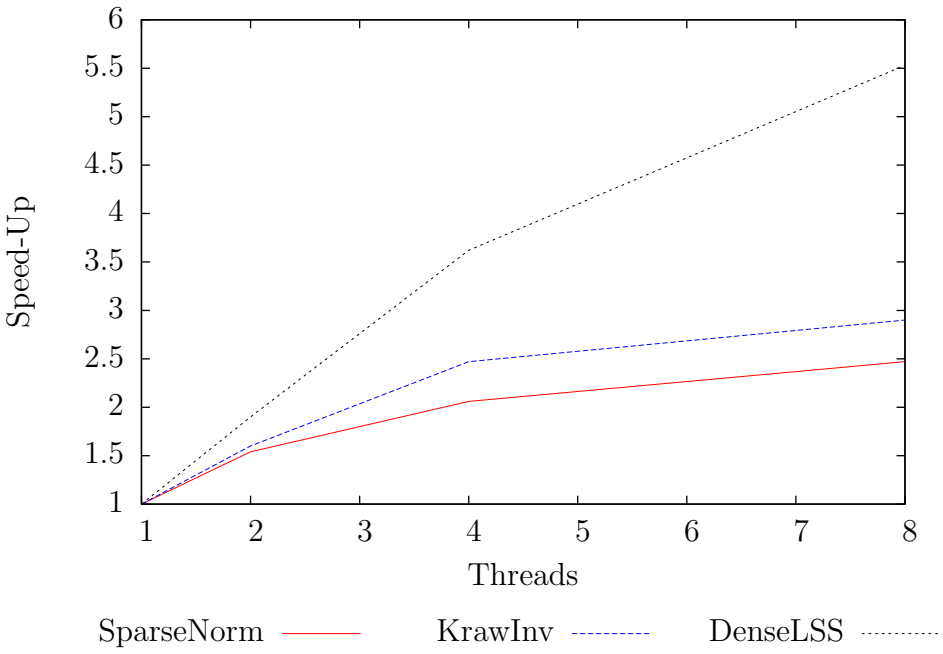


Abbildung 5.7.: Speed-Up für System mit Matrix *lns_3937*

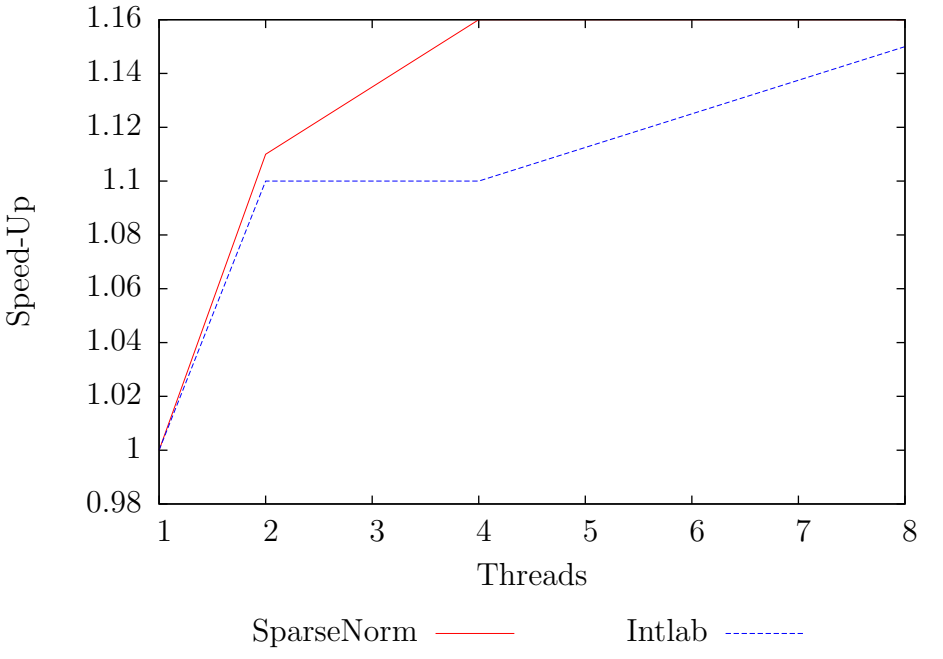


Abbildung 5.8.: Speed-Up für System mit Matrix *af23560*

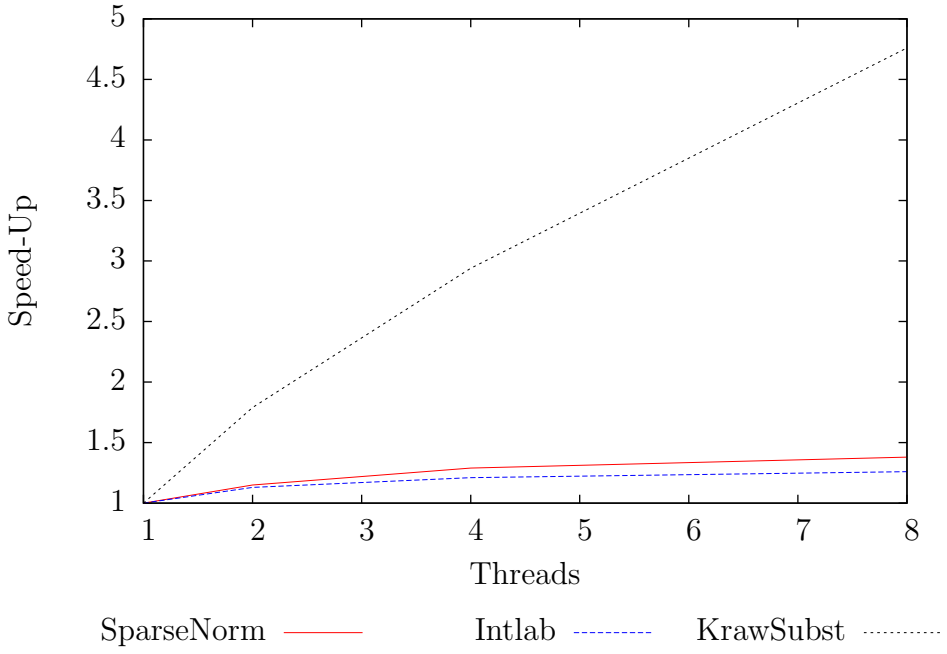


Abbildung 5.9.: Speed-Up für System mit Matrix *stomach*

Für das System *lns_3937* zeigt sich ein relativ guter Speed-Up für den Löser *SparseNorm* und zwei Threads, welcher durch die parallele Berechnung des langsamen Verifikationsschrittes entsteht. Für vier und insbesondere acht Threads ist der Anstieg nicht mehr so stark, was im Wesentlichen an der vergleichsweise kleinen Dimension des Testsystems liegt. Ähnliches gilt für den Löser *KrawInv*. Der voll besetzte Löser muss hier bereits deutlich mehr Aufwand betreiben, wodurch auch der Speed-Up besser ausfällt.

Der Test mit dem System *af23560* zeigt das typische Verhalten der Löser *SparseNorm* und *Intlab* bei Erfolg der schnellen Verifikation. Durch die relativ geringen Ansatzpunkte für eine Parallelisierung ist der Speed-Up hier gering. Durch die Struktur der Matrix nimmt er weiterhin für mehr als zwei Threads nicht mehr wesentlich zu.

Ähnliches gilt für das System *stomach*, wobei hier der Vergleich mit dem Löser *KrawSubst* durchgeführt werden kann. Da dieser immer die gut parallelisierbare Berechnung $[LU - A]$ durchführen muss, welche einen großen Teil des Gesamtaufwands ausmacht, ist der Speed-Up hier wesentlich höher. Auch beim Sprung von vier auf acht Threads ergibt sich noch ein deutlicher Geschwindigkeitszuwachs.

Nach den Tests mit Punktsystemen folgt nun ein entsprechender Test mit allgemeinen Intervallsystemen. Dazu werden wieder die gleichen Testsysteme wie zuvor verwendet, wobei die Systemmatrix und die rechte Seite um einen Faktor 10^{-12} aufgebläht werden. Tabelle 5.19 zeigt die erzielten Laufzeiten, Tabelle 5.20 die relative Breite der berechneten Einschließungen (wiederum unabhängig von der Anzahl der verwendeten Threads).

Die Verhältnisse der Laufzeiten untereinander sind im Wesentlichen unverändert, die Zeiten der Löser *SparseNorm*, *Intlab* und *DenseLSS* sind durch den zusätzlichen Aufwand wegen des Übergangs auf Intervallsysteme leicht angestiegen. Im Unterschied zu

Threads	**Matrix**	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
$P = 1$	af23560	7.87	11.1	-	-	-
	cavity26	0.89	4.12	-	41.26	84.3
	ecology1	-	-	-	-	-
	lns_3937	1.22	-	-	19.2	55.5
	mahindas	0.056	-	0.042	0.19	2.13
	Poisson3Db	196.9	573.2	1971.8	-	-
	stomach	99.2	302.6	741.5	-	-
	tub1000	0.020	-	0.021	0.31	1.10
$P = 8$	af23560	6.89	9.47	-	-	-
	cavity26	0.76	3.79	-	12.32	14.2
	ecology1	-	-	-	-	-
	lns_3937	0.51	-	-	6.69	9.39
	mahindas	0.055	-	0.033	0.21	0.42
	Poisson3Db	97.7	272.7	336.6	-	-
	stomach	72.9	240.6	156.3	-	-
	tub1000	0.018	-	0.015	0.33	0.24

Tabelle 5.19.: Zeit in Sekunden zur verifizierten Lösung allgemeiner Intervallsysteme

den Punktsystemen kann *Intlab* im Intervallfall für das System *tub1000* keine verifizierte Einschließung mehr berechnen.

Die Krawczyk-Operator basierten Löser *KrawSubst* und *KrawInv* sind in den meisten Fällen sogar etwas schneller als im Punktfall, da für die Intervallsysteme ein Verifikationsschritt weniger nötig ist als für die entsprechenden Punktsysteme.

Tabelle 5.21 zeigt die Testergebnisse für das Lösen des Systems *tub1000* mit den Bandlösern. Auch hier sind die Zeiten im Vergleich zum Punktfall zurückgegangen, da im Intervallfall ein Verifikationsschritt weniger nötig ist. Insgesamt sind die Zeiten der Bandlöser im Vergleich zu den übrigen dünn besetzten Lösern aber erneut recht hoch.

Die relative Breite der berechneten Einschließungen steigt für die Löser *SparseNorm* und *Intlab*, aufgrund der verwendeten normweisen Abschätzung, im Intervallfall recht stark an. Der C-XSC Löser erreicht aber erneut einen deutlich bessere Einschließung als *Intlab*, zum einen da bei der verwendeten kleinen Intervallbreite die Qualität der Näherungslösung noch eine recht starke Rolle spielt, zum anderen da die Kondition (und damit die berechnete Fehlerschranke) durch die nur in *SparseNorm* verwendete Skalierung oftmals reduziert wird.

Wie zu erwarten weisen die Krawczyk-Operator basierten Löser für Intervallsyste-

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
af23560	$1.1 \cdot 10^{-4}$	$1.4 \cdot 10^{-1}$	-	-	-
cavity26	$1.0 \cdot 10^{-4}$	$1.2 \cdot 10^{6}$	-	$3.3 \cdot 10^{-6}$	$3.2 \cdot 10^{-9}$
ecology1	-	-	-	-	-
lns_3937	$1.9 \cdot 10^{8}$	-	-	$2.7 \cdot 10^{-2}$	$1.9 \cdot 10^{-9}$
mahindas	$3.1 \cdot 10^{-2}$	-	$6.4 \cdot 10^{-7}$	$2.7 \cdot 10^{-10}$	$5.4 \cdot 10^{-11}$
Poisson3Db	$8.6 \cdot 10^{-7}$	$1.3 \cdot 10^{-2}$	$4.2 \cdot 10^{-8}$	-	-
stomach	$1.1 \cdot 10^{-8}$	$1.1 \cdot 10^{-6}$	$6.4 \cdot 10^{-12}$	-	-
tub1000	$7.0 \cdot 10^{0}$	-	$5.3 \cdot 10^{-6}$	$4.7 \cdot 10^{-7}$	$3.0 \cdot 10^{-7}$

Tabelle 5.20.: Relative Breite der Einschließungen bei der verifizierten Lösung allgemeiner Intervallsysteme

Matrix	KrawBand	KrawBandStore
Zeit, $P = 1$	0.087	0.066
Zeit, $P = 8$	0.083	0.061
Relative Breite	$3.8 \cdot 10^{-6}$	$3.8 \cdot 10^{-6}$

Tabelle 5.21.: Testergebnisse für Bandlöser bei Intervallsystem mit Matrix *tub1000*

me durchweg eine (oftmals deutlich) bessere relative Breite auf, da in dieser Methode elementweise Fehlerschranken berechnet werden. Der Löser *KrawSubst* zeigt für die Systeme *mahindas* und *tub1000* eine etwas größere relative Breite der Einschließung als die Löser *KrawInv* und *DenseLSS*. Dies lässt darauf schließen, dass es bei der Vorwärts-/Rückwärtssubstitution zu Überschätzungen kommt, welche aber klein genug bleiben, um trotzdem eine Einschließung ins Innere im Verifikationsschritt zu erreichen.

Für schlechter konditionierte Systeme, z.B. *lns_3937*, zeigt der Löser *KrawInv* eine deutlich höheren relative Breite der berechneten Einschließungen als *DenseLSS*. Dieser Unterschied wird zum einen durch die Symmetrisierung des jeweiligen Dreieckssystems bei dieser Methode sowie durch die dann recht starken Rundungsfehler bei der Berechnung des Betrages der Inversen der Dreiecksmatrix verursacht (siehe Abschnitt 5.1.1).

Bei Verwendung mehrerer Threads zeigt sich ein ähnliches Verhalten wie im Punktfall. Der Löser *SparseNorm* profitiert deutlich stärker als *Intlab*, bei einem Erfolg der schnellen Verifikation ist der Laufzeitgewinn aber oft relativ gering. Der Löser *KrawSubst* profitiert durch die Parallelisierung der Berechnung von $[LU - \boldsymbol{A}]$ sehr deutlich von der Verwendung mehrerer Threads. Insgesamt gelten bezüglich des Speed-Ups analoge Aussagen zum Punktfall.

Zum Abschluss der Tests mit Testmatrizen aus praktischen Anwendungen werden nun

Testfälle mit komplexen Systemen betrachtet. Tabelle 5.22 zeigt die wichtigsten Informationen zu den verwendeten Testmatrizen. Die Testmatrix *mhd1280b* ist hierbei die einzig verfügbare hermitesch positiv definite Matrix. Alle anderen Matrizen erfüllen diese Bedingung nicht. Für genauere Details zu den Testmatrizen sei wiederum auf Anhang B verwiesen.

Name	Dimension n	nnz	Bandbreite	Kondition
ABACUS_shell_hd	23412	218484	379/379	$2.0 \cdot 10^{16}$
kim1	38415	933195	14763/14763	$1.9 \cdot 10^{4}$
light_in_tissue	29282	406084	14763/14763	$1.1 \cdot 10^{4}$
mhd1280b	1280	22778	43/43	$6.0 \cdot 10^{12}$
mplate	5962	142190	3354/3354	$3.4 \cdot 10^{16}$
qc324	324	26730	81/81	$7.4 \cdot 10^{4}$
waveguide3D	21036	303468	20882/20882	$1.0 \cdot 10^{4}$
young1c	841	4089	29/29	$2.9 \cdot 10^{2}$

Tabelle 5.22.: Komplexe dünn besetzte Testmatrizen

Zunächst werden mit den gleichen Lösern wie zuvor Systeme mit diesen Testmatrizen und dem Vektor mit allen Elementen gleich eins als rechter Seite durchgeführt. Die Löser *KrawInv* und *DenseLSS* werden wieder nur für Systeme mit Dimension $n < 10000$ verwendet. Zunächst werden Punktsysteme betrachtet. Tabelle 5.23 gibt die erzielten Laufzeiten für einen und acht Threads, Tabelle 5.24 den relativen Fehler der berechneten Einschließungen (unabhängig von der Anzahl der Threads) an. Wie zuvor wurde der relative Fehler getrennt für Real- und Imaginärteil bestimmt, der in der Tabelle angegebene Wert ist jeweils der größere der beiden Fehler.

Zunächst fällt auf, dass Intlab in keinem der Tests erfolgreich eine verifizierte Einschließung berechnen konnte. Ursache hierfür ist, dass die Funktion `isspd`, welcher der Löser aufruft, um die schnelle Verifikation durchzuführen, wie bei den Tests mit Bandmatrizen stets mit der (falschen) Fehlermeldung abbricht, die getestete Matrix sei nicht hermitesch. Anscheinend handelt es sich hierbei um einen Bug in Intlab. Wird die Abfrage entfernt, so bricht der Intlab-Löser trotzdem für einen Vielzahl von Systemen mit einem Fehler in der Funktion `midrad` ab. Für die wenigen getesteten Systeme, für welche nicht mit einer Fehlermeldung abgebrochen wird, liefert der Löser stets `NaN` als Ergebnis zurück.

Der Löser *SparseNorm* hingegen, welcher prinzipiell den gleichen Lösungsalgorithmus wie *Intlab* verwendet, bestimmt bei sechs der acht getesteten Systeme erfolgreich eine verifizierte Einschließung. Ausnahmen sind die Systeme *ABACUS_shell_hd* und *mplate*, welche beide Konditionszahlen der Größenordnung 10^{16} aufweisen. Das System *ABACUS_shell_hd* kann mit dem Löser *KrawSubst* erfolgreich gelöst werden, für *mplate* hingegen ist nur der dicht besetzte Löser *DenseLSS* erfolgreich.

Threads	**Matrix**	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
$P = 1$	ABACUS_shell_hd	-	-	18.0	-	-
	kim1	33.6	-	-	-	-
	light_in_tissue	12.4	-	27.9	-	-
	mhd1280b	0.161	-	0.447	0.476	7.39
	mplate	-	-	-	-	553.8
	qc324	0.485	-	-	0.59	0.15
	waveguide3D	12.9	-	-	-	-
	young1c	0.128	-	-	0.286	1.93
$P = 8$	ABACUS_shell_hd	-	-	9.10	-	-
	kim1	26.1	-	-	-	-
	light_in_tissue	10.4	-	13.7	-	-
	mhd1280b	0.126	-	0.346	0.385	1.21
	mplate	-	-	-	-	80.4
	qc324	0.354	-	-	0.19	0.033
	waveguide3D	9.89	-	-	-	-
	young1c	0.126	-	-	0.275	0.33

Tabelle 5.23.: Zeit in Sekunden zur verifizierten Lösung komplexer Punktsysteme

Wie zuvor ist der Löser *SparseNorm* allgemein am schnellsten, allerdings profitiert *KrawSubst* stärker von der Parallelisierung und erzielt daher bei Verwendung von acht Threads ebenfalls ordentliche Laufzeiten. Bei den hier verwendeten Testsystemen war stets die schnelle Verifikation von *SparseNorm* erfolgreich, wodurch der Speed-Up für diesen Löser meist recht gering ausfällt.

Allgemein gelten für den Speed-Up ähnliche Aussagen wie im reellen Fall. Abbildung 5.10 zeigt den Speed-Up für das System *light_in_tissue*. Da bei diesem der schnelle Verifikationsschritt von *SparseNorm* erfolgreich ist, gibt es wenige Ansatzpunkte für eine Parallelisierung, der Speed-Up fällt entsprechend gering aus. Der Löser *KrawSubst* zeigt wiederum einen stärkeren Speed-Up durch die parallele Berechnung der Iterationsmatrix. Abbildung 5.11 zeigt den Speed-Up für das hermitesch positiv definite System *mhd1280b*. Da es sich um ein relativ kleines dünn besetztes System handelt, ist der Speed-Up für alle dünn besetzten Löser sehr niedrig. Der dicht besetzte Löser zeigt den aus Abschnitt 4.1.4 bekannten guten Speed-Up.

Der relative Fehler der dünn besetzten Löser ist jeweils für alle Systeme, bei denen erfolgreich eine verifizierte Einschließung berechnet werden konnte, nahezu ideal. Der recht hohe relative Fehler des dicht besetzten Lösers für das System *mplate* kommt

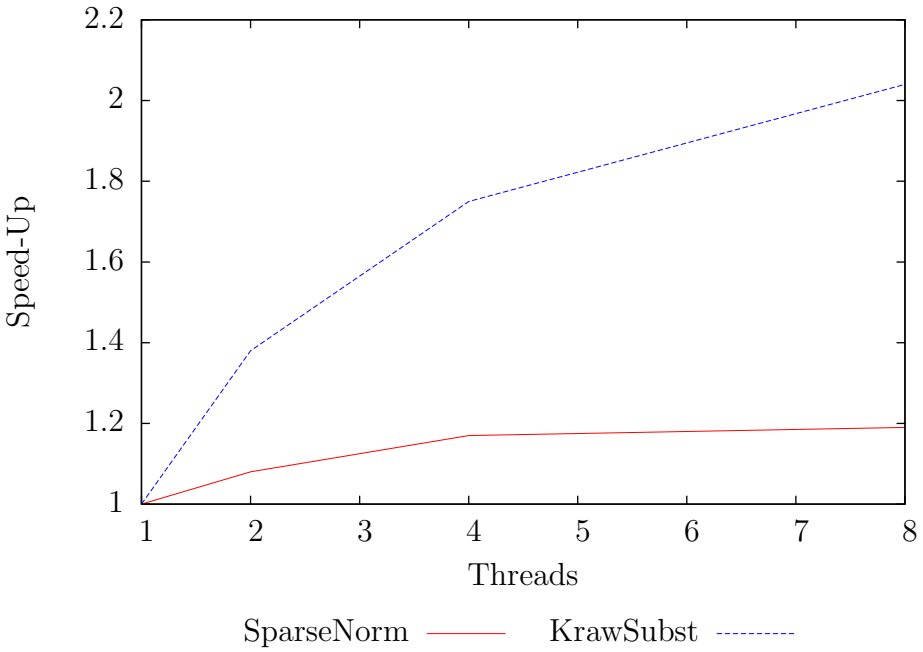


Abbildung 5.10.: Speed-Up für System mit Matrix *light_in_tissue*

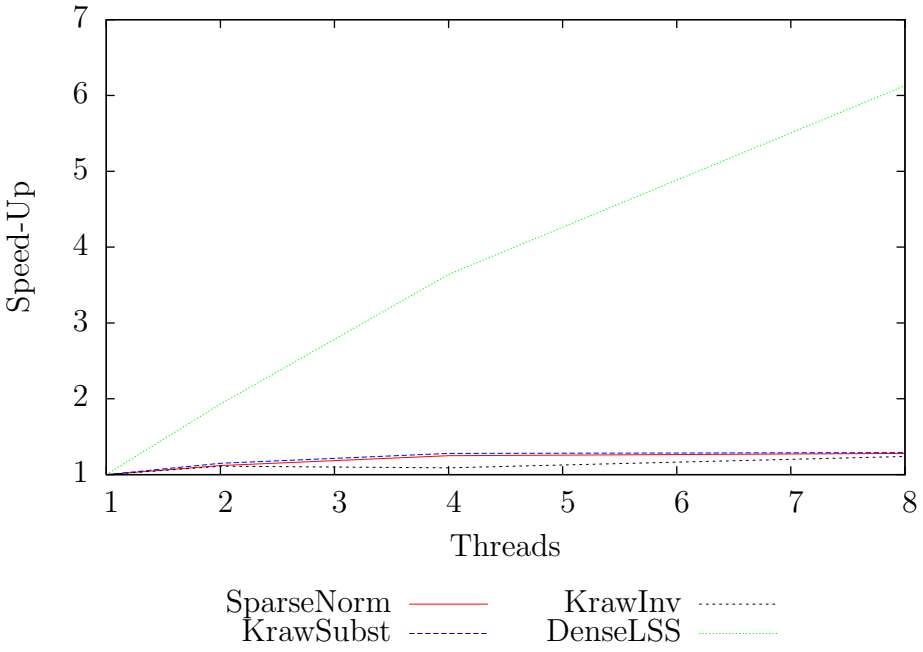


Abbildung 5.11.: Speed-Up für System mit Matrix *mhd1280b*

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
ABACUS_shell_hd	-	-	$7.6 \cdot 10^{-17}$	-	-
kim1	$4.2 \cdot 10^{-20}$	-	-	-	-
light_in_tissue	$2.5 \cdot 10^{-16}$	-	$8.3 \cdot 10^{-17}$	-	-
mhd1280b	$2.1 \cdot 10^{-16}$	-	$7.0 \cdot 10^{-17}$	$7.0 \cdot 10^{-17}$	$1.7 \cdot 10^{-15}$
mplate	-	-	-	-	$7.3 \cdot 10^{-9}$
qc324	$2.4 \cdot 10^{-16}$	-	-	$7.9 \cdot 10^{-17}$	$1.6 \cdot 10^{-15}$
waveguide3D	$2.4 \cdot 10^{-16}$	-	-	-	-
young1c	$1.5 \cdot 10^{-18}$	-	-	$5.0 \cdot 10^{-19}$	$2.3 \cdot 10^{-17}$

Tabelle 5.24.: Relativer Fehler bei der verifizierten Lösung komplexer Punktsysteme

durch dessen hohe Kondition zustande.

Wie bei den Tests im Reellen werden für Tests mit Intervallsystemen die zuvor verwendeten Testsysteme um einen Faktor 10^{-12} aufgebläht und mit den gleichen Lösern unter den gleichen Rahmenbedingungen gelöst. Tabelle 5.25 zeigt die erzielten Laufzeiten, Tabelle 5.26 die relative Breite der berechneten Einschließungen.

Wie zuvor wächst die Laufzeit für den Löser *SparseNorm* für Intervallsysteme zwar an, aber nur leicht, da der Mehraufwand recht gering ist. Der Löser *KrawSubst* wird sogar etwas schneller, da für die Intervallsysteme ein aufwändiger Verifikationsschritt weniger nötig ist als für die entsprechenden Punktsysteme. Für das System *ABACUS_shell_hd* kann nun auch *KrawSubst* keine Einschließung mehr berechnen, da die aufgeblähte Intervallmatrix vermutlich deutlich schlechter konditionierte oder auch singuläre Matrizen mit einschließt, wodurch der Krawczyk-Operator nicht mehr zum Erfolg führt. Ansonsten bleiben die wesentlichen Aussagen aus dem Punktfall, auch hinsichtlich der Parallelisierung, bestehen.

Die relative Breite der berechneten Einschließungen ist, wie zu erwarten, bei den Lösern, welche auf dem Krawczyk-Operator basieren, für die Intervallsysteme deutlich kleiner als bei Verwendung von *SparseNorm*. Trotzdem berechnet auch dieser durchweg noch aussagekräftige Einschließungen.

Zur besseren Einordnung der gemessenen Laufzeiten wird ein Vergleich mit einem einfachen Näherungslöser durchgeführt. Dazu werden einige der allgemeinen und komplexen Testsysteme mit dem \-Operator von Matlab gelöst. Die dabei erzielten Zeiten werden mit 1 gewichtet, die Zeiten der anderen Löser werden entsprechend umgerechnet. Dabei werden für alle Tests acht Threads verwendet. Tabelle 5.14 zeigt die Ergebnisse.

Die meisten hier getesteten Systeme sind vergleichsweise groß. Für diese erzielt der Löser *SparseNorm* im Allgemeinen Zeiten, die (im Reellen) innerhalb einer Größenordnung im Vergleich zum Matlab-Näherungslöser oder (im Komplexen) nur wenig darüber liegen. Eine Ausnahme ist hier das relativ kleine System *mhd1280b*, bei welchem der Abstand deutlich größer ist. Der Löser *KrawSubst* benötigt im Allgemeinen mehr Zeit als

Threads	**Matrix**	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
$P = 1$	ABACUS_shell_hd	-	-	-	-	-
	kim1	35.5	-	-	-	-
	light_in_tissue	13.3	-	21.9	-	-
	mhd1280b	0.202	-	0.354	0.373	9.49
	mplate	-	-	-	293.0	740.1
	qc324	0.525	-	-	0.459	0.211
	waveguide3D	13.6	-	-	-	-
	young1c	0.135	-	-	0.221	2.62
$P = 8$	ABACUS_shell_hd	-	-	-	-	-
	kim1	28.1	-	-	-	-
	light_in_tissue	11.7	-	10.8	-	-
	mhd1280b	0.177	-	0.278	0.290	1.81
	mplate	-	-	-	60.5	129.5
	qc324	0.415	-	-	0.178	0.055
	waveguide3D	10.7	-	-	-	-
	young1c	0.137	-	-	0.200	0.56

Tabelle 5.25.: Zeit in Sekunden zur verifizierten Lösung komplexer Intervallsysteme

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS
ABACUS_shell_hd	-	-	-	-	-
kim1	$7.8 \cdot 10^{-8}$	-	-	-	-
light_in_tissue	$1.3 \cdot 10^{-4}$	-	$4.2 \cdot 10^{-9}$	-	-
mhd1280b	$1.1 \cdot 10^{-3}$	-	$3.5 \cdot 10^{-8}$	$1.5 \cdot 10^{-10}$	$7.7 \cdot 10^{-10}$
mplate	-	-	-	$1.1 \cdot 10^{-8}$	$1.1 \cdot 10^{-5}$
qc324	$2.9 \cdot 10^{-5}$	-	-	$8.5 \cdot 10^{-12}$	$1.7 \cdot 10^{-8}$
waveguide3D	$4.8 \cdot 10^{-5}$	-	-	-	-
young1c	$4.25 \cdot 10^{-9}$	-	-	$5.7 \cdot 10^{-10}$	$1.4 \cdot 10^{-12}$

Tabelle 5.26.: Relative Breite der Einschließungen bei der verifizierten Lösung komplexer Intervallsysteme

Matrix	SparseNorm	Intlab	KrawSubst	KrawInv	DenseLSS	Matlab
af23560	7.9	11.3	-	-	-	1.0
cavity26	9.1	56.5	-	267.6	153.9	1.0
poisson3Db	7.0	19.6	25.5	-	-	1.0
stomach	7.1	23.7	16.9	-	-	1.0
mhd1280b	52.5	-	144.2	160.4	504.2	1.0
ABACUS_shell_hd	-	-	7.8	-	-	1.0
kim1	16.2	-	-	-	-	1.0
light_in_tissue	17.6	-	23.8	-	-	1.0
waveguide3D	12.2	-	-	-	-	1.0

Tabelle 5.27.: Verhältnis der Laufzeiten der verifizierenden Löser zur Laufzeit einer näherungsweisen Lösung mit Matlab für allgemeine und komplexe Matrizen

SparseNorm, ist oftmals aber nicht wesentlich langsamer. In den meisten Fällen bleibt die Laufzeit innerhalb von zwei Größenordnungen im Vergleich zum Näherungslöser.

5.3.3. Sonstige Tests

Es werden nun einige spezielle Tests mit den dünn besetzten C-XSC Lösern besprochen. Zunächst soll dabei kurz auf die Auswirkungen von Optionen eingegangen werden, welche standardmäßig deaktiviert sind. In Abschnitt 5.1.1 wurden einige Möglichkeiten angesprochen, mit denen die Löser auf Basis des dünn besetzten Krawczyk-Operators beschleunigt werden können, wobei im Allgemeinen aber mit weniger genauen Einschließungen oder sogar mit einem Scheitern der Verifikation zu rechnen ist. Um deren Auswirkungen genauer zu untersuchen, werden die Tests mit dem Löser *KrawSubst* für einige der bereits im letzten Abschnitt genutzten Testsysteme wiederholt, wobei folgende Optionen von der Standardeinstellung abweichen:

- Die hochgenaue Berechnung der Näherungslösung und des Residuums werden deaktiviert (die Option `highPrecApprxSolution` wird auf `false` gesetzt).
- Die schnelle, aber ungenauere Berechnung von $[LU-A]$ nach Satz 5.3 wird aktiviert (die Option `fastDecompErrorComputation` wird auf `true` gesetzt).
- Der schnelle Verifikationsmodus, welcher die Anzahl der Verifikationsschritte auf 2 beschränkt, wird aktiviert (die Option `fastVerification` wird auf `true` gesetzt).

Die alten und neuen Laufzeiten und relativen Fehler für die neu getesteten Systeme werden in Tabelle 5.28 angegeben. Alle Tests wurden dabei mit acht Threads durchgeführt.

Wie zu sehen ist, verringert sich die benötigte Zeit jeweils in etwa um ein Drittel. Allerdings steigt der relative Fehler an. Für besser konditionierte Systeme, wie z.B.

Matrix	Standardoptionen		Schnelle Optionen	
	Zeit	Relativer Fehler	Zeit	Relativer Fehler
ABACUS_shell_hd	9.10	$7.6 \cdot 10^{-17}$	-	-
ecology2	139.8	$7.1 \cdot 10^{-17}$	90.9	$1.6 \cdot 10^{-9}$
G3_circuit	342.8	$8.1 \cdot 10^{-17}$	215.3	$2.9 \cdot 10^{-12}$
light_in_tissue	13.7	$2.5 \cdot 10^{-16}$	-	-
poisson3Db	349.1	$7.8 \cdot 10^{-17}$	213.8	$6.5 \cdot 10^{-15}$
stomach	168.5	$6.4 \cdot 10^{-17}$	110.8	$1.3 \cdot 10^{-15}$
thermomech_TC	8.61	$8.1 \cdot 10^{-17}$	6.02	$2.9 \cdot 10^{-15}$

Tabelle 5.28.: Unterschied in den Testergebnissen bei Verwendung schneller Optionen für den Löser *KrawSubst*

`poisson3Db`, ist der Anstieg recht gering, für das System `ecology2` hingegen steigt der Fehler recht stark an. Allerdings werden weiterhin aussagekräftige Einschließungen bestimmt. Weiterer Nachteil der schnellen Optionen ist, dass für manche Systeme überhaupt keine verifizierte Lösung mehr berechnet werden kann, wie in diesem Test für *ABACUS_shell_hd* und *light_in_tissue*.

Bei den Tests mit reellen symmetrisch positiv definiten Matrizen war zu sehen, dass der relative Fehler für den Löser `SparseNorm` im Allgemeinen nahezu ideal war. Für manche schlechter konditionierten Systeme, in den Testläufen waren dies *s3dkq4m2* und *s3dkt3m2*, konnten aber einige Stellen Genauigkeit verloren gehen. Dies sollte theoretisch durch eine Erhöhung der Genauigkeit für die Berechnung von Näherungslösung und Residuum ausgeglichen werden können. Für diese beiden Matrizen werden daher die Testläufe mit acht Threads noch einmal wiederholt, wobei Näherungslösung und Residuum in vierfacher statt dreifacher Länge bestimmt werden. Tabelle 5.29 zeigt die Ergebnisse.

Matrix	Standardoptionen		Residuum in 4-facher Länge	
	Zeit	Relativer Fehler	Zeit	Relativer Fehler
s3dkq4m2	28.6	$7.7 \cdot 10^{-11}$	30.0	$2.4 \cdot 10^{-16}$
s3dkt3m2	25.8	$1.5 \cdot 10^{-12}$	26.7	$2.4 \cdot 10^{-16}$

Tabelle 5.29.: Unterschied in den Testergebnissen bei genauerer Residuumsberechnung für den Löser *SparseNorm*

Der relative Fehler konnte durch diese Erhöhung der Genauigkeit also auf das gleiche Niveau verbessert werden, das auch bei den anderen Testsystemen erreicht wurde. Der hierfür nötige Mehraufwand liegt bei unter 5%, ist also vernachlässigbar.

Diese beiden Ergebnisse zeigen die Flexibilität der dünn besetzten C-XSC Löser, welche durch die vielfältigen Optionen sowohl auf große Genauigkeit als auch auf eine optimierte Laufzeit hin konfiguriert werden können.

Als letzter Test wird nun das bereits in Abschnitt 4.1.4 für den parallelen dicht besetzten Löser verwendete Testsystem aus Problem Nummer 7 des SIAM 100 Digit Challenge [25] verwendet. Es ist also wiederum Element $(1,1)$ der Inversen der Matrix A mit der Diagonalen

$$2, 3, 5, 7, \ldots, 224737$$

sowie $a_{ij} = 1$ für alle i, j mit

$$|i - j| = 1, 2, 4, 8, \ldots, 16384$$

und den restlichen Einträgen gleich 0 zu bestimmen. Das exakte Ergebnis lautet

$$A_{11}^{-1} = 0.725078346268401167\ldots.$$

In Abschnitt 4.1.4 wurde hierzu ein Cluster mit 100 Knoten verwendet. Mit Hilfe der dünn besetzten C-XSC Löser lässt sich dieses Problem nun auf einem einfachen PC lösen. Bei Verwendung von acht Threads benötigt der Löser *SparseNorm* hierzu 74.0 Sekunden und berechnet als Einschließung von A_{11}^{-1} das Intervall

$$[7.2507834626840095E-001, 7.2507834626840129E-001].$$

Er ist somit bei gleicher Genauigkeit und deutlich geringerem Ressourcenverbrauch schneller als der zuvor verwendete parallele Löser.

Der Löser *KrawSubst* kann dieses System ebenfalls lösen, benötigt aber etwa 1674 Sekunden, da die Matrix $[LU - A]$ für dieses System recht dicht besetzt und die Berechnung somit enorm aufwändig ist. Die berechnete Einschließung ist

$$[7.2507834626840106E-001, 7.2507834626840118E-001].$$

Zusammenfassend haben die Tests in diesem Kapitel gezeigt, dass die neuen dünn besetzten C-XSC Löser nicht nur recht flexibel, sondern auch performant sind, sowohl im Vergleich zu Intlab als auch zu einfachen Näherungslösern. In vielen Fällen kann eine verifizierte (und gerade für Punktsysteme oder Intervallsysteme mit schmalen Intervalleinträgen) sehr enge Einschließung des exakten Ergebnisses bestimmt werden, ohne dass sich der Zeitaufwand um wesentlich mehr als eine Größenordnung gegenüber einem einfachen Näherungslöser erhöht.

6. Zusammenfassung und Fazit

In diesem abschließenden Kapitel soll eine kurze Zusammenfassung dieser Arbeit sowie ein Ausblick auf mögliche zukünftige Entwicklungen formuliert werden. Dazu wird in Abschnitt 6.1 zunächst ein Gesamtüberblick über die im Rahmen dieser Arbeit erstellten Softwarewerkzeuge, also zum einen die Erweiterungen der C-XSC Bibliothek sowie vor allem die verschiedenen verifizierenden Löser für lineare Gleichungssysteme, gegeben. In Abschnitt 6.2 folgt eine Übersicht zu möglichen zukünftigen Erweiterungen dieser Werkzeuge. Dabei wird zunächst auf Erweiterungen der C-XSC Bibliothek im Allgemeinen eingegangen, welche in einer zukünftigen Version 3.0 einen größeren Umbruch vollziehen wird. Weiterhin werden mögliche Erweiterungen für und Alternativen zu den verschiedenen Löser besprochen. In Abschnitt 6.3 wird schließlich ein kurzes Gesamtfazit dieser Arbeit gezogen.

6.1. Gesamtüberblick

Im Rahmen dieser Arbeit wurden etliche neue Werkzeuge für die Erstellung verifizierender numerischer Software und die verifizierte Lösung linearer Gleichungssysteme implementiert. Die folgenden Aufzählungen sollen noch einmal einen Kurzüberblick über alle Neuerungen ermöglichen.

Ein erster großer Teil der Arbeit beschäftigte sich mit der Erweiterung von C-XSC um etliche neue Funktionen und Möglichkeiten, welche den Einsatz von C-XSC im Rahmen des High Performance Computing ermöglichen:

- **Skalarprodukte in K-facher Arbeitsgenauigkeit**: Als Alternative zum langsamen langen Akkumulator können Skalarprodukte und Skalarproduktausdrücke nun in (ungefähr) K-facher *double*-Präzision (u.a. auch in einfacher Gleitkommarechnung) berechnet werden. Die Wahl der Genauigkeit ist jederzeit zur Laufzeit möglich. So kann ein Programm dynamisch zwischen Geschwindigkeit und Genauigkeit in den Berechnungen wechseln.

- **Hochoptimierte Berechnung von Produkten von Matrizen und Vektoren mittels BLAS**: Für alle Operatoren, welche explizit oder implizit Skalarprodukte berechnen, können nun hocheffiziente BLAS-Routinen verwendet werden. Vor allem die Berechnung von dicht besetzten Matrix-Matrix-Produkten profitiert enorm von der Verwendung dieser Routinen. Bei Benutzung einer BLAS-Bibliothek mit Multithreading-Unterstützung ist außerdem eine Parallelisierung dieser Operationen ohne zusätzlichen Aufwand möglich. Steht keine BLAS-Bibliothek zur Verfü-

gung, so können die entsprechenden mittels OpenMP parallelisierten Operationen verwendet werden.

- **Datentypen für dünn besetzte Vektoren und Matrizen**: In C-XSC stehen nun vollwertige Datentypen für dünn besetzte Vektoren und Matrizen zur Verfügung, welche nicht nur den gleichen Funktionsumfang wie die bestehenden dicht besetzten Typen bieten, sondern auch ähnlich performant wie die entsprechenden Operationen in Matlab sind. Diese Datentypen ermöglichen erst die effektive Implementierung vieler numerischer Problemstellungen in C-XSC, wie z.B. der dünn besetzten Löser in dieser Arbeit.

- **Vorbereitungen für die Parallelisierung von C-XSC Programmen**: Bisherige C-XSC Versionen hatten durch den starken Gebrauch globaler Variablen große Probleme mit der Threadsicherheit, welche nun behoben sind. Eine Parallelisierung mit OpenMP ist nun sehr viel einfacher möglich. Auch für Cluster können mit der aktualisierten MPI-Schnittstelle nun parallele Programme geschrieben werden, welche alle neuen Möglichkeiten von C-XSC vollständig unterstützen.

Der zweite große Teil dieser Arbeit beschäftigte sich mit der Implementierung laufzeiteffizienter verifizierender Löser für verschiedene Klassen linearer Gleichungssysteme. Insgesamt stehen nun Löser für folgende Problemstellungen für alle Grunddatentypen zur Verfügung:

- **Allgemeine dicht besetzte Gleichungssysteme**: Dicht besetzte Gleichungssysteme beliebiger Struktur können bis zu einer Kondition von etwa 10^{16} durch Unterstützung von BLAS- und LAPACK-Routinen effektiv gelöst werden. Durch die Möglichkeiten von C-XSC zur genauen Skalarproduktberechnung können für Punktsysteme und Intervallsysteme mit kleinem Durchmesser sehr enge Einschließungen berechnet werden. Auch für schlechter konditionierte Gleichungssysteme kann dieser Löser durch Unterstützung der Näherungsinversen doppelter Länge verwendet werden.

- **Sehr große allgemeine dicht besetzte Gleichungssysteme**: Für sehr große dicht besetzte Systeme sind die Ressourcen handelsüblicher PCs normalerweise nicht ausreichend. Mit Hilfe des parallelen Lösers für Distributed-Memory-Systeme lassen sich, bei ausreichender Größe des verwendeten Clusters, auch Systeme mit Dimensionen von 100000×100000 oder größer noch verifiziert lösen.

- **Parametrische dicht besetzte Gleichungssysteme**: Oftmals treten in praktischen Problemstellungen Gleichungssysteme auf, welche eine affin-lineare Abhängigkeit von Parametern aufweisen, die innerhalb bestimmter Intervalle variieren. C-XSC stellt nun für diese Problemstellung zwei verschiedene Löser bereit, welche zum einen allgemeine parametrische Gleichungssysteme und zum anderen Systeme mit einer speziellen Struktur, wie sie bei Verwendung der Finite-Elemente-Methode vorkommen, lösen können.

- **Dünn besetzte Gleichungssysteme**: Für dünn besetzte Gleichungssysteme stehen nun auch C-XSC Löser bereit. Durch die Implementierung verschiedener Lösungsalgorithmen sind für viele Klassen von Systemen effektive Lösungsmöglichkeiten vorhanden. Wie die Tests gezeigt haben, sind diese vielfach deutlich schneller als bekannte Löser dieser Art und darüber hinaus oftmals auch noch wesentlich genauer.

6.2. Erweiterungsmöglichkeiten und Ausblick

Die in dieser Arbeit präsentierten Softwarewerkzeuge haben C-XSC um vielfältige Möglichkeiten erweitert. Dennoch gibt es natürlich in vielerlei Hinsicht noch Verbesserungs- bzw. Erweiterungsmöglichkeiten, von denen einige im Folgenden kurz diskutiert werden.

6.2.1. C-XSC

Die C-XSC Bibliothek wird ständig weiterentwickelt, weshalb auch abseits der in dieser Arbeit beschriebenen Erweiterungen zukünftig viele Änderungen zu erwarten sind. Im Folgenden werden etliche Punkte aufgezählt, die entweder bereits geplant sind oder nach Meinung des Autors sinnvolle Änderungen bzw. Erweiterungen der Bibliothek wären.

- **„C-XSC 3.0“**: Die Version 3.0 von C-XSC ist für die nähere Zukunft geplant und soll einige Schwachstellen mit älteren Teilen der Bibliothek beheben. Der Kern von C-XSC ist das sogenannte *Run-Time-System* (RTS), welches vor allem den Code für den langen Akkumulator enthält. Dieser Teil der Bibliothek ist in C geschrieben und wenig optimiert. An der Universität Würzburg wurden bereits einige Versuche mit einer Implementierung des langen Akkumulators in C++ durchgeführt, welche in etwa doppelt so schnell ist wie die C-XSC Implementierung. Ein großes Problem der aktuellen Version ist, dass Sonderfälle wie `NaN` und `inf` bei Verwendung des Akkumulators in der Regel einen Programmabbruch verursachen. Dies ist auch für die Löser aus dieser Arbeit ein Problem, da ein Überlauf so in der Regel einen Absturz zur Folge hat, der sich in den Lösern nur schwer sinnvoll abfangen lässt. Eine Neuimplementierung des langen Akkumulators in C++ könnte also zum einen performanter sein und zum anderen eine bessere Fehlerbehandlung mit Verwendung sinnvoller Exceptions bieten.

  Allgemein ist ein vereinheitlichtes Exception-Handling eines der wichtigsten Ziele für C-XSC 3.0. Neben dem RTS ist auch die fi_lib, welche von C-XSC für die Berechnung mathematischer Standardfunktionen wie z.B. Sinus und Kosinus verwendet wird, in C geschrieben. Bei Fehlern in diesem Teil der Bibliothek kommt es zwar zu keinem unkontrollierten Absturz, allerdings wird das Programm abrupt mit einer Fehlermeldung beendet. Auch hier wäre eine Umstellung auf C++ mit einem einheitlichen Exception-Handling von besonderem Interesse.

- **Anpassung an neue Standards**: Kürzlich wurde der neue C++ Standard [66], auch als C++11 bezeichnet, verabschiedet. Dieser enthält einige neue Sprachkon-

strukte, die auch für die Verbesserung von C-XSC von Bedeutung sein können. Dazu zählen zum einen die sogenannten *rvalue*-Referenzen und *Move-Konstruktoren*, welche in C-XSC verhindern können, das z.B. bei Berechnungen der Form `A=B+C;` mit Matrixobjekten `A`, `B` und `C` das Ergebnis der Berechnung `B+C` nach `A` kopiert werden muss. Auch enthält C++ etliche Verbesserungen hinsichtlich des Multithreading-Supports. Eine Umstellung von C-XSC auf C++11 erscheint also mittelfristig sinnvoll, sobald der Compiler-Support für C++11 ein ausreichendes Niveau erreicht hat.

Weiterhin wird zur Zeit an einem offiziellen IEEE Standard für Intervallarithmetik gearbeitet [4]. Der aktuelle Entwurf sieht etliche Punkte vor, die in C-XSC momentan nicht implementiert sind, so z.B. die Unterstützung für unbeschränkte Intervalle und die sogenannten Intervalldekorationen (dies sind im Prinzip Flags, welche den Status des Intervalls bzw. der Berechnung, aus welcher das Intervall hervorging, beschreiben). Eine effektive Implementierung des Standards ohne entsprechende Hardware-Unterstützung könnte insbesondere durch die Dekorationen einige Probleme verursachen, da diese etliche zusätzliche Verzweigungen in den Grundoperationen und Standardfunktionen nötig machen und den Speicherbedarf für jedes Intervallobjekt erhöhen. Möglicherweise wäre aber eine zusätzliche Implementierung mit zur Compile-Zeit wählbarer Standardkonformität für C-XSC interessant.

- **Vollständige LAPACK- und SuiteSparse-Schnittstelle**: Für einige der in dieser Arbeit benötigten Funktionen wurden bereits Schnittstellen zu den hochoptimierten Bibliotheken LAPACK und SuiteSparse erstellt. Für den Benutzer von C-XSC wäre es sicherlich vorteilhaft, den kompletten Funktionsumfang dieser Bibliotheken als Schnittstelle zur Verfügung zu haben. So könnten vielfältige Punktprobleme näherungsweise mit diesen sehr effektiven Routinen gelöst werden. Da auch in der Verifikationsnumerik vielfach von Näherungslösungen ausgegangen wird, wäre eine entsprechende Unterstützung sehr sinnvoll.

- **Verwendung von Indizes im `long`-Format**: C-XSC verwendet durchgehend den Datentyp `int` für alle Indizes, welcher in C++ üblicherweise eine Länge von 4 Byte aufweist. In vielen Fällen wäre die optionale Unterstützung von 8 Byte langen Integern (üblicherweise der Datentyp `long` oder `int64`), gerade auf 64-bit-Systemen und bei der Arbeit mit dünn besetzten Matrizen, sinnvoll. Dies könnte relativ einfach über ein Präprozessor-Makro implementiert werden, bedarf aber größerer Änderungen an weiten Teilen der Bibliothek.

- **Optimierung dünn besetzter Teilmatrizen**: Aktuell wird beim Ausschneiden von Teilmatrizen aus dünn besetzten Matrizen stets eine Kopie der Teilmatrix gespeichert, um möglichst einfach performante Berechnungen mit diesem Ausschnitt durchführen zu können. Gerade bei Verwendung von Teilmatrizen in mehreren Threads kann dies zu einem stark erhöhten Speicherbedarf führen. Hier wäre also eine, evtl. optionale, alternative Art der Berechnung, welche nur mit den Originaldaten arbeitet, sinnvoll.

Methode	Zeit in Sekunden	Relativer Fehler	Zeitfaktor zu BLAS
BLAS	0.049	$3.0 \cdot 10^{35}$	1.0
DotK, $K = 2$	2.06	$4.2 \cdot 10^{20}$	42.0
DotK, $K = 3$	4.03	$1.5 \cdot 10^{5}$	82.0
Akkumulator	13.9	0	287.5
FastMatMul	2.44	0	45.7

Tabelle 6.1.: Testergebnisse für alternatives exaktes Matrixprodukt

- **Alternatives hochgenaues Matrixprodukt**: Kürzlich wurde von Ozaki et al. [110] ein alternativer Algorithmus für die hochgenaue Berechnung von Matrix-Matrix-Produkten mit Hilfe von einfachen Gleitkomma-Matrixprodukten veröffentlicht. Dieser basiert auf einer (auch bei Auftreten von Unterlauf) fehlerfreien Transformation des Produktes in eine Summe von Gleitkomma-Matrizen. Da dabei ausschließlich einfache Gleitkommaprodukte verwendet werden, können diese mittels hochoptimierter BLAS-Routinen ausgeführt werden. Die Berechnung der Summe nach der Transformation kann in C-XSC mittels des Akkumulators exakt erfolgen, so dass so auch eine exakte Berechnung des Gesamtproduktes möglich ist. Eine erste Implementierung zur Berechnung des Produkts reeller Punktmatrizen zeigt bereits vielversprechende Ergebnisse (Tabelle 6.1). Alle Tests wurden dabei mit 8 Threads auf einem 8-Kern-Testrechner und mit der Intel MKL 10.3 als BLAS-Bibliothek durchgeführt. Es wurde ein Matrixprodukt mit der Kondition 10^{50} und der Dimension $n = 1000$ berechnet. Zur Bestimmung des relativen Fehlers wurde das maximal genaue Ergebnis mit Hilfe des Akkumulators berechnet.

  Wie die Ergebnisse zeigen, erscheint eine Verwendung in C-XSC, vor allem als Alternative zur Berechnung maximal genauer Produkte mit dem Akkumulator, reizvoll. Zu beachten ist aber stets, dass der Speicheraufwand deutlich höher ist als bei den anderen Berechnungsmethoden, da die Summe von Matrizen nach der fehlerfreien Transformation sehr lang sein kann. In obigem Testfall müssen für die Berechnung 33 Matrizen der Dimension 1000×1000 gespeichert werden. In den hier vorgenommenen Testläufen handelt es sich dabei bei nahezu allen Matrizen um dicht besetzte Matrizen. Dies kann also schon für mittelgroße Matrixprodukte zu Speicherproblemen führen. Allerdings ist zu beachten, dass Laufzeit und Zahl der zu speichernden Matrizen direkt von der Kondition abhängen. Für ein Produkt zweier Zufallsmatrizen der Dimension 1000 benötigt die Implementierung des neuen Algorithmus z.B. nur 1.19 Sekunden und es müssen nur noch 16 Matrizen gespeichert werden.

- **Unterstützung von GPUs**: In den letzten Jahren hat die Rechenleistung von Grafikprozessoren (GPUs) enorm zugenommen. Durch die starke Parallelität dieser Prozessoren können sie bei geeigneten Problemstellungen um ein Vielfaches schnel-

ler als übliche CPUs sein. In jüngster Zeit sind solche GPUs auch für den Bereich der Verifikationsnumerik interessant geworden, da sie nun auch Berechnungen im IEEE-*double*-Format in guter Geschwindigkeit ermöglichen und darüber hinaus Operationen in verschiedenen Rundungsmodi anbieten. Der Rundungsmodus ist bei GPUs direkt an die einzelne Operation geknüpft und wird nicht mehr global gesetzt, was bei der Intervallrechnung große Laufzeitvorteile bieten kann. Eine entsprechende Implementierung der C-XSC Intervallarithmetik für solche GPUs könnte also für viele Problemstellungen sehr interessant sein.

6.2.2. Verifizierende Löser für lineare Gleichungssysteme

Die in den Kapiteln 4 und 5 vorgestellten verifizierenden Löser für lineare Gleichungssysteme sind effiziente Implementierungen von für den allgemeinen Fall in der Regel gut geeigneten Algorithmen. Dennoch sind in einigen Bereichen sinnvolle Erweiterungen und Optimierungen denkbar, von denen einige im Folgenden, getrennt nach Lösern für dicht und dünn besetzte Gleichungssysteme, aufgezählt werden.

Dicht besetzte Gleichungssysteme

- **Alternative Algorithmen zur schnellen verifizierten Lösung dicht besetzter Gleichungssysteme**: Zur Beschleunigung der Verifikation könnten Abwandlungen des in Kapitel 4 genutzten Algorithmus implementiert werden, welche die Berechnung $\boldsymbol{C} = [I - RA]$ nicht direkt ausführen, sondern geeignete Abschätzungen verwenden, welche (für Punktsysteme) ein Matrix-Matrix-Produkt einsparen. Die Herleitung entsprechender Abschätzungen ist allerdings problematisch. Im Allgemeinen würde hierdurch auch der relative Fehler der Einschließung ansteigen. Außerdem wird durch die gröbere Einschließung von $\boldsymbol{C}$ evtl. eine erfolgreiche Verifizierung verhindert.

  Oishi et al. [107] beschreiben einen alternativen schnellen Ansatz, welcher auf einer normweisen Abschätzung des Fehlers einer Näherungslösung basiert. In der schnellsten Variante ist der hierfür zusätzlich nötige Aufwand im Wesentlichen auf die Invertierung der Matrizen L und U aus eine LU-Zerlegung der Systemmatrix beschränkt. Mit diesen Näherungsinversen kann dann über Fehlerabschätzungen eine verifizierte Einschließung berechnet werden. Der Gesamtaufwand steigt dabei gegenüber einem einfachen Gleitkommalöser auf Basis einer LU-Zerlegung nur um den Faktor zwei. Auch hier ist aber, vor allem bei Intervallsystemen, mit einem vergleichsweise großen relativen Fehler zu rechnen.

  Es existieren alternative Methoden, welche ausnutzen, dass das Produkt RA mit einer ausreichend guten Näherungsinversen R von A eine H-Matrix ist (siehe z.B. Hong Diep et al. [39]). Diese weisen in der Regel einen ähnlichen Aufwand wie die in dieser Arbeit verwendete Methode auf Basis des Krawczyk-Operators auf. Eine Implementierung mit C-XSC und direkte Vergleiche solcher alternativer Methoden

mit den Implementierungen aus dieser Arbeit wären in Zukunft aber sicherlich interessant.

- **MPI-Version des allgemeinen parametrischen Lösers**: Der Löser für allgemeine parametrische Systeme weist einen hohen Speicher- und, insbesondere bei Verwendung der genauen Iterationsmatrix, sehr hohen Rechenzeitbedarf auf. Prinzipiell könnte daher eine Parallelisierung mittels MPI, analog zum parallelen Löser für allgemeine dicht besetzte Systeme, interessant sein. Zu bedenken ist allerdings, dass die Methode, insbesondere bei breiten Parameterintervallen, für größere Dimensionen, bei denen eine solche Parallelisierung vor allem von Interesse wäre, in der Regel nicht zum Erfolg führt.

- **Verwendung von Langzahl-Datentypen**: C-XSC stellt Langzahl-Datentypen mit enorm breitem Exponentenbereich zur Verfügung [21]. Eine Nutzung dieser Datentypen in einem angepassten Löser würde es ermöglichen, auch enorm schlecht konditionierte Systeme zu lösen. Weiterhin könnte schon in den Eingangsdaten der sehr große Exponentenbereich dieser Datentypen ausgenutzt werden. Die Laufzeit wäre aber sicherlich um mehrere Größenordnungen langsamer als bei den in dieser Arbeit beschriebenen Lösern.

Dünn besetzte Gleichungssysteme

- **Optimierung des Speicherbedarfs**: Eine wesentliche Beschränkung der mit Lösern für dünn besetzte Systeme lösbaren Systeme ist der Speicherbedarf. Die aktuelle Implementierung geht davon aus, dass, wie bei allen C-XSC Toolbox-Modulen und -Erweiterungen üblich, die übergebene Systemmatrix im Verlauf des Lösers nicht verändert wird. Daher muss z.B. bei Skalierung der Matrix mit einer Kopie weitergearbeitet werden. Würde auf diese Beschränkung verzichtet, so könnte die Originalmatrix im Verlauf des Lösers überschrieben und so an einigen Stellen Speicherplatz gespart werden.

- **MPI-Version der dünn besetzten Löser**: Auch für die dünn besetzten Löser wäre eine Parallelisierung mit MPI denkbar. Allerdings wäre dafür zum einen die weiter oben angesprochene Umstellung der C-XSC Indizierung auf den Datentyp `long` eine sinnvolle Voraussetzung. Zum anderen sind die direkten Methoden, auf welchen die dünn besetzten Löser basieren, für sehr große dünn besetzte Matrizen oftmals nicht mehr sinnvoll. Prinzipiell ist aber durch das MPI-Interface für C-XSC die Grundlage für eine solche Parallelisierung vorhanden.

- **Anpassung des dünn besetzten Krawczyk-Lösers für parametrische Systeme**: Analog zum allgemeinen parametrischen Löser aus Kapitel 4 wäre eine Anpassung des dünn besetzten Lösers auf Basis des Krawczyk-Operators für parametrische Systeme denkbar. Da der dünn besetzte Krawczyk-Löser aber bei vergleichsweise wenigen Systemen zum Erfolg führt, ist fraglich, ob der Aufwand für eine solche Anpassung lohnenswert wäre.

- **Verwendung von Langzahl-Datentypen**: Wie im dicht besetzten Fall auch könnte eine Anpassung der dünn besetzten Löser für die Langzahl-Datentypen die Klasse der lösbaren Systeme drastisch erweitern. Für dünn besetzte Systeme wäre dazu aber zunächst die Erstellung entsprechender dünn besetzter Matrixdatentypen für die Langzahl-Datentypen erforderlich.

- **Spezialisierung des normweisen Lösers für allgemeine symmetrische Systeme**: Rump [126] beschreibt eine Modifizierung des in 5.1.2 beschriebenen Vorgehens für allgemeine symmetrische Matrizen, bei der Diagonalmatrizen D_1 und D_2 bestimmt werden, so dass für die LDL^T-Zerlegung der Systemmatrix

  $$LDL^T = (LD_1) \cdot (LD_2)^T$$

  gilt mit $D_1 D_2 = D$ und $|D_1| = |D_2| = |D|^{\frac{1}{2}}$. LD_1 und LD_2 haben dann die gleichen Singulärwerte und die Kondition der Matrix wird auf die beiden Faktoren aufgeteilt. Über die Bestimmung einer unteren Schranke von $\sigma_n(LD_1)$ (einfacher möglich als im allgemeinen Fall, da die Kondition von LD_1 in etwa $\sqrt{\text{cond}(A)}$ entspricht) kann somit eine Unterschranke für $\sigma_n(LDL^T)$ berechnet werden. Eine zusätzliche Implementierung dieses Vorgehens könnte also für allgemeine symmetrische Matrizen bessere Ergebnisse liefern.

Allgemein besteht bei der verifizierten Lösung dünn besetzter linearer Gleichungssysteme weiterhin das Problem, dass für allgemeine, nicht positiv definite Systeme, kein Algorithmus bekannt ist, der auch bei höheren, aber im Allgemeinen noch hinreichend kleinen Konditionszahlen (bis etwa 10^{16} für *double*-Präzision) stets zum Erfolg führt und dabei nicht um deutlich mehr als eine Größenordnung langsamer ist als einfache Gleitkommalöser. Zwar führen die in dieser Arbeit implementierten Löser für einige solcher Systeme zu einer erfolgreichen Verifizierung, allerdings zeigten die Tests auch, dass für viele Systeme keine verifizierte Lösung bestimmt werden kann. Evtl. kann eine Anpassung bestehender iterativer Verfahren in Zukunft zum Erfolg führen, insgesamt bleibt dies aber ein wichtiges ungelöstes Problem der Verifikationsnumerik.

6.3. Fazit

Diese Arbeit hatte zwei wesentliche Kernthemen: Zum einen die Erweiterung und Modernisierung der C-XSC Bibliothek zur Erhöhung der Flexibilität und zur Vorbereitung auf den Einsatz im High Performance Computing. Zum anderen die Implementierung von modernen, effizienten Lösern zur verifizierten Lösung linearer Gleichungssysteme, welche in erster Linie auf eher allgemeine Problemstellungen ausgerichtet und somit für viele Anwendungsbereiche interessant sind. Wie die ausführlichen Tests im Rahmen dieser Arbeit gezeigt haben, wurden in beiden Bereichen eine Reihe nützlicher neuer Werkzeuge erstellt, deren Anwendung auch über diese Arbeit hinaus von Interesse sein kann.

C-XSC hat durch die neuen Skalarproduktalgorithmen und die dünn besetzten Datentypen wesentlich an Flexibilität gewonnen, und liegt außerdem nun, u.a. durch die BLAS-Unterstützung, bei der Laufzeit von Matrix- und Vektoroperationen in etwa gleichauf mit Intlab. Die Berechnung von Skalarprodukten in K-facher Arbeitsgenauigkeit, sowohl für dicht also auch dünn besetzte Matrizen und Vektoren, wird in dieser Form (nach Wissen des Autors) von keiner vergleichbaren Bibliothek angeboten. Weiterhin sind die erzielten Laufzeiten denen anderer bekannter Bibliotheken, welche mit höherer Genauigkeit rechnen, wie z.B. XBLAS, überlegen. Durch die OpenMP-Parallelisierung werden zudem auch Mehrkernsysteme bereits ausgenutzt.

Der allgemeine Löser für dicht besetzte Systeme liegt auf einem ähnlichen Geschwindigkeitsniveau wie der wohl meist genutzte Löser dieser Art aus Intlab, ist dabei aber deutlich genauer und bietet mehr Konfigurationsmöglichkeiten an, wie z.B. die Möglichkeit, Inneneinschließungen zu berechnen. Die mittels MPI parallelisierte Version dieses Lösers ist die schnellste bekannte Implementierung für die verifizierte Lösung dicht besetzter Matrizen auf Clustern. Weiterhin ist sie die einzige Implementierung, welche auch komplexe Systeme unterstützt.

Der Löser für parametrische Systeme ist ebenfalls die schnellste bekannte Implementierung des verwendeten Algorithmus. Auch hier ist durch Nutzung der neuen C-XSC Möglichkeiten die Qualität der Ergebnisse entweder besser oder auf dem gleichen Niveau wie bei vergleichbaren Lösern. Durch den Löser auf Basis des alternativen Algorithmus nach Neumaier und Pownuk, welcher im Rahmen dieser Arbeit erstmalig in einer verifizierenden Version implementiert wird, steht darüber hinaus für Systeme mit passender Struktur eine noch schnellere und genauere Alternative zur Verfügung. Beide parametrischen Löser können dabei durch die verschiedenen Maßnahmen zur Parallelisierung auch moderne Mehrkernsysteme gut ausnutzen.

Auch die dünn besetzten Löser zählen zu den schnellsten dem Autor bekannten Implementierungen. Der Löser auf Basis des Krawczyk-Operators kann bei geeigneten Systemen komponentenweise gute Einschließungen berechnen. Erstmals wird hier auch eine optimierte Implementierung der Parallelepiped-Methode nach Lohner für alle Grunddatentypen und den vielfältigen bereitgestellten Konfigurationsoptionen (z.B. der schnellen Verifikation) implementiert. Der Löser für dünn besetzte Systeme auf Basis von Rumps auch in Intlab verwendetem Algorithmus zur normweisen Defekt-Abschätzung ist nicht nur deutlich schneller und effizienter parallelisiert als die verbreitet genutzte Implementierung in Intlab, sondern kann durch die Möglichkeiten von C-XSC zur hochgenauen Berechnung von Näherungslösung und Residuum auch deutlich bessere Ergebnisse erzielen. Die Laufzeit bleibt dabei für viele Systeme innerhalb einer Größenordnung verglichen mit modernen Näherungslösern.

Insgesamt zählen also alle im Rahmen dieser Arbeit erstellten Softwarewerkzeuge nicht nur zu den schnellsten, sondern auch zu den genauesten ihrer Art. Es ist die Hoffnung des Autors, dass die C-XSC Bibliothek durch die angesprochenen zukünftigen Änderungen und die Nutzung der in dieser Arbeit präsentierten Softwarewerkzeuge auch zukünftig eine wichtige Rolle auf dem Gebiet der Verifikationsnumerik spielen kann.

A. Beispiele für Dateien im Matrix-Market-Format

Im Folgenden werden einige Beispiele für Dateien im Matrix-Market-Format angegeben. Dabei wird auch auf die Erweiterungen zur Unterstützung von Intervallmatrizen in C-XSC eingegangen. Die folgenden Beispiele sind nach Datentypen unterteilt, es werden aber zwischendurch zusätzliche Beispiele zur Ausgabe im dicht besetzten, symmetrischen oder hexadezimalen Fall angegeben.

Reelle Punktmatrix

Die Matrix

$$A = \begin{pmatrix} 1.0 & 2.0 & 0 & 0 & 0 \\ 0 & 2.5 & 0 & 0 & 0 \\ 0 & 0 & 5.2 & 0 & 1.1 \\ 0 & 0 & 0 & 0 & 0.3 \\ 0 & 2.0 & 0 & 0 & 0 \end{pmatrix}$$

kann im Matrix-Market-Format als dünn besetzte Matrix wie folgt gespeichert werden:

```
%%MatrixMarket matrix coordinate real general
5 5 7
1 1 1.0
1 2 2.0
2 2 2.5
3 3 5.2
3 5 1.1
4 5 0.3
5 2 2.0
```

Die zweite Zeile gibt dabei die Zahl der Zeilen, die Zahl der Spalten und die Zahl der Einträge an. Wird sie als dicht besetzte Matrix gespeichert, so hat sie folgendes Format:

```
%%MatrixMarket matrix array real general
1.0
2.0
0
0
0
```

```
0
2.5
0
0
0
0
0
5.2
0
1.1
0
0
0
0
0.3
0
2.0
0
0
0
```

Reelle Intervallmatrix

Die Matrix

$$A = \begin{pmatrix} [1,2] & [2,4] & 0 & 0 & 0 \\ [2,4] & [1,1] & 0 & 0 & 0 \\ 0 & 0 & [1,1] & 0 & [-1,1] \\ 0 & 0 & 0 & [1,1] & 0 \\ 0 & 0 & [-1,1] & 0 & [2,2] \end{pmatrix}$$

kann im Matrix-Market-Format als dünn besetzte Matrix wie folgt gespeichert werden:

```
%%MatrixMarket matrix coordinate interval symmetric
5 5 7
1 1 1 2
2 1 2 4
2 2 1 1
3 3 1 1
4 4 1 1
5 3 -1 1
5 5 2 2
```

Infimum und Supremum werden also getrennt angegeben. Da die Matrix symmetrisch gespeichert wird, muss nur das untere Dreieck angegeben werden. C-XSC kann zwar

symmetrische Matrizen einlesen, speichert sie aber nicht als symmetrisch. Einlesen und ausgeben obiger Datei würde also folgende Matrix-Market-Datei ergeben:

```
%%MatrixMarket matrix coordinate interval general
%Generated by C-XSC
5 5 9
1 1 1 2
2 1 2 4
1 2 2 4
2 2 1 1
3 3 1 1
3 5 -1 1
4 4 1 1
5 3 -1 1
5 5 2 2
```

Komplexe Punktmatrix

Die Matrix

$$A = \begin{pmatrix} 1.0+3i & 2.0 & 0 & 0 & 0 \\ 0 & 2.5+i & 0 & 0 & 0 \\ 0 & 0 & 5.2 & 0 & -2i \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 2.0+5i & 0 & 0 & 0 \end{pmatrix}$$

kann im Matrix-Market-Format als dünn besetzte Matrix wie folgt gespeichert werden:

```
%%MatrixMarket matrix coordinate complex general
5 5 7
1 1 1.0 3
1 2 2.0 0
2 2 2.5 1
3 3 5.2 0
3 5 0 -2
4 5 0.5 0
5 2 2.0 5
```

Real- und Imaginärteil werden also getrennt angegeben. Um Konvertierungsfehler beim Einlesen und Schreiben von binär gespeicherten Daten in dieses Format zu vermeiden, kann C-XSC auf hexadezimale Ein- und Ausgabe umgestellt werden. Obige Matrix sähe dann folgendermaßen aus:

```
%%MatrixMarket matrix coordinate complex general
%Generated by C-XSC
5 5 7
```

```
1 1 +10000000000000e3FF +18000000000000e400
1 2 +10000000000000e400 +10000000000000e000
2 2 +14000000000000e400 +10000000000000e3FF
5 2 +10000000000000e400 +14000000000000e401
3 3 +14CCCCCCCCCCDe401 +10000000000000e000
3 5 +10000000000000e000 -10000000000000e400
4 5 +10000000000000e3FE +10000000000000e000
```

Das hexadezimale Format wird nicht explizit in der Datei vermerkt. Der Benutzer muss also selbstständig das entsprechende Ein-/Ausgabe-Flag setzen.

Komplexe Intervallmatrix

Die Matrix

$$A = \begin{pmatrix} [1,2] & [2,4]+i[1,3] & 0 & 0 & 0 \\ [2,4]-i[1,3] & [1,1] & 0 & 0 & 0 \\ 0 & 0 & [1,1] & 0 & [-1,1]+i[-1,1] \\ 0 & 0 & 0 & [1,1] & 0 \\ 0 & 0 & [-1,1]-i[-1,1] & 0 & [2,2]+i[0,1] \end{pmatrix}$$

kann im Matrix-Market-Format als dünn besetzte Matrix wie folgt gespeichert werden:

```
%%MatrixMarket matrix coordinate cinterval hermitian
5 5 7
1 1 1 2 0 0
2 1 2 4 -3 -1
2 2 1 1 0 0
3 3 1 1 0 0
4 4 1 1 0 0
5 3 -1 1 -1 1
5 5 2 2 0 1
```

Real- und Imaginärteil werden jeweils getrennt mit Infimum und Supremum angegeben. Da die Matrix hermitesch gespeichert wird, muss nur das untere Dreieck angegeben werden. C-XSC speichert wiederum keine Symmetrien und würde die Matrix daher wie folgt ausgeben:

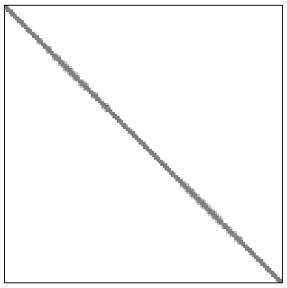
```
%%MatrixMarket matrix coordinate cinterval general
%Generated by C-XSC
5 5 9
1 1 1 2 0 0
1 2 2 4 1 3
2 1 2 4 -3 -1
2 2 1 1 0 0
```

```
3 3 1 1 0 0
3 5 -1 1 -1 1
4 4 1 1 0 0
5 3 -1 1 -1 1
5 5 2 2 0 1
```

B. Dünn besetzte Testmatrizen

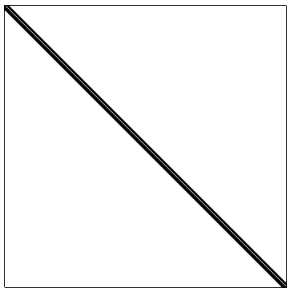
Im Folgenden werden die in dieser Arbeit verwendeten dünn besetzten Testmatrizen aus praktischen Anwendungen aufgeführt. Die Matrizen stammen entweder aus dem Matrix-Market [6] oder aus der Sammlung der Universität von Florida (UF Collection) auf der SuiteSparse Webseite [7]. Die folgende Aufzählung sortiert die Matrizen alphabetisch nach ihrer Bezeichnung, welche auch in den verschiedenen Testkapiteln dieser Arbeit verwendet wird. Angegeben werden neben Quelle, Dimension (alle Matrizen sind quadratisch) und Zahl der Nichtnullen (nnz) jeweils der Datentyp (reell oder komplex), Symmetrie, positive Definitheit sowie eine Schätzung der Konditionszahl (entweder aus der Angabe in der Quelle oder berechnet mit der Funktion `condest` von Matlab). Weiterhin wird jeweils die Besetztheitsstruktur der Matrix mit einer entsprechenden Grafik veranschaulicht.

ABACUS_shell_hd



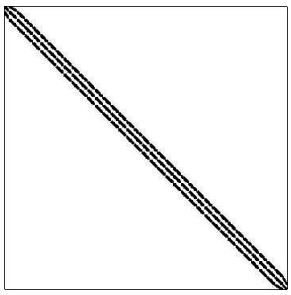
Dimension:	23412×23412
nnz:	218484
Datentyp:	Komplex
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$2.0 \cdot 10^{16}$
Quelle:	UF Collection

af23560



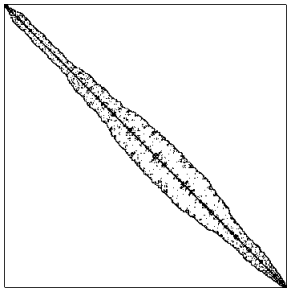
Dimension:	23560×23560
nnz:	484256
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$3.5 \cdot 10^5$
Quelle:	Matrix Market

bcsstk10



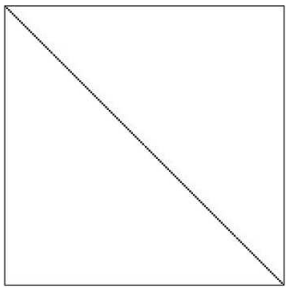
Dimension:	1086×1086
nnz:	11578
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	$1.3 \cdot 10^6$
Quelle:	Matrix Market

bcsstk18



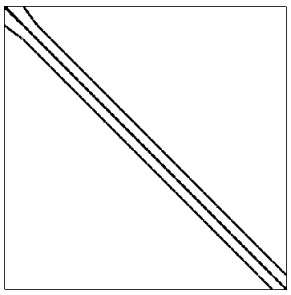
Dimension:	11948×11948
nnz:	80519
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	65
Quelle:	Matrix Market

bcsstm05



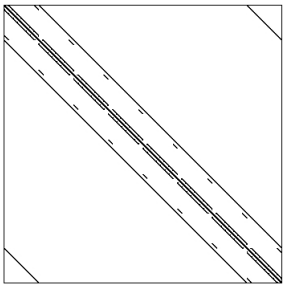
Dimension:	153×153
nnz:	153
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Nein (positiv semi-definit)
Kondition (geschätzt):	$1.3 \cdot 10^6$
Quelle:	Matrix Market

cavity26



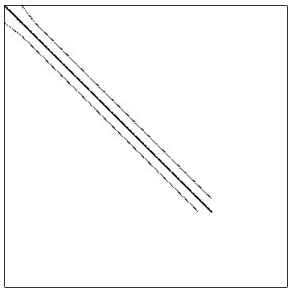
Dimension:	4562×4562
nnz:	138187
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.2 \cdot 10^8$
Quelle:	Matrix Market

conf5.4-00l8x8-0500



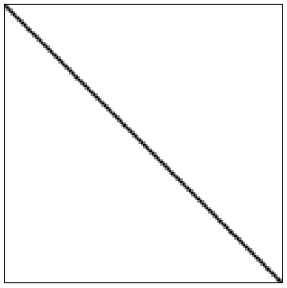
Dimension:	49152×49152
nnz:	1916928
Datentyp:	Komplex
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$3.2 \cdot 10^5$
Quelle:	Matrix Market

dwg961a



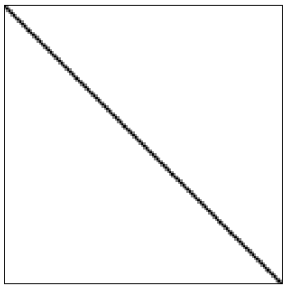
Dimension:	961×961
nnz:	3405
Datentyp:	Komplex
Symmetrisch:	Ja
Positiv definit:	Nein
Kondition (geschätzt):	∞
Quelle:	Matrix Market

ecology1



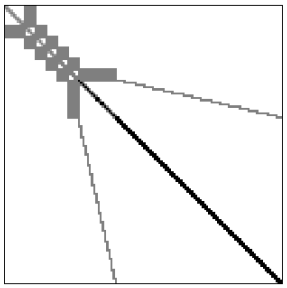
Dimension:	1000000×1000000
nnz:	4996000
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Nein
Kondition (geschätzt):	$7.6 \cdot 10^{17}$
Quelle:	UF Collection

ecology2



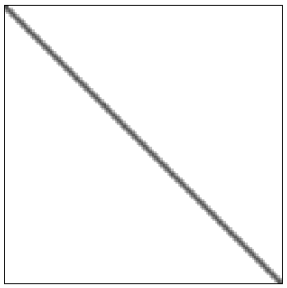
Dimension:	999999×999999
nnz:	4995991
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	$6.7 \cdot 10^7$
Quelle:	UF Collection

G3_circuit



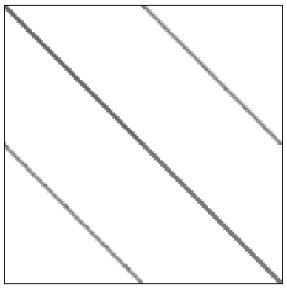
Dimension:	1585478×1585478
nnz:	7660826
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	$2.2 \cdot 10^7$
Quelle:	UF Collection

kim1



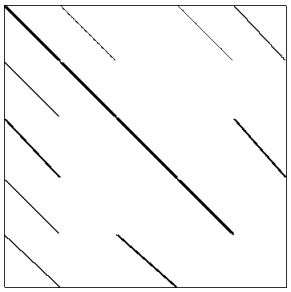
Dimension:	38415×38415
nnz:	933195
Datentyp:	Komplex
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.9 \cdot 10^4$
Quelle:	UF Collection

light_in_tissue



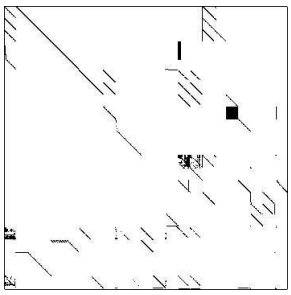
Dimension:	29282×29282
nnz:	406084
Datentyp:	Komplex
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.1 \cdot 10^4$
Quelle:	UF Collection

lns_3937



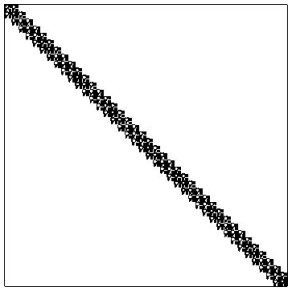
Dimension:	3937×3937
nnz:	25407
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.0 \cdot 10^{17}$
Quelle:	Matrix Market

mahindas



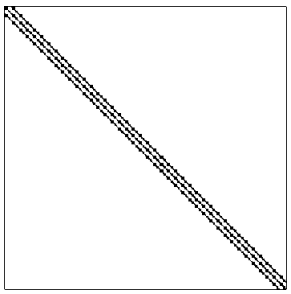
Dimension:	1258×1258
nnz:	7682
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.03 \cdot 10^{13}$
Quelle:	Matrix Market

mhd1280a



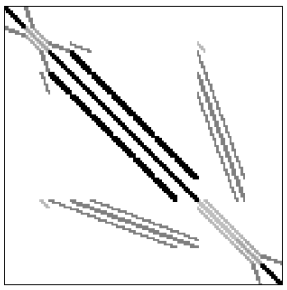
Dimension:	1280×1280
nnz:	47906
Datentyp:	Komplex
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$9.88 \cdot 10^{24}$
Quelle:	Matrix Market

mhd1280b



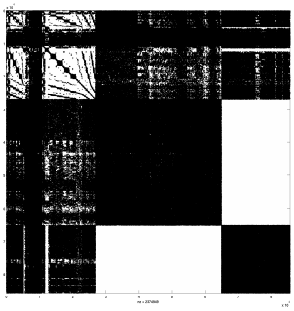
Dimension:	1280×1280
nnz:	22778
Datentyp:	Komplex
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	$6.0 \cdot 10^{12}$
Quelle:	Matrix Market

mplate



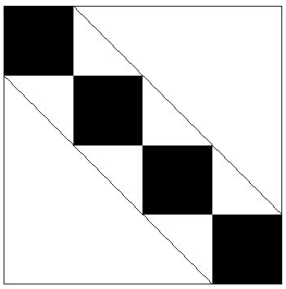
Dimension:	5962×5962
nnz:	142190
Datentyp:	Komplex
Symmetrisch:	Ja
Positiv definit:	Nein
Kondition (geschätzt):	$3.4 \cdot 10^{16}$
Quelle:	UF Collection

poisson3Db



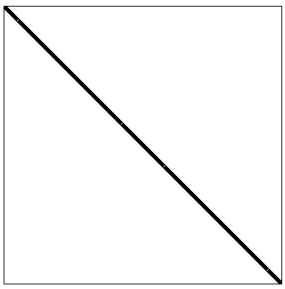
Dimension:	85623×85623
nnz:	2374949
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.7 \cdot 10^{5}$
Quelle:	UF Collection

qc324



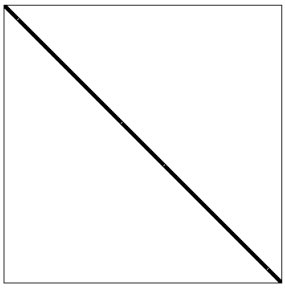
Dimension:	324×324
nnz:	26730
Datentyp:	Komplex
Symmetrisch:	Ja
Positiv definit:	Nein
Kondition (geschätzt):	$7.4 \cdot 10^4$
Quelle:	Matrix Market

s3dkq4m2



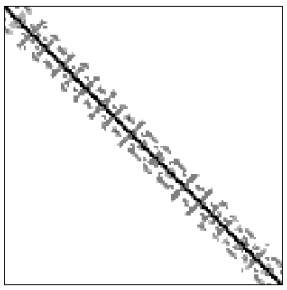
Dimension:	90449×90449
nnz:	2455670
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	$3.5 \cdot 10^{11}$
Quelle:	Matrix Market

s3dkt3m2



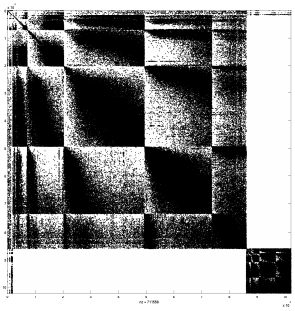
Dimension:	90449×90449
nnz:	1921955
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	$6.3 \cdot 10^{11}$
Quelle:	Matrix Market

stomach



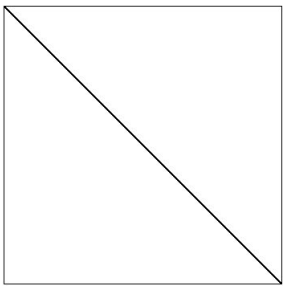
Dimension:	213360×213360
nnz:	3021648
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	80
Quelle:	UF Collection

thermomech_TC



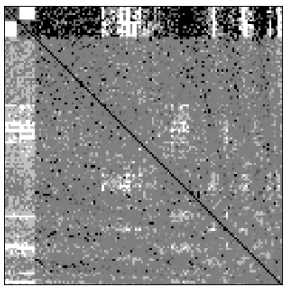
Dimension:	102158×102158
nnz:	711558
Datentyp:	Reell
Symmetrisch:	Ja
Positiv definit:	Ja
Kondition (geschätzt):	123
Quelle:	Matrix Market

tub1000



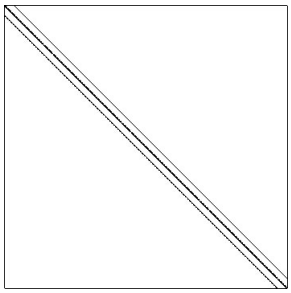
Dimension:	1000×1000
nnz:	3996
Datentyp:	Reell
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$2.3 \cdot 10^6$
Quelle:	Matrix Market

waveguide3D



Dimension:	21036×21036
nnz:	303468
Datentyp:	Komplex
Symmetrisch:	Nein
Positiv definit:	Nein
Kondition (geschätzt):	$1.0 \cdot 10^4$
Quelle:	UF Collection

young1c



Dimension:	841×841
nnz:	4089
Datentyp:	Komplex
Symmetrisch:	Ja
Positiv definit:	Nein
Kondition (geschätzt):	$2.9 \cdot 10^2$
Quelle:	Matrix Market

Literaturverzeichnis

[1] *BLAS Webseite*. http://www.netlib.org/blas/.

[2] *C-XSC Webseite*. http://www.xsc.de/.

[3] *CXSparse Webseite*. http://www.cise.ufl.edu/research/sparse/CXSparse/.

[4] *IEEE1788 Working Group Webseite*. http://grouper.ieee.org/groups/1788/.

[5] *LAPACK Webseite*. http://www.netlib.org/lapack/.

[6] *Matrix Market Webseite*. http://math.nist.gov/MatrixMarket/.

[7] *SuiteSparse Webseite*. http://www.cise.ufl.edu/research/sparse/SuiteSparse/.

[8] *Top 500 Supercomputer Webseite*. http://www.top500.org.

[9] Alefeld, G. und J. Herzberger: *Einführung in die Intervallrechnung*. Nr. 12 in *Reihe Informatik*. Bibliographisches Institut, 1974.

[10] Alefeld, G. und G. Mayer: *On the Symmetric and Unsymmetric Solution Set of Interval Systems*. SIAM J. Matrix Anal. Appl., 16(4):1223–1240, 1995, ISSN 0895-4798.

[11] Amdahl, G. M.: *Validity of the single processor approach to achieving large scale computing capabilities*. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), S. 483–485, New York, NY, USA, 1967. ACM.

[12] Amestoy, P., T. A. Davis und I. S. Duff: *Algorithm 837: AMD, An approximate minimum degree ordering algorithm*. ACM Transactions on Mathematical Software, 30(3):381–388, 2004.

[13] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney und D. Sorensen: *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 3. Aufl., 1999.

[14] ANSI/IEEE: *IEEE Standard for Binary Floating Point Arithmetic*, Bd. Std 754. IEEE, 1985.

[15] Barth, W. und E. Nuding: *Optimale Lösung von Intervallgleichungssystemen.* Computing, 12(2):117–125, 1974, ISSN 0010-485X.

[16] Beeck, H.: *Charakterisierung der Lösungsmenge von Intervallgleichungssystemen.* ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik, 53(12):T181–T182, 1973, ISSN 1521-4001. http://dx.doi.org/10.1002/zamm.19730531292.

[17] Behnke, H.: *Die Bestimmung von Eigenwertschranken mit Hilfe von Variationsmethoden und Intervallarithmetik.* Dissertation, TU Clausthal, 1989.

[18] Berner, S.: *Ein paralleles Verfahren zur verifizierten globalen Optimierung.* Dissertation, Bergische Universität Wuppertal, 1995.

[19] Blackford, L. S., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker und R. C. Whaley: *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997, ISBN 0-89871-397-8.

[20] Blackford, L. S., J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington und R. C. Whaley: *An Updated Set of Basic Linear Algebra Subprograms (BLAS).* ACM Trans. Math. Soft, 28(2):135–151, 2002.

[21] Blomquist, F., W. Hofschuster und W. Krämer: *C-XSC-Langzahlarithmetiken für reelle und komplexe Intervalle basierend auf den Bibliotheken MPFR und MPFI.* BUW-WRSWT Preprints, 2011(1), 2011.

[22] Bohlender, G.: *Computer arithmetic and self-validating numerical methods.* Academic Press, San Diego, 1990.

[23] Bohlender, G., W. Walter, P. Kornerup und D. W. Matula: *Semantics for exact floating point operations.* In: *IEEE Symposium on Computer Arithmetic'91*, S. 22–26, 1991.

[24] Boisvert, R. F., R. Pozo und K. A. Remington: *The Matrix Market Exchange Formats: Initial Design.* http://math.nist.gov/MatrixMarket/reports/MMformat.ps.

[25] Bornemann, F., D. Laurie, S. Wagon und J. Waldvogel: *The SIAM 100-Digit Challenge.* SIAM, Philadelphia, 2004.

[26] Bradley, A. M.: *Algorithms for the Equilibration of Matrices and their Application to Limited-Memory Quasi-Newton Methods.* Dissertation, Stanford University, 2010.

[27] Braune, K. und W. Krämer: *High Accuracy Standard Functions for Real and Complex Intervals*. In: Kaucher, E., U. Kulisch und C. Ullrich (Hrsg.): *Computerarithmetic: Scientific Computation and Programming Languages*, S. 81–114. Teubner, Stuttgart, 1987.

[28] Chapman, B., G. Jost und R. V. D. Pas: *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[29] Chen, Y., T. A. Davis, W. W. Hager und S. Rajamanickam: *Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate*. ACM Transactions on Mathematical Software, 35(3), 2009.

[30] Cordes, D.: *Spärlich besetzte Matrizen*. In: Kulisch, U. (Hrsg.): *Wissenschaftliches Rechnen mit Ergebisverifikation - Eine Einführung*. Akademie Verlag, Berlin, und Vieweg Verlagsgesellschaft, Wiesbaden, 1989.

[31] Cordes, D. und E. Kaucher: *Self-Validating Computations for Sparse Matrix Problems*. In: Kaucher, E., U. Kulisch und C. Ullrich (Hrsg.): *Computerarithmetic: Scientific Computation and Programming Languages*. B.G. Teubner Verlag, Stuttgart, 1987.

[32] Darema, F.: *A single-program-multiple-data computational model for Epex/Fortran*. Parallel Computing, 5(7):11–24, 1988.

[33] Davidenkoff, A.: *Arithmetische Ausstattung von Parallelrechnern für zuverlässiges numerisches Rechnen*. Dissertation, Universität Karlsruhe, 1992.

[34] Davis, T.: *SPARSEINV: a MATLAB toolbox for computing the sparse inverse subset using the Takahashi equations*. `http://www.cise.ufl.edu/research/sparse/SuiteSparse`.

[35] Davis, T. A.: *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. SIAM, Philadelphia, 2006.

[36] Davis, T. A. und I. S. Duff: *Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method*. ACM Transactions on Mathematical Software, 25(1):1–19, 1999.

[37] Davis, T. A., J. R. Gilbert, S. Larimore und E. Ng: *Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm*. ACM Transactions on Mathematical Software, 30(3):377–380, 2004.

[38] Dekker, T. J.: *A floating-point technique for extending the available precision*. Numerical Mathematics, 18:224–242, 1971.

[39] Diep, N. H. und N. Revol: *Solving and Certifying the Solution of a Linear System*. Reliable Computing, 15(2):120–131, 2011.

[40] Dongarra, J.: *Basic Linear Algebra Subprograms Technical Forum Standard*. International Journal of High Performance Applications and Supercomputing, 16(1 und 2):1–111 and 115–199, 2002.

[41] Drepper, U.: *What Every Programmer Should Know About Memory*. `http://people.redhat.com/drepper/cpumemory.pdf`, 2007.

[42] Dreyer, A.: *Interval Analysis of Analog Circuits with Component Tolerances*. Dissertation, TU Kaiserslautern, 2005.

[43] Flynn, M.: *Some Computer Organizations and Their Effectiveness*. IEEE Trans. Comput., C-21(9), 1972.

[44] Frommer, A.: *Parallele Algorithmen*. Vorlesungsmitschrift WS 2005/06, 2005.

[45] George, A. und J. Liu: *The evolution of the Minimum Degree Ordering Algorithm*. SIAM Review, 31(1):1–19, 1989.

[46] Gilbert, J. R., C. Moler und R. Schreiber: *Sparse matrices in Matlab: Design and implementation*. SIAM Journal on Matrix Analysis and Applications, 13(1):333–356, 1992.

[47] Goto, K. und R. van de Geijn: *High-Performance Implementation of the Level-3 BLAS*. ACM Transactions on Mathematical Software, 35(1), 2008.

[48] Gregory, R. T. und D. L. Karney: *A Collection of Matrices for Testing Computational Algorithms*. Wiley-Interscience, 1969.

[49] Grimmer, M.: *An MPI Extension for the Use of C-XSC in Parallel Environments*. BUW-WRSWT Preprints, 2005(3), 2005.

[50] Grimmer, M.: *Selbstverifizierende mathematische Softwarewerkzeuge im High Performance Computing*. Dissertation, Universität Wuppertal, 2007.

[51] Grote, M. und T. Huckle: *Parallel preconditioning with sparse approximate inverses*. SIAM Journal on Scientific Computing, 18(3):838–853, 1997.

[52] Hammer, R.: *Maximal genaue Berechnung von Skalarproduktausdrücken und hochgenaue Auswertung von Programmteilen*. Dissertation, Universität Karlsruhe, 1992.

[53] Hammer, R., M. Hocks, U. Kulisch und D. Ratz: *Numerical Toolbox for Verified Computing I: Basic Numerical Problems*. Springer Verlag, Heidelberg, 1993.

[54] Hammer, R., M. Hocks, U. Kulisch und D. Ratz: *C++ Toolbox for Verified Computing*. Springer Verlag, Heidelberg, 1995.

[55] Hashemi, B. und M. Dehghan: *The interval Lyapunov matrix equation: Analytical results and an efficient numerical technique for outer estimation of the united solution set.* Mathematical and Computer Modelling, 55(3-4):622–633, 2012.

[56] Hauser, J. R.: *Handling Floating-Point Exceptions in Numeric Programs.* ACM Transactions on Programming Languages and Systems, 18(2):139–174, 1996.

[57] Heindl, G.: *Solving Linear Interval Systems Is NP-Hard Even If We Exclude Overflow and Underflow.* Reliable Computing, 4(4):383–388, 1998.

[58] Higham, N. J.: *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, 2. Aufl., 2002.

[59] Hofschuster, W. und W. Krämer: *C-XSC 2.0: A C++ Library for Extended Scientific Computing.* In: *Numerical Software with Result Verification*, Lecture Notes in Computer Science. Springer Verlag, Heidelberg, 2004.

[60] Hölbig, C. und W. Krämer: *Selfverifying solvers for Dense Systems of Linear Equations Realized in C-XSC.* BUW-WRSWT Preprints, 2003(1), 2003.

[61] Hölbig, C., W. Krämer und T. A. Diverio: *An Accurate and Efficient Selfverifying Solver for Systems with Banded Coefficient Matrix.* In: *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, S. 283–290, Amsterdam, 2004. Elsevier Science.

[62] IBM: *High Accuracy Arithmetic - Extended Scientific Computation (ACRITH-XSC).* IBM Corp., 1990. GC 33-6462-00.

[63] Intel: *Intel C++ Compiler User and Reference Guides.* Intel, 2011.

[64] Intel: *Intel Math Kernel Library Reference Manual.* Intel, 2011.

[65] ISO/IEC: *ISO C Standard 2011.* Norm ISO/IEC 9899:2011, ISO, Genf, Schweitz, 2011.

[66] ISO/IEC: *ISO C++ Standard 2011.* Norm ISO/IEC 14882:2011, ISO, Genf, Schweiz, 2011.

[67] Jansson, C.: *Calculation of exact bounds for the solution set of linear interval systems.* Linear Algebra Applications, 251:321–340, 1997.

[68] Karimi, K., N. G. Dickson und F. Hamze: *A Performance Comparison of CUDA and OpenCL.* CoRR, abs/1005.2581, 2010.

[69] Karniadakis, G. E. und R. M. K. II: *Parallel Scientific Computing in C++ and MPI.* Cambridge University Press, Cambridge, 2003.

[70] Kaucher, E.: *Interval Analysis in the Extended Interval Space IR.* Computing Suppl., 2:33–49, 1980.

[71] Kearfoot, R., M. Nakao, A. Neumaier, S. Rump, S. Shary und P. van Hentenryck: *Standardized Notation in Interval Analysis.* TOM, 15(1), 2010.

[72] Kersten, T.: *Verifizierende rechnerinvariante Numerikmodule.* Dissertation, Universität Karlsruhe, 1998.

[73] Khronos OpenCL Working Group: *The OpenCL Specification 1.1.* `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[74] Klatte, R., U. Kulisch, M. Neaga, D. Rat und C. Ullrich: *PASCAL-XSC - Sprachbeschreibung mit Beispielen.* Springer Verlag, Heidelberg, 1991.

[75] Klatte, R., U. Kulisch, A. Wiethoff, C. Lawo und M. Rauch: *C-XSC - A C++ Class Library for Extended Scientific Computing.* Springer Verlag, Heidelberg, 1993.

[76] Knuth, D. E.: *The Art of Computer Programing*, Bd. 2 - Seminumerical Algorithms. Addison Wesley, 3. Aufl., 1998.

[77] Kolberg, M. L.: *Parallel Self-Verified Solver for Dense Linear Systems.* Dissertation, Pontificia Universidade Catolica do Rio Grande do Sul, Porto Alegre, 2009.

[78] Kolberg, M. L., W. Krämer und M. Zimmer: *Efficient parallel solvers for large dense systems of linear interval equations.* Reliable Computing, 15:193–206, 2011.

[79] Kolev, L.: *Interval Methods for Circuit Analysis.* World Scientific, 1993.

[80] Kolev, L.: *Worst-case tolerance analysis of linear DC and AC electric circuits.* IEEE Transactions on Circuits and Systems, 49(12):1–9, 2002.

[81] Krämer, W.: *Verifikationsnumerik I.* Vorlesungsmitschrift WS 2006/07, 2006.

[82] Krämer, W.: *Verifikationsnumerik II.* Vorlesungsmitschrift SS 2007, 2007.

[83] Krämer, W., U. Kulisch und R. Lohner: *Numerical Toolbox for Verified Computing II: Advanced Numerical Problems.* Springer Verlag, 2009.

[84] Krämer, W. und M. Zimmer: *Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS.* In: *Numerical Validation in Current Hardware Architectures*, Bd. 5492 d. Reihe *Lecture Notes in Computer Science*, S. 230–249. Springer-Verlag, Heidelberg, 2009.

[85] Krawczyk, R.: *Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken.* Computing, 4:187–201, 1969.

[86] Kreinovich, V., A. Lakeyev und S. Noskov: *Optimal solution of interval linear systems is intractable (NP-hard).* Interval Computations, 1:6–14, 1993.

[87] Krier, R.: *Komplexe Kreisarithmetik.* Dissertation, Universität Karlsruhe, 1973.

[88] Kulisch, U.: *Die fünfte Gleitkommaoperation für Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation.* Universität Karlsruhe, 1997.

[89] Kulisch, U. W. und W. L. Miranker: *The arithmetic of the digital computer: A new approach.* SIAM Rev., 28:1–40, March 1986, ISSN 0036-1445.

[90] Li, X. S., J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung und D. J. Yoo: *Design, implementation and testing of extended and mixed precision BLAS.* ACM Trans. Math. Softw., 28(2):152–205, Juni 2002, ISSN 0098-3500.

[91] Mariana Luderitz Kolberg, W. K. und M. Zimmer: *A Note on Solving Problem 7 of the SIAM 100-Digit Challenge Using C-XSC.* In: *Numerical Validation in Current Hardware Architectures*, Bd. 5492 d. Reihe *Lecture Notes in Computer Science*, S. 250–261. Springer-Verlag, Heidelberg, 2009.

[92] Markowitz, H. M.: *The elimination form of the inverse and its application to linear programming.* Management Science, 3(3):255–269, 1957.

[93] Meister, A.: *Numerik linearer Gleichungssysteme.* Vieweg+Teubner Verlag, 2011.

[94] Message Passing Interface Forum: *MPI: A Message Passing Interface Standard.* University of Tennesse, Knoxville, Tennesse, 1993-1995.

[95] Message Passing Interface Forum: *MPI-2: Extensions to the Message Passing Interface.* University of Tennesse, Knoxville, Tennesse, 1995-1997.

[96] Meyers, S.: *Effektiv C++ Programmieren: 55 Möglichkeiten, Ihre Programme und Entwürfe zu verbessern.* Addison Wesley, 2011.

[97] Moore, R. E.: *Interval Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

[98] Moore, R. E.: *A test for existence of solutions for non-linear systems.* SIAM Journal on Numerical Analysis, 14(4):611–615, 1977.

[99] Moore, R. E.: *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[100] Muller, J. M., N. Brisebarre, F. de Dinechin, C. P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé und S. Torres: *Handbook of Floating-Point Arithmetic.* Birkhäuser Boston, 2010.

[101] Neumaier, A.: *Overestimation in linear interval equations.* SIAM Journal on Numerical Analysis, 24(1):207–214, 1987.

[102] Neumaier, A.: *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, 1990.

[103] Neumaier, A. und A. Pownuk: *Linear Systems with Large Uncertainties, with Applications to Truss Structures*. Reliable Computing, 13:149–172, 2007.

[104] N.P. Seif, S.A. Hussein, A. D.: *The interval Sylvester matrix equation*. Computing, 52:233–244, 1994.

[105] Numerik Software GmbH: *PASCAL-XSC: A PASCAL Extension for Scientific Computation. User's Guide*. Numerik Software GmbH, 1991.

[106] Ogita, T., S. M. Rump und S. Oishi: *Accurate sum and dot product*. SIAM Journal on Scientific Computing, 26(6):1955–1988, 2005.

[107] Oishi, S. und S. M. Rump: *Fast Verification of Solutions of Matrix Equations*. Numerical Mathematics, 90(4):755–773, 2002.

[108] Oishi, S., K. Tanabe, T. Ogita, S. M. Rump und N. Yamanaka: *A Parallel Algorithm of Accurate Dot Product*. Parallel Computing, 34:392–410, 2007.

[109] OpenMP Architecture Review Board: *OpenMP Application Programming Interface 3.1*. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[110] Ozaki, K., T. Ogita, S. Oishi und S. M. Rump: *Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications*. Numerical Algorithms, 59(1):95–118, 2012.

[111] Pacheco, P.: *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[112] Popova, E.: *Strong Regularity of Parametric Interval Matrices*. In: *Mathematics & Education in Mathematics, Proceedings of the 33rd Spring Conference of UBM*, S. 446–451, 2004.

[113] Popova, E.: *Explicit Description of AE Solution Sets to Parametric Linear Systems*. Bulgarian Academy of Sciences Preprint, 7, 2011. Erscheint in SIAM Journal of Matrix Analysis and Applications.

[114] Popova, E., L. Kolev und W. Krämer: *A Solver for Complex-Valued Parametric Linear Systems*. Serdica Journal of Computing, 4(1):123–132, 2010.

[115] Popova, E. und W. Krämer: *Parametric Fixed-Point Iteration Implemented in C-XSC*. BUW-WRSWT Preprints, 2003(3), 2003.

[116] Popova, E. und W. Krämer: *Inner and Outer Bounds for the Solution Set of Parametric Linear Systems*. Journal of Computational and Applied Mathematics, 199(2):310–316, 2007.

[117] Ratschek, H.: *Die Subdistributivität der Intervallarithmetik*. ZAMM, 51(3):189–192, 1971.

[118] Rohn, J.: *VerSoft - Verification Software in Matlab/Intlab.* `http://uivtx.cs.cas.cz/~rohn/matlab`.

[119] Rohn, J.: *Computing exact componentwise bounds on solutions of linear systems with interval data is NP-hard.* SIAM Journal on Matrix Analysis and Applications, 16(2):415–420, 1995.

[120] Rohn, J.: *Linear Interval Equations: Computing Enclosures with Bounded Relative Overestimation is NP-Hard.* In: Kearfoot, R. und V. Kreinovich (Hrsg.): *Applications of Interval Computations*, S. 81–89. Kluwer Academic Press, 1996.

[121] Rohn, J.: *An Algorithm for Solving the Absolute Value Equation.* Electronic Journal of Linear Algebra, 18:589–599, 2009.

[122] Rohn, J.: *An Algorithm for Solving the Absolute Value Equation: An Improvement.* Techn. Ber. V-1063, Institute of Computer Science, Academy of Sciences of the Czech Republic, 2010.

[123] Rump, S. M.: *Kleine Fehlerschranken bei Matrixproblemen.* Dissertation, Universität Karlsruhe, 1980.

[124] Rump, S. M.: *Solving Algebraic Problems with High Accuracy.* In: Kulisch, U. W. und W. L. Miranker (Hrsg.): *A New Approach to Scientific Computation*, S. 51–120. Academic Press, New York, 1983.

[125] Rump, S. M.: *Rigorous sensitivity analysis for systems of linear and non-linear equations.* Math. Comput., 54(190):721–736, 1990.

[126] Rump, S. M.: *Validated Solution of Large Linear Systems.* In: Albrecht, R., G. Alefeld und H. Stetter (Hrsg.): *Validation numerics: theory and applications*, Bd. 9 d. Reihe *Computing Supplementum*, S. 191–212. Springer, 1993.

[127] Rump, S. M.: *Verification Methods for Dense and Sparse Systems of Equations.* In: Herzberger, J. (Hrsg.): *Topics in Validated Numerics*, Studies in Computational Mathematics, S. 63–136. Elsevier, Amsterdam, 1994.

[128] Rump, S. M.: *Fast and parallel interval arithmetic.* BIT, 39(3):539–560, 1999.

[129] Rump, S. M.: *Ill-conditioned Matrices are componentwise near to singularity.* SIAM Review, 41(1):102–112, 1999.

[130] Rump, S. M.: *Intlab - Interval Laboratory.* In: Csendes, T. (Hrsg.): *Developments in Reliable Computing*, S. 77–104. Kluwer Academic Press, 1999.

[131] Rump, S. M.: *Verification of Positive Definiteness.* Numerical Mathematics, 46:433–452, 2006.

[132] Rump, S. M.: *Verification Methods: Rigorous results using floating-point arithmetic.* Acta Numerica, 19:287–449, 2010.

[133] Sanders, J. und E. Kandrot: *CUDA by example. An Introduction to General Purpose GPU Programming*. Addison-Wesley Longman, Amsterdam, 2010.

[134] Shashikhin, V.: *Robust assignment of poles in large-scale interval systems*. Autom. Remote Control, 63(2):200–208, 2002.

[135] Shashikhin, V.: *Robust stabilization of linear interval systems*. Journal of Applied Mathematics and Mechanics, 66(3):393–400, 2002.

[136] Sluis, A. van der: *Condition number and equilibration of matrices*. Numerical Mathematics, 14:14–23, 1969.

[137] Spaniol, O.: *Die Distributivität in der Intervallarithmetik*. Computing, 5(1):6–16, 1970.

[138] Stallman, R. M. und die GCC Developer Community: *Using the GNU Compiler Collection 4.6.1*. GNU Press, 2010.

[139] Stoer, J. und R. Bulirsch: *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[140] Valgrind Developers: *Valgrind User Manual*. `http://valgrind.org/docs/manual/manual.html`, 2010.

[141] Whaley, R. C. und A. Petitet: *Minimizing development and maintenance costs in supporting persistently optimized BLAS*. Software: Practice and Experience, 35(2):101–121, 2005.

[142] Wiethoff, A.: *Verifizierte globale Optimierung auf Parallelrechnern*. Dissertation, Universität Karlsruhe, 1997.

[143] Wilkinson, J. H.: *Rounding Errors in Algebraic Processes*. Prentice Hall, NJ, USA, 1963.

[144] Zienkiewicz, O. und R. Taylor: *Finite Element Method: Volume 1. The Basis.* Butterworth Heinemann, London, 2000.

[145] Zimmer, M.: *Implementierung eines Präcompilers für Skalarproduktausdrücke in C-XSC*. Bachelor Thesis, Bergische Universität Wuppertal, 2005.

[146] Zimmer, M.: *Laufzeiteffiziente, parallele Löser für lineare Intervallgleichungssysteme in C-XSC*. Master Thesis, Bergische Universität Wuppertal, 2007.

[147] Zimmer, M.: *Using C-XSC in a Multi-Threaded Environment*. BUW-WRSWT Preprints, 2011(2), 2011.

[148] Zimmer, M., W. Krämer, G. Bohlender und W. Hofschuster: *Extension of the C-XSC Library With Scalar Products With Selectable Accuracy*. Serdica Journal of Computing, 4(3):349–370, 2010.

[149] Zimmer, M., W. Krämer und W. Hofschuster: *Sparse Matrices and Vectors in C-XSC*. Reliable Computing, 14:138–160, 2009.

[150] Zimmer, M., W. Krämer und W. Hofschuster: *Using C-XSC for High Performance Verified Computing*. In: *Applied Parallel and Scientific Computing*, Bd. 7134 d. Reihe *Lecture Notes in Computer Science*, S. 168–178. Springer-Verlag, Heidelberg, 2012.

[151] Zimmer, M., W. Krämer und E. Popova: *Solvers for the verified solution of parametric linear systems*. Computing, 94(2-4):109–123, 2011.

Stand der angegebenen Weblinks: 25. Februar 2013