Selbstverifizierende mathematische Softwarewerkzeuge im High Performance Computing

Konzeption, Entwicklung und Analyse
am Beispiel der parallelen verifizierten Lösung
linearer Fredholmscher Integralgleichungen zweiter Art

Markus Grimmer

Am Fachbereich C - Mathematik und Naturwissenschaften der Bergischen Universität Wuppertal genehmigte Dissertation zur Erlangung des akdemischen Grads eines Doktors der Naturwissenschaften (Dr. rer. nat.) von Dipl.-Math. Markus Grimmer aus Wuppertal

Tag der mündlichen Prüfung: 8.5.2007
Erstgutachter: Prof. Dr. Walter Krämer
Zweitgutachter: Prof. Dr. Bruno Lang

Danke!

Ich danke allen, die mich auf dem Weg zu dieser Arbeit unterstützt haben. Danke an Prof. Dr. Walter Krämer für die fachlich und menschlich ausgezeichnete Betreuung. Danke an Prof. Dr. Bruno Lang für die Zweitbegutachtung dieser Arbeit. Danke an Dr. Werner Hofschuster für die freundschaftliche Zusammenarbeit, für viele gute Ratschläge und Hilfe und für das Korrekturlesen. Danke an das gesamte *ALiCEnext*-Team für alle Tipps und technische Hilfe. Danke an Ulrich Engler, der wahrscheinlich an allem schuld ist, ohne es zu wissen. Und Danke an Kerstin und an meine Eltern dafür, dass Ihr immer dagewesen seid!

Inhaltsverzeichnis

1 Einführung **1**

2 Grundlagen und Erweiterungen **5**

2.1 Intervallarithmetik 6
2.2 Rechnerarithmetik 11
2.3 Taylorarithmetik 14
2.3.1 Funktoide 16
2.3.2 Rekursive Berechnung von Taylorkoeffizienten . . 17
2.3.3 Neue Rekursionsformeln 22
2.4 Steigungsarithmetik 25
2.5 Integralgleichungen 28
2.5.1 Integralgleichungen mit entartetem Kern 31
2.5.2 Integralgleichungen mit nicht entartetem Kern . . 34
2.5.3 Integralgleichungssysteme 35
2.6 Paralleles Rechnen 44

3 Verfahren **51**

3.1 Kleins Verfahren für lineare Fredholmsche Integralgleichungen und Integralgleichungssysteme zweiter Art . . 54
3.1.1 Das praktische Verfahren 58
3.1.2 Bemerkungen zur Verwendung der Taylorentwicklung 59
3.1.3 Bemerkungen zum verifizierten Verfahren 62
3.1.4 Der Algorithmus für das Einzelverfahren 64
3.1.5 Das Verfahren für Integralgleichungssysteme . . . 68
3.1.6 Der Algorithmus für das Systemverfahren 70
3.1.7 Lösung von Integralgleichungssystemen zur Verbesserung der Lösung von Integralgleichungen . 75
3.2 Lösungsverfahren für Teilprobleme 77

3.2.1 Rumps Verfahren zur verifizierten Lösung linearer Gleichungssysteme 78
3.2.2 Verfahren zur Matrixinversion 81
3.3 Obermaiers Verfahren für nichtlineare Fredholmsche Integralgleichungen zweiter Art 83
3.3.1 Die Näherungslösung 85
3.3.2 Die weiteren Teilschritte 88

4 Parallele Verfahren 93
4.1 Aufwandsabschätzungen 93
4.2 Paralleles Verfahren zur Lösung linearer Fredholmscher Integralgleichungssysteme zweiter Art 96
4.3 Parallele Teilverfahren 103
4.3.1 Paralleles Verfahren zur verifizierten Lösung linearer Gleichungssysteme 104
4.3.2 Parallele Matrixmultiplikation 108
4.3.3 Paralleles Verfahren zur Matrixinversion 109
4.4 Obermaiers paralleles Verfahren 110

5 Softwarekomponenten - Erweiterungen und Neuentwicklungen 115
5.1 Werkzeuge und Bibliotheken 117
5.1.1 C-XSC 117
5.1.2 FILIB++ 122
5.1.3 MPI 125
5.2 Taylorarithmetik in C-XSC 127
5.2.1 Existierende Implementierung 128
5.2.2 Update und Erweiterung 128
5.3 MPI-Kommunikationspaket für C-XSC-Datentypen 132
5.3.1 Verarbeitung benutzerdefinierter Datentypen mit MPI 133
5.3.2 Tools 136
5.3.3 Das neue Kommunikationspaket 137
5.3.4 Erweiterung für Taylorarithmetik und STL-Vektoren 146
5.4 Paralleler Intervall-Gleichungssystem-Löser in C-XSC 149
5.4.1 Existierende Implementierungen 149
5.4.2 Neue parallele Implementierung 153

5.5 Integralgleichungslöser nach Klein 162
5.5.1 Existierende Implementierung 162
5.5.2 Neue Implementierung 165
5.5.3 Neue Implementierung für Integralgleichungssysteme 173
5.5.4 Neue parallele Implementierung 181
5.6 Ergebnisexport und Visualisierung in Maple mit intpakX . 185
5.7 Integralgleichungslöser nach Obermaier 190
5.7.1 Existierende Implementierung 190
5.7.2 Portierung 192
5.8 Weitere Anwendungen des MPI-Kommunikationspakets für C-XSC-Datentypen 195

6 Test und Ergebnisse 199
6.1 Hard- und Softwareumgebung 200
6.2 Parallele Kommunikation 201
6.2.1 Parallele Kommunikation mit C-XSC-Datentypen . 201
6.2.2 Netzwerkstruktur 203
6.3 Paralleler verifizierter Integralgleichungslöser 204
6.3.1 Testkriterien im Überblick 205
6.3.2 Entwicklung der Genauigkeit bei variabler Taylorordnung bzw. variabler Systemordnung 207
6.3.3 Entwicklung der Rechenzeit bei variabler Taylorordnung bzw. variabler Systemordnung 209
6.3.4 Parallele Performance 211
6.3.5 Anteile einzelner Programmteile an der Rechenzeit 215
6.3.6 Ein Beispiel 217
6.3.7 Vergleich mit der Software aus [72] 219
6.3.8 Vergleich mit der Software nach Obermaier aus Abschnitt 5.7 223

7 Zusammenfassung und Weiterentwicklung 229
7.1 Zusammenfassung: Verfahren, Softwarekomponenten, Ergebnisse 229
7.2 Weiterentwicklung 232

Anhang A: Installation und Anwendung der seriellen Software 235

Anhang B: Installation und Anwendung der parallelen Software auf ALiCEnext 239

1 Einführung

Numerische Vefahren haben oft *Näherungslösungen* als Ergebnis, die das exakte Ergebnis eines Rechenverfahrens approximieren und durch Rundungs- und andere Fehler bei der praktischen Berechnung beeinflusst sind. Ohne die zusätzliche Berechnung von Fehlerschranken kann jedoch keine mathematische Aussage über die Güte des Ergebnisses getroffen werden. Methoden des *verifizierten Rechnens* suchen den Fehler bereits während der Berechnung zu erfassen. Die Intervallarithmetik stellt Rechenoperationen und Funktionen zur Verfügung, bei denen statt mit reellen Zahlen mit abgeschlossenen *Intervallen* reeller Zahlen gerechnet wird. Die Operationen der Intervallarithmetik sind so gestaltet, dass das Ergebnisintervall jeder Operation, sofern es wohldefiniert ist, das exakte mathematische Ergebnis der zu Grunde liegenden Punktoperation einschließt. Diese Eigenschaft wird zur Entwicklung verifizierter Rechenverfahren verwendet, die, sofern sie erfolgreich durchgeführt wurden, ein Intervall zum Ergebnis haben, das das exakte Ergebnis des Verfahrens einschließt. Mit der C++-Klassenbibliothek C-XSC steht eine Implementierung der Intervallarithmetik in C++ zur Verfügung, die zur Erstellung der Software in dieser Arbeit verwendet wurde.

Die numerische Lösung von Integralgleichungen stellt einen Anwendungsbereich für verifizierte Rechenverfahren dar. Integralgleichungen treten in verschiedenen theoretischen und anwendungsbezogenen Problemen auf, etwa der Bestimmung von Wärmeabstrahlung paralleler Platten (siehe Beispiel in Abschnitt 6.3.6).

Viele Rechenverfahren erfordern hohen Rechenaufwand, der auch auf aktueller Hardware Stunden oder Tage betragen kann. Im *High Performance Computing* wird der Zeitaufwand zur Berechnung der Ergebnisse reduziert, indem spezielle Verfahren zur parallelen Verarbeitung auf mehreren Prozessoren konzipiert und auf entsprechenden Parallelrechnern (z.B.

Großrechnern oder Rechnerclustern) ausgeführt werden. Neben speziell zur parallelen Verarbeitung entwickelten Anwendungen müssen auch verwendete externe Komponenten parallelisiert werden und geeignete Werkzeuge zur Kommunikation in parallelen Umgebungen bereitgestellt werden. Dabei sollen die entwickelten Werkzeuge sowohl zu existierenden Anwendungen und Bibliotheken passen und ihren Einsatzbereich erweitern als auch in neue Anwendungen integrierbar sein oder Wege der Weiterverwendung von Ergebnissen in anderen Umgebungen aufzeigen. Oft liegt in diesem Zusammenhang auch Software vor, die in einer jeweils aktuellen Einsatzumgebung nicht funktionsfähig ist und des Redesigns oder der Modifikation bedarf.

Diese Problemstellungen werden in dieser Arbeit am Beispiel der parallelen verifzierten Lösung linearer Fredholmscher Integralgleichungen zweiter Art[1] betrachtet. Aufbauend auf bestehenden Vorgehensweisen werden neue parallele Verfahren entwickelt und Softwarekomponenten zur Umsetzung dieser Verfahren implementiert. Dabei werden neben den Anwendungen, die die genannten Verfahren implementieren, verschiedene Softwarekomponenten neu entwickelt, erweitert oder modifiziert: Einerseits die Anwendungen selbst, andererseits die Werkzeuge, die zur Umsetzung der Verfahren allgemein oder speziell in parallelen Umgebungen benötigt werden, die aber gleichzeitig als eigenständige Komponenten auch unabhängig von der Anwendungssoftware eingesetzt werden können und in andere existierende Anwendungen integrierbar sind. Neben parallelen Komponenten für mathematische Teilverfahren wie der verifizierten Lösung linearer Gleichungssysteme, die in zahlreichen unterschiedlichen Verfahren auftritt, sind dies im Rahmen der parallelen verifzierten Lösung linearer Fredholmscher Integralgleichungen zweiter Art die Taylorarithmetik zur Darstellung von Funktionen, ein Interface zur Verwendung der hier zum Einsatz kommenden C++-Klassenbibliothek C-XSC in parallelen Umgebungen mit MPI sowie ein Paket zur Verwendung der Intervall-Arithmetik in dem Computer-Algebra-System Maple, mit dem die gewonnenen Ergebnisse visualisiert und weiterverwendet werden können.

Die Arbeit ist wie nachfolgend beschrieben gegliedert.

In Kapitel 2 werden die mathematischen Grundlagen für die betrachteten Probleme angegeben und Erweiterungen entwickelt. Es werden die

[1] Zur Klassifikation von Integralgleichungen siehe Abschnitt 2.5.

wichtigsten Begriffe der Intervallarithmetik sowie Rechnerarithmetik eingeführt und wichtige Definitionen und Eigenschaften von Integralgleichungen angegeben, zudem werden die Taylorarithmetik und die Steigungsarithmetik zur Darstellung von Funktionen beschrieben, die in den später vorgestellten Verfahren Verwendung finden. Darüberhinaus werden neue Rekursionsformeln zur Anwendung der Taylorarithmetik auf bisher nicht betrachtete Funktionen entwickelt.

In Kapitel 3 werden die den Anwendungen zu Grunde liegenden Verfahren zur Lösung von Integralgleichungen dargestellt. Als Verfahren, das den zentralen Anwendungen dieser Arbeit, dem seriellen und parallelen Löser für lineare Fredholmsche Integralgleichungen zweiter Art, zu Grunde liegt, wird das Verfahren von Klein [72] dargestellt, das die Lösung einer linearen Fredholmschen Integralgleichung zweiter Art durch die Aufteilung in einen entarteten und einen nicht entarteten Anteil sowie den Übergang zu Integralgleichungssystemen zur Lösung der einzelnen Integralgleichung beinhaltet und u.a. auf Betrachtungen von Dobner [25] zurückgeht. Als Teilverfahren wird das Verfahren von Rump zur verifizierten Lösung linearer Gleichungssysteme beschrieben. Zusätzlich wird als Vergleichsverfahren das Verfahren von Obermaier [106] zur Lösung von nichtlinearen Fredholmschen Integralgleichungen zweiter Art erläutert.

Die genannten Verfahren haben die Eigenschaft, dass sie für wachsende Problemgrößen nur noch unter sehr hohem Zeitaufwand durchführbar sind und es somit aus Gründen des Aufwands nicht möglich ist, Lösungen von Integralgleichungen mit optimaler Einschließungsgenauigkeit zu berechnen. Zu den vorstehenden Verfahren werden daher in Kapitel 4 neue parallele Verfahren entwickelt, die die Anwendung der Verfahren in parallelen Umgebungen und somit eine drastische Reduktion des Zeitaufwands erlauben.

Im Rahmen dieser Arbeit wurden zahlreiche neue Softwarekomponenten erstellt. Zum einen handelt es sich um neu konzipierte Implementierungen der seriellen und parallelen Verfahren für Integralgleichungen, zum anderen um Werkzeuge zur Umsetzung der Verfahren in parallelen Umgebungen. Insbesondere wird mit den neuen Komponenten das Anwendungsspektrum der Klassenbibliothek C-XSC erweitert. Die einzelnen Komponenten werden in Kapitel 5 vorgestellt. Zunächst werden die Werkzeuge

und Bibliotheken beschrieben, die zur Implementierung der parallelen Anwendungen verwendet wurden. Danach folgen Beschreibungen der neuen und erweiterten Softwarekomponenten für Taylorarithmetik und parallele Kommunikation mit C-XSC und MPI sowie des neuen parallelen linearen Gleichungssystem-Lösers. Schließlich werden die seriellen und parallelen Versionen des neuen linearen Integralgleichungslösers vorgestellt, gefolgt von Beschreibungen der neu geschaffenen Möglichkeiten zur Weiterverwendung der Ergebnisse in Maple. Weiterhin wird eine modifizierte Implementierung des Integralgleichungslösers nach Obermaier als Vergleichssoftware vorgestellt. Zusätzlich wird zur Demonstration des neu entwickelten MPI-Kommunikationspakets für C-XSC-Datentypen eine modifizierte Implementierung einer Anwendung aus dem Gebiet der globalen Optimierung angegeben.

Die vorgestellte Software wurde unter unterschiedlichen Aspekten getestet und die Ergebnisse analysiert. Hier liegt ein Schwerpunkt auf der Betrachtung der Rechenzeiten insbesondere des parallelen Integralgleichungslösers und der erreichten Einschließungsgenauigkeit, aber auch die weiteren neuen Komponenten werden untersucht. Schließlich werden Vergleiche mit den existierenden alten Implementierungen und zwischen den unterschiedlichen Verfahren gezogen. Die Ergebnisse dieser Tests sind in Kapitel 6 zu finden.

Abschließend werden in Kapitel 7 die Ergebnisse noch einmal zusammengefasst und einige Möglichkeiten für weitere Untersuchungen und Weiterentwicklung der Software angegeben.

2 Grundlagen und Erweiterungen

In diesem Kapitel sollen die Grundlagen dargelegt werden, die für die betrachteten Verfahren zur verifizierten Lösung linearer Fredholmscher Integralgleichungen und Integralgleichungssysteme zweiter Art sowie nichtlinearer Fredholmscher Integralgleichungen zweiter Art vom Urysohntyp benötigt werden. Insbesondere werden verschiedene Arithmetiken angegeben, die auch als Grundlage für die praktische Umsetzung der später beschriebenen Verfahren und Implementierungen dienen.

Gleichzeitig wird im Fall der Taylorarithmetik eine Erweiterung auf neue, bislang nicht von der Arithmetik abgedeckte Funktionen vorgenommen.

Die Grundlagen gliedern sich in folgende Bereiche:

Intervallarithmetik

Die Intervallarithmetik bietet die Möglichkeit, *verifizierte* numerische Berechnungen auszuführen, d.h. garantierte mathematische Aussagen über Ergebnisse von Berechnungen zu treffen. Grundlagen der Intervallarithmetik werden im ersten Abschnitt dieses Kapitels erläutert.

Rechnerarithmetik

Im Abschnitt *Rechnerarithmetik* wird auf die konkrete Umsetzung von Intervallarithmetik auf dem Rechner Bezug genommen, und die Möglichkeiten und Beschränkungen von dort zu verwendenden endlichen Zahlsystemen werden geschildert.

Funktionswerteinschließungen

Um den Wertebereich von Funktionen einzuschließen, existieren verschiedene Vorgehensweisen. Neben der direkten Auswertung von Funktionsausdrücken für Intervalle sind *Taylorarithmetik* und *Steigungsarithmetik* hierzu einsetzbar. Insbesondere können die dort Verwendung findenden *Taylorkoeffizienten* und *Steigungen* auf dem Rechner automatisch berechnet werden.

Integralgleichungen

Die Definition und die grundlegenden Eigenschaften von Integralgleichungen werden in einem eigenen Abschnitt vorgestellt.

Paralleles Rechnen

Zuletzt werden einige Grundlagen des parallelen Rechnens angegeben, sowohl in Bezug auf den Aufbau von Hardware als auch im Hinblick auf die Erstellung von Software für Parallelrechner.

2.1 Intervallarithmetik

Numerische Verfahren liefern oft Näherungslösungen für die jeweils betrachteten Probleme. Abhängig von unterschiedlichen Parametern und Einflüssen können gleichermaßen *exakte* Ergebnisse oder gute Näherungen, aber auch Ergebnisse erzielt werden, die nicht einmal mehr als Näherung bezeichnet werden können; oft genug ist die Güte des Ergebnisses schon deshalb schwer zu beurteilen, weil das exakte Ergebnis unbekannt ist. Um die Genauigkeit des Ergebnisses dennoch abzuschätzen, werden oft zusätzliche Verfahren durchgeführt, um die Größe des Approximationsfehlers zu bestimmen.

Gleichzeitig sind alle konkreten Implementierungen numerischer Algorithmen auch verfahrensunabhängig mit Rundungsfehlern behaftet, da auf real existierenden Computern nur ein *endliches* Zahlensystem benutzt werden kann (siehe Abschnitt 2.2). Bereits bei einfachsten Rechnungen kann das Ergebnis durch Rundungsfehler völlig unbrauchbar werden.

Die *Intervallarithmetik* rechnet mit *Intervallen* zur Einschließung von reellen Zahlen. Alle Operationen der reellen Arithmetik werden dabei durch besonders angepasste Intervall-Operationen ersetzt, so dass ein Ergebnisintervall einer Intervall-Operation stets das exakte Ergebnis der zugehörigen reellen Operation enthält. Dabei ist die *Intervallarithmetik* sowohl auf Basis der reellen Zahlen als auch im Rahmen der Rechnerarithmetik und der dort verwendeten Zahlsysteme (siehe Abschnitt 2.2) darstellbar.

Die Intervallarithmetik wurde von *Moore* [96, 97], *Kulisch* [80, 81, 82] und anderen ausführlich beschrieben. Einführungen und detaillierte Darstellungen sind etwa in [2, 3] zu finden. Intervalloperationen können auch als Spezialfall allgemeiner Mengenoperationen aufgefasst werden. Derartige Operationen wurden bereits von *Young* [133] untersucht. Einige grundlegende Definitionen und Eigenschaften der Intervallarithmetik werden im Folgenden angegeben.

Definition 2.1.1 *Ein* (reelles) Intervall $[x]$ *ist definiert als*

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

$\inf([x]) := \underline{x}$ *und* $\sup([x]) := \overline{x}$ *heißen* Unter- und Obergrenze *von* $[x]$ *bzw.* Infimum *und* Supremum *von* $[x]$. *Die Menge aller rellen Intervalle wird mit* $\mathbb{IR}$ *bezeichnet. Ein* Punktintervall *ist ein Intervall, das nur ein Element enthält, d.h. ein Intervall* $[x] := [\underline{x}, \overline{x}]$ *mit* $\underline{x} = \overline{x}$.[1]

Da es sich, wie oben definiert, bei Intervallen um Mengen handelt, sind für Intervalle die bekannten Mengenoperationen (wie z.B. $\subset$, $\supset$ etc.) anwendbar. Zusätzlich werden der Vergleichsoperator

$$[x] \overset{\circ}{\subset} [y] :\equiv [x] \subset [y] \wedge \underline{x} > \underline{y} \wedge \overline{x} < \overline{y} \quad \text{([x] liegt \textit{im Inneren} von [y])}$$

sowie die *Intervallhülle*

$$[x] \underline{\cup} [y] := [\min(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$$

[1] Sofern es im inhaltlichen Zusammenhang nicht zu Missverständnissen führt, wird gelegentlich auch die vereinfachte Schreibweise x für ein Intervall $[x]$ gewählt.

definiert, da die Vereinigung zweier Intervalle nicht notwendigerweise ein Intervall im Sinne der obigen Definition darstellt.

Um von Intervall*arithmetik* zu sprechen, müssen insbesondere Rechenoperationen für Intervalle definiert werden. Die Grundoperationen $+,-,\cdot,/$ für zwei Intervalle $[x]$ und $[y]$ werden definiert durch die Ergebnismengen der reellen Operationen für $x \in [x]$ und $y \in [y]$:

Definition 2.1.2 *Sei* $\circ \in \{+,-,\cdot,/\}, [x],[y] \in \mathbb{IR}$*, sowie* $0 \notin [y]$*, falls* $\circ = /$*. Dann werden die folgenden Intervallverknüpfungen definiert* [2]*:*

$$[x] \circ [y] := \{z \in \mathbb{R} | z = x \circ y, x \in [x], y \in [y]\}.$$

Aus der Stetigkeit und Monotonie der reellen Operationen folgen die expliziten Darstellungen

$$\begin{aligned} [x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\ [x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\ [x] \cdot [y] &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \\ [x]/[y] &= [x] \cdot \left[\frac{1}{\overline{y}}, \frac{1}{\underline{y}}\right] \end{aligned}$$

Durch den Übergang von einer reellen Zahl x zum Intervall $[x, x]$ lassen sich auch gemischte Operationen von reellen Zahlen und Intervallen ausdrücken.

Des Weiteren werden noch die folgenden Definitionen getroffen:

Definition 2.1.3 *Seien* $[x],[y] \in \mathbb{IR}$.

$$\begin{aligned} mid([x]) &:= \frac{\underline{x} + \overline{x}}{2} \text{ heißt Mittelpunkt von } [x]. \\ diam([x]) &:= \overline{x} - \underline{x} \text{ heißt Durchmesser von } [x]. \end{aligned}$$

[2]Der Multiplikationspunkt wird bei der Darstellung von Produkten gelegentlich ausgelassen.

$$diam_{rel}([x]) := \begin{cases} \frac{diam([x])}{\min\{|x|, x\in[x]\}} & 0 \notin [x] \\ diam([x]) & sonst \end{cases}$$

heißt relativer Durchmesser von $[x]$.

$$rad([x]) := \frac{\overline{x} - \underline{x}}{2}$$ *heißt Radius von* $[x]$.

$$d([x],[y]) := \max\{|\underline{x} - \underline{y}|, |\overline{x} - \overline{y}|\}$$

heißt Abstand von $[x]$ *und* $[y]$.

Zunächst lassen sich folgende wichtige Eigenschaften der eingeführten Operationen zeigen:

Satz 2.1.1 *Für* $[x],[y],[z] \in \mathbb{IR}$ *gilt:*

- $[x] \circ [y] = [y] \circ [x], \ \circ \in \{+, \cdot\}$ (Kommutativität)
- $([x] \circ [y]) \circ [z] = [x] \circ ([y] \circ [z]), \ \circ \in \{+, \cdot\}$ (Assoziativität)
- *Die Operationen* $+, \cdot$ *besitzen die eindeutig bestimmten* neutralen Elemente $[0,0]$ *bzw.* $[1,1]$.
- $0 \in [x] - [x]$ *und* $1 \in [x]/[x]$, *aber für Intervalle, die keine Punktintervalle sind, existieren keine inversen Elemente bzgl* $+, \cdot$.
- $[x] \cdot ([y] + [z]) \subseteq [x] \cdot [y] + [x] \cdot [z]$ (Subdistributivität)
  Wählt man eine reelle Zahl x *statt* $[x]$, *gilt sogar Gleichheit.*

Den Beweis findet man z.B. in [3]. Fälle, in denen *echte* Distributivität gilt, werden in [110] und [120] diskutiert.

Überdies lassen sich mit der obigen Definition des Abstands d die Begriffe der Konvergenz und Stetigkeit in $\mathbb{IR}$ einführen und es lässt sich zeigen, dass d eine Metrik (die Hausdorff-Metrik für $\mathbb{IR}$ [54]) und $(\mathbb{IR}, d)$ ein vollständiger metrischer Raum ist [3].

Um wie oben gefordert *garantierte* Einschließungen von Ergebnissen zu berechnen, ist die folgende, per definitionem gegebene Eigenschaft der oben definierten Operationen von besonderer Bedeutung:

Satz 2.1.2 (Inklusionsmonotonie) *Seien* $[x_1] \subseteq [x_2], [y_1] \subseteq [y_2] \in \mathbb{IR}$ *und* $\circ \in \{+, -, \cdot, /\}$. *Existieren* $[x_1] \circ [y_1]$ *und* $[x_2] \circ [y_2]$, *so gilt:*

$$[x_1] \circ [y_1] \subseteq [x_2] \circ [y_2]$$

Einstellige Operationen (insbesondere die so genannten Standardfunktionen) definiert man für Intervalle wie folgt:

Definition 2.1.4 *Die zu* $\phi : \mathbb{R} \to \mathbb{R}, x \mapsto \phi(x)$ *gehörige Intervalloperation wird definiert als*

$$\Phi : \mathbb{IR} \to \mathbb{IR}, [x] \mapsto [\min_{x \in [x]} \phi(x), \max_{x \in [x]} \phi(x)]$$

Zu einer Funktion $f : \mathbb{R} \to \mathbb{R}, x \mapsto f(x)$, die sich aus den oben definierten Operationen zusammensetzt, kann nun durch Ersetzen der einzelnen Operationen durch ihre Intervall-Versionen und Anwendung auf Intervalle die *Intervallerweiterung* $f_{[]}$ einer Funktion gewonnen werden. Der Wertebereich der Intervallerweiterung $f_{[]}([x])$ über einem Intervall $[x]$ enthält dann per definitionem den tatsächlichen Wertebereich der reellen Funktion f über dem Intervall, $f([x]) := \{f(x)|x \in [x]\}$.

Oft wird jedoch der einzuschließende Wertebereich $f([x])$ der reellen Funktion weit überschätzt, so dass eine der zentralen Aufgaben der Intervallarithmetik in der Berechnung möglichst enger Einschließungen für den Wertebereich einer reellen Funktion liegt. Zahllose Arbeiten beschäftigen sich mit dieser Frage, etwa [3, 98, 111].

Zwei Techniken, die in den untersuchten Verfahren Verwendung finden und die Genauigkeit von Wertebereichseinschließungen verbessern sollen, werden in den Abschnitten 2.3 und 2.4 beschrieben.

Schließlich sollen noch *Intervallvektoren* und *-matrizen* definiert werden:

Definition 2.1.5 *Ein* Intervallvektor

$$[x] := ([x_i])_{i=1...n} = ([x_1], ..., [x_n])^T, n \in \mathbb{N}$$

ist ein Vektor, dessen Einträge Intervalle sind.

Eine Intervallmatrix

$$[A] := ([a_{ij}])_{i=1...n, j=1...m}$$

ist eine Matrix, deren Einträge Intervalle sind. Analog zu den reellen Zahlen bezeichnen $\mathbb{IR}^n$ *die Menge der Intervallvektoren der Dimension* n *und* $\mathbb{IR}^{n\times m}$ *die Menge der Intervallmatrizen der Dimension* $n \times m$.

Die für Intervalle definierten Mengenoperatoren $=, \subseteq, \overset{\circ}{\subset}$ werden für Intervallvektoren und -matrizen komponentenweise definiert.

Die arithmetischen Operationen $+, -, \cdot$ lassen sich analog zu den Operationen für reelle Vektoren und Matrizen definieren, wobei die Operationen zur Berechnung der einzelnen Komponenten durch ihre Intervall-Versionen ersetzt werden.

2.2 Rechnerarithmetik

Im vorangehenden Abschnitt wurden wichtige Begriffe der *Intervallarithmetik* eingeführt. Soll diese Arithmetik (oder allgemein *eine* Arithmetik über einer unendlichen Zahlenmenge) auf den Rechner übertragen werden, so stellt sich das Problem, dass ein Rechner in der Praxis nur eine endliche Menge an Zahlen darstellen kann und somit das ursprünglich zu Grunde gelegte Zahlsystem auf dem Rechner nicht darstellbar ist.

Der am weitesten verbreitete Ansatz zur Darstellung reeller Zahlen auf dem Rechner ist die *Gleitkommaarithmetik*. Sie wird heute in konkreter Form nach Maßgabe des *IEEE Standard for Binary Floating-Point Arithmetic* [5] in den meisten Rechnern verwendet.

Definition 2.2.1 *Eine (normalisierte)* Gleitkomma- *oder* Maschinenzahl *ist eine Zahl der Form*

$$x = \pm m \cdot b^e = \pm 0.m_1 m_2 ... m_l \cdot b^e$$

mit einer Mantisse m *der festen Länge* l*, der Basis* b *und dem Exponenten* e*, wobei die Ziffern* m_i *der Mantisse Elemente aus der Menge*

$\{0, ..., b-1\}$ *sind mit* $m_1 \neq 0$ *und für den Exponenten* $e_{min} \leq e \leq e_{max}$, $e, e_{min}, e_{max} \in \mathbb{Z}$ *gilt.*

Hierdurch erhält man ein *Gleitkommasystem* (auch *Gleitkommaraster*)

$$R := R(b, l, e_{min}, e_{max}),$$

wobei die 0 die besondere Darstellung

$$0 = 0.00...0 \cdot b^{e}_{min}$$

erhält.

Als Basis für in der Praxis verwendete Systeme wird meist die *Binärdarstellung*, d.h. ein Gleitkommasystem zur Basis $b = 2$ gewählt, die digital besonders einfach abzubilden ist.

Die Gleitkommaarithmetik nach *IEEE 754* [5] ist ein konkretes Beispiel für ein Gleitkommasystem.

Das vom Rechner verwendete Gleitkommasystem (bzw. das verwendete endliche Zahlsystem) besitzt jedoch nicht die gleiche *Mächtigkeit* (Definition vgl. z.B. [14], S.278) wie die zu Grunde gelegte unendliche Menge, daher können nicht alle in der Ursprungsmenge enthaltenen Zahlen direkt abgebildet werden. Zu diesem Zweck führt man *Rundungen* ein. Mit Rundungen beschäftigt sich z.B. Kulisch in [80, 81], insbesondere mit den mathematischen Eigenschaften des Rechnens auf *Rastern*.

Definition 2.2.2 *Sei* R *ein Gleitkommasystem.*
Eine Abbildung $\bigcirc : \mathbb{R} \rightarrow R$ *heißt* Rundung*, falls gilt:*

$$\bigcirc(x) = x \;\; \forall x \in R \tag{2.1}$$

$$x \leq y \Rightarrow \bigcirc(x) \leq \bigcirc(y) \tag{2.2}$$

Als konkrete Rundungen werden z.B. die Rundung $\square$ zur nächsten darstellbaren Zahl, die Rundung $\bigtriangledown$ zur nächstkleineren darstellbaren Zahl und die Rundung $\bigtriangleup$ zur nächstgrößeren darstellbaren Zahl verwendet.

Ist R ein Gleitkommasystem, so sind die Ergebnisse der reellen Grundoperationen nicht notwendigerweise in R darstellbar; sie werden zu einer

darstellbaren Zahl gerundet. Daher erhält man in einem Gleitkommasystem *gerundete Operationen* $\circledcirc$ für Grundoperationen $\circ \in \{+, -, \cdot, /\}$ und Rundungen $\bigcirc \in \{\square, \triangledown, \triangle\}$. Als Anforderung an eine Implementierung eines Gleitkommasystems wird dabei die Eigenschaft

$$x \circledcirc y = \bigcirc(x \circ y),\ x, y \in R$$

gestellt. Analog wird von einer Implementierung einer Standardfunktion ϕ die Eigenschaft

$$\bigcirc\!\!\!\!\phi\,(x) = \bigcirc(\phi(x))$$

verlangt[3].

Der Begriff des Gleitkommasystems muss nun noch auf Intervalle ausgedehnt werden.

Definition 2.2.3 *Sei R ein Gleitkommasystem.*

$$IR := \{[\underline{x}, \overline{x}] \in \mathbb{IR} | \underline{x}, \overline{x} \in R\}$$

ist die Menge der Gleitkommaintervalle über R. Die Elemente sind Intervalle im üblichen Sinn, d.h. ein Gleitkommaintervall $[x]$ enthält alle reellen *Zahlen x mit $\underline{x} \leq x \leq \overline{x}$.*

Dem Rundungsbegriff für reelle Zahlen entsprechend wird ebenfalls die Rundung für Intervalle definiert:

Definition 2.2.4 *Eine Abbildung*

$$\Diamond : \mathbb{IR} \to IR$$

mit den Eigenschaften

$$\Diamond([x]) = [x] \quad \forall [x] \in IR \tag{2.3}$$

$$[x] \subseteq [y] \Rightarrow \Diamond([x]) \subseteq \Diamond([y]) \tag{2.4}$$

heißt Intervallrundung.

[3] Im Regelfall werden wir später einfach $\circ$ und ϕ für die betreffenden Operationen auch in Bezug auf Gleitkommasysteme schreiben.

Zusätzlich wird für alle konkret verwandten Intervallrundungen gemäß der Bemerkung zu Beginn dieses Abschnitts die Eigenschaft

$$[x] \subseteq \Diamond([x])$$

verlangt. Aufgrund der oben geforderten Eigenschaften folgt mit der Inklusionsmonotonie für reelle Intervalle auch

Satz 2.2.1 (Inklusionsmonotonie) *Für* $[x_1] \subseteq [x_2], [y_1] \subseteq [y_2] \in IR$ *und* $\circ \in \{+, -, \cdot, /\}$ *gilt:*

$$[x_1] \circ [y_1] \subseteq [x_2] \circ [y_2]$$

2.3 Taylorarithmetik

Das Standardvorgehen bei der Auswertung einer Funktion auf einem Intervall ist die oben bereits erwähnte Intervallerweiterung oder *Intervallauswertung*. Ein Weg, um engere Einschließungen des Wertebereichs einer Funktion zu erhalten, ist die *Taylorarithmetik*. Sie beruht auf dem bekannten Satz von Taylor aus der Analysis, der sowohl für Funktionen einer als auch mehrerer Veränderlicher anwendbar ist und in leicht unterschiedlichen Formulierungen in verschiedenen Standardwerken enthalten ist (z.B. [32, 33, 127, 128]).

Hier werden Formulierungen für den ein- und zweidimensionalen Fall angegeben, da diese in dem in [72] vorgeschlagenen Verfahren Verwendung finden, das später genauer betrachtet wird, während die im nachfolgenden Abschnitt betrachteten Rekursionsformeln eindimensional und allgemein mehrdimensional betrachtet werden.

Satz 2.3.1 (Taylor) *Sei* I *ein reelles Intervall und sei* $f : I \to \mathbb{R}$ *eine* $(N+1)$*-mal stetig differenzierbare Funktion,* $N \in \mathbb{N}_0$*. Dann gilt für alle* $x, x_0 \in I$*:*

$$f(x) = T_N(x) + R_N(x)$$

mit

$$T_N(x) := \sum_{k=0}^{N} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \text{ (Taylorpolynom der Ordnung } N\text{)}$$

und dem Restglied $R_N(x)$, das z.B. in der Form von Lagrange *notiert werden kann für eine Zwischenstelle ξ zwischen x und x_0:*

$$R_N(x) := \frac{f^{(N+1)}(\xi)}{(N+1)!}(x - x_0)^{N+1}$$

Satz 2.3.2 (Taylor) *Sei $D \subseteq \mathbb{R}^2$ offen, $(x, y)^T$, $(x_0, y_0)^T \in D$. Sei ferner $f : D \to \mathbb{R}$ $(N + 1)$-mal stetig differenzierbar, $N \in \mathbb{N}_0$, und die Verbindungsstrecke von $(x, y)^T$ nach $(x_0, y_0)^T$ liege in D. Dann gilt:*

$$f(x, y) = \sum_{m=0}^{N} \sum_{n=0}^{N-m} \frac{1}{m!n!} \frac{\partial^{m+n} f}{\partial x^m \partial y^n}(x_0, y_0)(x-x_0)^m(y-y_0)^n + R_N(x, y)$$

mit dem Lagrangeschen Restglied

$$R_N(x, y) := \sum_{n=0}^{N+1} \frac{1}{(N + 1 - n)!n!} \frac{\partial^{N+1} f}{\partial x^n \partial y^{N+1-n}}(\xi, \eta)$$

für eine Zwischenstelle $(\xi, \eta)^T$ zwischen $(x, y)^T$ und $(x_0, y_0)^T$.

Nur bei Einbezug des Restglieds wird sichergestellt, dass bei Verwendung von Intervallkoeffizienten von einer *Einschließung* des Wertebereichs der Funktion gesprochen werden kann.

Mit Hilfe des Satzes von Taylor kann also der Funktionswert einer Funktion f in einer Umgebung des *Entwicklungspunktes* x_0 bestimmt werden. In der Praxis werden mit Hilfe dieser Darstellung Näherungen gewonnen durch Abbrechen der (auch als unendliche Reihendarstellung formulierbaren) Taylorentwicklung beim Grad N und Abschätzung des Restgliedes. Im Sinne des verifzierten Rechnens kann das gleiche Vorgehen unter Verwendung von Intervallen benutzt werden, wobei statt einer Approximation eine Einschließung des Restgliedes berechnet wird.

2.3.1 Funktoide

Mit dem oben beschriebenen Ansatz der Taylorentwicklung erhält man die Möglichkeit, Funktionen $f \in C^N[a,b]$ bzw. $f \in C^N(X), X \subseteq R^n$ in einer Weise darzustellen, die mit dem Rechner handhabbar ist, da für eine Funktion in Taylordarstellung nur der Entwicklungspunkt und die Taylorkoeffizienten zu speichern sind.

Die Taylordarstellungen einer festen Ordnung N spannen einen Unterraum von $C^N(X), X \subseteq R^n$ auf. Vergleichbar mit den Maschinenzahlen können auf diesem Raum Operationen und Rundungen eingeführt werden, so dass das Ergebnis einer Operation in gerundeter Form darstellbar ist.

Im Folgenden werden die wichtigsten Definitionen und Eigenschaften für die Arithmetik mit Taylordarstellungen in Funktionenräumen für den eindimensionalen Fall angegeben. [66] gibt eine ausführlichere Darstellung funktionaler Probleme.

Für die folgenden Betrachtungen sei $\mathcal{M} := C[a,b]$.

Definition 2.3.1 *$S_N(\mathcal{M})$ sei ein $(N+1)$-dimensionaler Unterraum in $\mathcal{M}$. Dieser besitze die Basis $\{\phi_0, ..., \phi_N\}$*[4].

Dann heißt $S_N(\mathcal{M})$ Raster *von $\mathcal{M}$.*

Eine Abbildung $S_N : \mathcal{M} \to S_N(\mathcal{M})$ mit $S_N(f) = f$ für alle $f \in S_N(\mathcal{M})$ heißt Rundung.

Definition 2.3.2 *Sind für $f, g \in S_N(\mathcal{M})$ Operationen*

$$f \circ g := S_N(f \circ g), \circ \in \{+, -, \cdot, /\}$$

und

$$\oint f := S_N(\int f)$$

[4] z.B. die Monombasis $\{(x-x_0)^i, 0 \leq i \leq N\}, x_0 \in \mathbb{R}$ Entwicklungspunkt. Hierdurch sieht man auch, dass man bei Wahl unterschiedlicher Entwicklungspunkte unterschiedliche Basen des aufgespannten Raumes erhält.

definiert[5], so nennt man

$$(S_N, \mathcal{O} := \{+, -, \cdot, /, \int\})$$

Funktoid.

Ist $\{\phi_0, ..., \phi_N\}$ wie oben Basis von $S_N(\mathcal{M})$, so wird durch den Isomorphismus

$$\iota : S_N(\mathcal{M}) \to \mathbb{R}^{N+1}, \sum_{i=0}^{N} a_i \phi_i \mapsto (a_0, ..., a_N)$$

die *adjungierte Rundung*

$$R_N : \mathcal{M} \to \mathbb{R}^N := S_N(\iota)$$

induziert. $S_N(\mathcal{M})$ ist mit den dargestellten Operationen ein normierter Raum [66], da auch $\mathcal{M}$ ein normierter Raum ist.

Analog zu den obigen reellen Definitionen kann durch Übergang zur Potenzmenge PM auch der Begriff *Intervallfunktoid* definiert werden. Dazu werden passende Operatoren betrachtet und wie oben die Begriffe *Rundung* und *adjungierte Rundung* erklärt. Die entsprechenden Definitionen werden in [72] aufgeführt.

2.3.2 Rekursive Berechnung von Taylorkoeffizienten

Die entscheidende Eigenschaft des Tayloransatzes ist, dass mit Objekten in dieser Funktionsdarstellung eine eigene Arithmetik definiert werden kann, die nicht nur die *Darstellung* von Funktionen, sondern auch das Rechnen mit den so gewonnenen Objekten erlaubt, so wie es bei der Lösung von Integralgleichungen notwendig ist (siehe Abschnitt 2.5).

Gleichzeitig ist die Taylorarithmetik auch von Interesse, da mit ihrer Hilfe Ableitungen von Funktionen automatisch berechnet werden können.

[5] Auf eine Darstellung der Operationen durch eigenständige Symbole wird verzichtet.

Die *automatische Differentiation* ist daher Gegenstand vielfältiger Forschungsaktivitäten (siehe z.B. [40, 41, 42, 108]).

Rekursionsformeln zur Berechnung von Taylorkoeffizienten werden in [11, 30, 67] angegeben und sollen hier kurz dargestellt werden, um im nachfolgenden Abschnitt neue zusätzliche Rekursionsformeln zu entwickeln.

Da die Taylorkoeffizienten einer Funktion im Wesentlichen aus den Ableitungen (im Mehrdimensionalen: partiellen Ableitungen) der betrachteten Funktionen gebildet werden, folgt ihre Bildung den Differentiationsregeln für reelle Funktionen in $\mathbb{R}$ bzw. $\mathbb{R}^n$.

Zunächst wird der eindimensionale Fall betrachtet.

Definition 2.3.3 *Sei eine Funktion $f : D \subseteq \mathbb{R} \to \mathbb{R}$ ausreichend oft differenzierbar (so dass die unten verwendeten Ableitungen existieren) und $x_0 \in D$. Dann wird der k*-te Taylorkoeffizient *von f definiert als*

$$(f)_k := \frac{1}{k!} f^{(k)}(x_0) := \frac{1}{k!} \frac{\partial^k f}{\partial x^k}(x_0), k = 0, 1, ...$$

Bemerkung 2.3.1 *Mit der obigen Definition gilt offensichtlich*

$$\begin{aligned} (f)_0 &= f(x_0) & (2.5)\\ (f)_1 &= f'(x_0) & (2.6)\\ k!(f)_k &= f^{(k)}(x_0) & (2.7)\\ (f)_k &= \frac{1}{k}(f')_{k-1},\ k > 0 & (2.8)\\ T_N(x) &= \sum_{k=0}^{N} (f)_k (x - x_0)^k & (2.9) \end{aligned}$$

Bemerkung 2.3.2 *Für $\mathrm{x} := f : x \mapsto x$ (Identität) bzw. $\mathrm{c} := f : x \mapsto c$ (konstante Funktion), $c \in \mathbb{R}$ gilt*

$$(\mathrm{x})_0 = x_0, (\mathrm{x})_1 = 1, (\mathrm{x})_k = 0, k > 1. \quad (2.10)$$

$$(\mathrm{c})_0 = c, (\mathrm{c})_k = 0, k > 0. \quad (2.11)$$

Aus den Differentiationsregeln für reelle Funktionen folgt (vgl. z.B. auch [11])

Satz 2.3.3 (Grundoperationen) *Sei* $k \geq 0$*,* $f, g : D \subseteq \mathbb{R} \to \mathbb{R}$ *ausreichend oft differenzierbar (so dass die benötigten Ableitungen gemäß der reellen Differentiationsregeln existieren). Dann gilt:*

$$(-f)_k = -(f)_k \tag{2.12}$$

$$(f \pm g)_k = (f)_k \pm (g)_k \tag{2.13}$$

$$(f \cdot g)_k = \sum_{j=0}^{k} (f)_j (g)_{k-j} \tag{2.14}$$

$$(f/g)_k = \frac{1}{(g)_0}\left((f)_k - \sum_{j=0}^{k-1} (f/g)_j (g)_{k-j}\right), (g)_0 \neq 0 \tag{2.15}$$

Mit Hilfe der Differentiationsregeln für reelle Funktionen und den oben genannten Beziehungen lassen sich überdies Rekursionsformeln für die Standardfunktionen angeben. Als Beispiel hierfür sei die Exponentialfunktion genannt, weitere Rekursionsformeln finden sich z.B. in [11, 30] und gehen z.T. auf [108] zurück.

Satz 2.3.4 *Für die Taylorkoeffizienten der Exponentialfunktion* exp *gilt für* $k > 0$ *und eine Funktion* f*, für die die angegeben Ausdrücke wohldefiniert sind:*

$$(\exp(f))_0 = \exp((f)_0) \tag{2.16}$$

$$(\exp(f))_k = \sum_{j=0}^{k-1} (1 - \frac{j}{k})(\exp(f))_j (f)_{k-j} \tag{2.17}$$

Entsprechende Aussagen können auch für mehrdimensionale Taylorkoeffizienten getroffen werden.

Definition 2.3.4 *Sei eine Funktion* $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ *ausreichend oft differenzierbar (so dass die unten verwendeten partiellen Ableitungen exi-*

stieren) und $x_0 \in D$. *Sei ferner* $k = (k_1...k_n), k_i \geq 0, i = 1...n$ *ein* Multiindex *mit* $|k| := \sum_{i=1}^{n} k_i$.

Dann wird der k-te Taylorkoeffizient von f *definiert als*

$$(f)_k := (f)_{k_1...k_n} := \frac{1}{k_1! \cdot ... \cdot k_n!} \frac{\partial^{|k|} f}{\partial x_1^{k_1} ... \partial x_n^{k_n}}(x_0).$$

Sind die obigen Ableitungen von f in x_0 stetig, so kann die Reihenfolge der partiellen Ableitungen vertauscht werden, ohne dass der entstehende Ausdruck sich ändert (Beweis siehe z.B. [33]).

Bemerkung 2.3.3 *Insbesondere gilt*

$$(f)_k = (f)_{k_1...k_n} = (...(((f)_{k_1\ 0...0})_{0\ k_2\ 0...0})...)_{0...0\ k_n} \tag{2.18}$$

oder kürzer

$$(f)_{k_1...k_n} = (...(((f)_{k_1})_{k_2})...)_{k_n}. \tag{2.19}$$

Satz 2.3.5 (Grundoperationen) *Sei* k *wie oben ein Multiindex und seien* $f, g : D \subseteq \mathbb{R}^n \to \mathbb{R}$ *ausreichend oft differenzierbar (so dass die benötigten partiellen Ableitungen gemäß der reellen Differentiationsregeln existieren). Dann gilt:*

$$(-f)_k = -(f)_k \tag{2.20}$$

$$(f \pm g)_k = (f)_k \pm (g)_k \tag{2.21}$$

$$(f \cdot g)_k = \sum_{j_1=0}^{k_1} ... \sum_{j_n=0}^{k_n} (f)_{j_1...j_n} (g)_{k_1-j_1...k_n-j_n} \tag{2.22}$$

$$\left(\frac{f}{g}\right)_k = \frac{(f)_k - \sum\limits_{j_1=0}^{k_1} ... \sum\limits_{j_n=0}^{k_n} (g)_{j_1...j_n} (\frac{f}{g})_{k_1-j_1...k_n-j_n}}{(g)_0} \tag{2.23}$$

falls $(g)_0 \neq 0$.

Beweis: siehe z.B. [11] □

Ebenso lassen sich Rekursionsformeln für Standardfunktionen angeben, die z.B. in [11, 30] detaillierter aufgeführt werden.

Bemerkung 2.3.4 *Indem man Funktionen über Intervallen betrachtet, kann gemäß der Aussagen in Abschnitt 2.1 eine den vorstehenden Aussagen entsprechende Taylorarithmetik auch auf Intervalle angewandt werden. Als* Entwicklungspunkt x_0 *der Taylorentwicklung kann dabei der Mittelpunkt des betrachteten Intervalls dienen. Da das Lagrangesche Restglied die Auswertung der betrachteten Funktion f an einer Zwischenstelle erfordert, wird hier das ganze Definitionsintervall eingesetzt, um alle möglichen Zwischenstellen einzuschließen [87]. Hier zeigt sich, dass die Taylorentwicklung in der Regel umso bessere Einschließungen des Wertebereichs annimmt, je kleiner die zu Grunde liegenden Intervalle sind. Untersuchungen hierzu wurden auch in [68] vorgenommen.*

Für die praktische Anwendung der Rekursionsformeln ist von Interesse, dass sich die betrachteten Funktionen mittels der obigen Formeln rekursiv auf Funktionen mit bekannten Taylorkoeffizienten zurückführen lassen. In [11] wird hierzu z.B. der Begriff der *Faktorisierbarkeit* definiert.

Definition 2.3.5

$$\mathcal{OP} := \{-, +, -, \cdot, /\}$$

bezeichne die Menge der unären und binären elementaren (reellen) Operationen,

$$\begin{aligned} \mathcal{SF} \quad := \quad & \{\mathrm{sqr}, \sqrt{\ }, \mathrm{power}, \exp, \ln, \\ & \sin, \cos, \tan, \cot, \\ & \arcsin, \arccos, \arctan, \mathrm{arccot}, \\ & \sinh, \cosh, \tanh, \coth, \\ & \mathrm{arsinh}, \mathrm{arcosh}, \mathrm{artanh}, \mathrm{arcoth}\} \end{aligned}$$

bezeichne die Menge der (reellen) Standardfunktionen.

Sei $\pi_i^n : \mathbb{R}^n \rightarrow \mathbb{R}$ die Projektion eines reellen n-Tupels auf seine i-te Komponente, $i \leq n \in \mathbb{N}$.

Eine Funktion $f : D \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$ heißt faktorisierbar, *falls es eine endliche Folge von Funktionen $f_1,...,f_k$, $k \geq n \in \mathbb{N}$, $f_j : D \rightarrow \mathbb{R}$ für $j = 1...k$ gibt, so dass gilt:*

1. $f = f_k$

2. $f_i = \pi_i^n$, $i = 1...n$

3. *Für* $n < j \leq k$*:*

   - $f_j = g(f_l)$ *mit* $g \in \mathcal{SF}$ *und* $l < j$ *oder*
   - $f_j = f_l \circ f_m, l, m < j$ *mit einem binären Operator* $\circ \in \mathcal{OP}$ *oder*
   - $f_j = -f_l, l < j$ *oder*
   - $f_j \equiv c \in \mathbb{R}$ *konstant.*

So lassen sich für alle Funktionen, die sich in der definierten Weise zusammensetzen lassen, auch in der Praxis die Taylorkoeffizienten rekursiv berechnen.

2.3.3 Neue Rekursionsformeln

Im vorangehenden Abschnitt wurde die rekursive Bestimmung von Taylorkoeffizienten vorgestellt. Es sollen nun zusätzliche Rekursionsformeln für die Funktionen

$$\mathrm{erf} : \mathbb{R} \to \mathbb{R}, x \mapsto \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)dt \quad \textit{(Gaußsche Fehlerfunktion)}$$

und

$$\mathrm{erfc} : \mathbb{R} \to \mathbb{R}, x \mapsto 1 - \mathrm{erf}(x) \quad \textit{(Komplementäre Fehlerfunktion)}$$

angegeben werden.

Die *Gaußsche Fehlerfunktion* ist eine Formulierung der *Verteilungsfunktion* der *Standardnormalverteilung* (vgl. z.B. [14, 114]). Als solche ist sie von zentralem Interesse in vielen Betrachtungen der Statistik bzw. Stochastik. Eine Betrachtung der Fehlerfunktion im Rahmen der Intervallarithmetik findet sich in [77].

Zunächst sollen Rekursionsformeln für die eindimensionale Taylorarithmetik angegeben werden.

Satz 2.3.6 *Sei eine Funktion* $f : D \subseteq \mathbb{R} \to \mathbb{R}$ *ausreichend oft differenzierbar (s.o.) und* erf *und* erfc *die* Gaußsche Fehlerfunktion *sowie die* Komplementäre Fehlerfunktion. *Für die eindimensionalen Taylorkoeffizienten* $(\mathrm{erf}(f))_k$ *und* $(\mathrm{erfc}(f))_k, k > 0$ *gelten die Rekursionsformeln*

$$(\mathrm{erf}(f))_0 = \mathrm{erf}((f)_0) \tag{2.24}$$

$$(\mathrm{erf}(f))_k = \frac{2}{k\sqrt{\pi}} \sum_{i=0}^{k-1} (k-i)(\exp(-f^2))_i (f)_{k-i} \tag{2.25}$$

$$(\mathrm{erfc}(f))_k = -\frac{2}{k\sqrt{\pi}} \sum_{i=0}^{k-1} (k-i)(\exp(-f^2))_i (f)_{k-i} \tag{2.26}$$

Beweis: Es gilt

$$\begin{aligned}
(\mathrm{erf}(f))_k &= \frac{1}{k}(\mathrm{erf}(f)')_{k-1} \\
&= \frac{1}{k}(\frac{2}{\sqrt{\pi}} \exp(-f^2) \cdot f')_{k-1} \\
&= \frac{2}{k\sqrt{\pi}} \sum_{i=0}^{k-1} (\exp(-f^2))_i (f')_{k-1-i} \\
&= \frac{2}{k\sqrt{\pi}} \sum_{i=0}^{k-1} (k-i)(\exp(-f^2))_i (f)_{k-i}
\end{aligned}$$

Dabei gehen die üblichen Differentiationsregeln sowie (2.8) und (2.14) ein. Mit (2.11) und (2.13) folgt (2.26) sofort aus (2.25). □

Der entsprechende Satz für die mehrdimensionale Taylorarithmetik wird im Folgenden formuliert.

Satz 2.3.7 *Sei eine Funktion* $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ *ausreichend oft differenzierbar (s.o.) und* erf *und* erfc *die* Gaußsche Fehlerfunktion *sowie die* Komplementäre Fehlerfunktion. *Sei ferner* $k = (k_1 ... k_n)$, $k_i \geq 0$, $i = 1...n$ *ein Multiindex. Für die* n*-dimensionalen Taylorkoeffizienten*

$(\mathrm{erf}(f))_k$, $(\mathrm{erfc}(f))_k$, $k_1 > 0$ *gilt*

$$(\mathrm{erf}(f))_k = \frac{2}{k_1\sqrt{\pi}} \sum_{i_1=0}^{k_1-1} \sum_{i_2=0}^{k_2} ... \sum_{i_n=0}^{k_n} \Big((k_1 - i_1) \cdot (\exp(-f^2))_{i_1 ... i_n} (f)_{k_1-i_1 ... k_n-i_n} \Big) \quad (2.27)$$

$$(\mathrm{erfc}(f))_k = -\frac{2}{k_1\sqrt{\pi}} \sum_{i_1=0}^{k_1-1} \sum_{i_2=0}^{k_2} ... \sum_{i_n=0}^{k_n} \Big((k_1 - i_1) \cdot (\exp(-f^2))_{i_1 ... i_n} (f)_{k_1-i_1 ... k_n-i_n} \Big) \quad (2.28)$$

Beweis: Es gilt

$$(\mathrm{erf}(f))_k = (...(((\mathrm{erf}(f))_{k_1\ 0...0})_{0\ k_2\ 0...0})...)_{0...0\ k_n}$$

$$\begin{aligned}
&= \frac{2}{k_1\sqrt{\pi}} \sum_{i_1=0}^{k_1-1} (k_1 - i_1) \Big(... \Big((\exp(-f^2))_{i_1 0...0} \cdot (f)_{k_1-i_1 0...0} \Big)_{0 k_2 0...0} ... \Big)_{0...0 k_n} \\
&= \frac{2}{k_1\sqrt{\pi}} \sum_{i_1=0}^{k_1-1} (k_1 - i_1) \cdot \left[\sum_{i_2=0}^{k_2} ... \sum_{i_n=0}^{k_n} (\exp(-f^2))_{i_1 ... i_n} (f)_{k_1-i_1 ... k_n-i_n} \right] \\
&= \frac{2}{k_1\sqrt{\pi}} \sum_{i_1=0}^{k_1-1} \sum_{i_2=0}^{k_2} ... \sum_{i_n=0}^{k_n} \Big((k_1 - i_1) \cdot (\exp(-f^2))_{i_1 ... i_n} (f)_{k_1-i_1 ... k_n-i_n} \Big)
\end{aligned}$$

Dabei gehen (2.22) und Satz 2.3.7 ein. Mit (2.21) und dem mehrdimensionalen Äquivalent zu (2.11) folgt dann (2.28) aus (2.27). □

Für den Fall der zweidimensionalen Taylorarithmetik schreibt sich (2.27)

als

$$(\mathrm{erf}(f))_{k_1 k_2} = \frac{2}{k_1\sqrt{\pi}} \sum_{i_1=0}^{k_1-1} \sum_{i_2=0}^{k_2} \Big((k_1 - i_1) \cdot (\exp(-f^2))_{i_1 i_2} (f)_{k_1-i_1 k_2-i_2} \Big) \quad (2.29)$$

In Kapitel 5 werden Implementierungen der Taylorarithmetik in *C-XSC* (siehe Abschnitt 5.1.1) vorgestellt, die die angegebenen Rekursionsformeln umsetzen.

2.4 Steigungsarithmetik

Bisher wurden die *Intervallauswertung* und die *Taylorarithmetik* als Vorgehensweisen zum Einschließen des Wertebereichs einer Funktion betrachtet. Eine weitere Möglichkeit besteht in der *Steigungsarithmetik*, die in dem als Vergleichsverfahren untersuchten Verfahren aus [106] Verwendung findet.

Taylorarithmetik und Steigungsarithmetik eint die Eigenschaft, dass Taylorkoeffizienten und Steigungen in vergleichbarer Weise automatisch berechnet werden können [12, 112]. Im Gegensatz zur Taylorarithmetik ist die Steigungsarithmetik jedoch auch auf nicht differenzierbare Funktionen anwendbar [12]. Die Steigungsarithmetik liefert in der Regel, ebenso wie die Taylorarithmetik für genügend enge Intervalle, engere Einschließungen als die Intervallauswertung [112] und ist daher zur Verbesserung von Wertebereichseinschließungen von Interesse.

Ebenso wie die Taylorarithmetik lässt sich auch die Steigungsarithmetik für den eindimensionalen oder mehrdimensionalen Fall in ähnlicher Weise formulieren. Im Folgenden wird der mehrdimensionale Fall kurz vorgestellt.

Definition 2.4.1 *Sei $f : D \subseteq R^n \to \mathbb{R}$ gegeben. Eine Funktion*

$$s_f : D \times D \to \mathbb{R}^n$$

heißt Steigung *von f bzgl.* $(x, c), x, c \in D$*, falls gilt:*

$$f(x) = f(c) + s_f(x, c)(x - c)$$

Ist $[f_s]$ eine Einschließung von

$$s_f([x], [c]) := \{s_f(x, c) | x \in [x] \subseteq D, c \in [c] \subseteq D, x \neq c\},$$

so gilt für $x \in [x]$

$$\begin{aligned} f(x) &= f(c) + s_f(x, c)(x - c) && (2.30) \\ &\in f(c) + [f_s](x - c) && (2.31) \\ &\subseteq f([c]) + [f_s]([x] - [c]) && (2.32) \end{aligned}$$

So erhält man eine Einschließung der Funktion f über dem Intervall $[x]$.

Bemerkung 2.4.1 *Steigungen existieren neben differenzierbaren Funktionen auch für die Funktionen* abs, min, max *und* χ *mit*

$$\chi(x, y_1, y_2) := \begin{cases} y_1, x < 0 \\ y_2, x \geq 0 \end{cases} \quad \text{(Verzweigungsfunktion)}$$

Zur Beschreibung einer Arithmetik mit Steigungen wird der Begriff des *Steigungstripels* verwendet. Ohne Beschränkung der Allgemeinheit sei im Folgenden jeweils $[c] \subseteq [x]$.

Definition 2.4.2 *Seien* $f : D \subseteq R^n \to \mathbb{R}$, $[x]$, $[c] \subseteq D$*. Ein Tripel* (f_x, f_c, f_s) *mit* $f_x, f_c \in \mathbb{IR}$, $f_s := (f_s^i)_{1 \leq i \leq n} \in \mathbb{IR}^n$ *heißt* Steigungstripel *bezüglich* f, $[x]$ *und* $[c]$*, falls gilt:*

$$\begin{aligned} \forall x \in [x] \quad & f(x) \in f_x && (2.33) \\ \forall c \in [c] \quad & f(c) \in f_c && (2.34) \\ \forall x \in [x], c \in [c]\ \exists s \in f_s \quad & f(x) = f(c) + s(x - c) && (2.35) \end{aligned}$$

Wie für die Taylorarithmetik können jetzt Regeln zur Berechnung von Steigungstripeln angegeben werden.

Satz 2.4.1 *Für* $h_1 : \mathbb{R}^n \to \mathbb{R}, x \mapsto \lambda, \lambda \in \mathbb{R}$, $h_2 : \mathbb{R}^n \to \mathbb{R}, x \mapsto x_i$ *und die Einheitsvektoren* $e_i := (0, ..., 0, \underbrace{1}_{i-te\,Stelle}, 0, ..., 0)^T$ *gilt:*

$$(\lambda, \lambda, 0) \quad \textit{und} \quad ([x]_i, [c]_i, e_i)$$

sind die Steigungstripel von h_1 *und* h_2 *bzgl. Intervallvektoren* $[x], [c]$.

Satz 2.4.2 *Seien* $f, g : D \in \mathbb{R}^n \to \mathbb{R}$ *mit Steigungstripeln* (f_x, f_c, f_s), (g_x, g_c, g_s). *Dann gilt für das Steigungstripel* (h_x, h_c, h_s) *der Funktion* $h := f \circ g, \circ \in \{+, -, \cdot, /\}$*:*

$$\begin{aligned} h_x &= f_x \circ g_x \\ h_c &= f_c \circ g_c \\ h_s &= \begin{cases} f_s \circ g_s & \textit{falls } \circ \in \{+, -\} \\ g_x \cdot f_s + f_c \cdot g_s & \textit{falls } \circ = \cdot \\ (f_s - h_c \cdot g_s)/g_x & \textit{falls } \circ = /, 0 \notin g_x \end{cases} \end{aligned} \tag{2.36}$$

Satz 2.4.3 *Seien* $f : D \in \mathbb{R}^n \to \mathbb{R}$, $\phi : \mathbb{R} \to \mathbb{R}$ *mit Steigungstripel* (f_x, f_c, f_s) *für* f. *Ferner existiere* $\phi(f_x)$ *(Intervallauswertung) und* $E_\phi(f_x, f_c)$ *sei eine Einschließung von* $s_\phi(f_x, f_c)$. *Dann gilt für das Steigungstripel* (h_x, h_c, h_s) *von* $h := \phi(f)$*:*

$$\begin{aligned} h_x &= \phi(f_x) \\ h_c &= \phi(f_c) \\ h_s &= E_\phi(f_x, f_c) \cdot f_s \end{aligned} \tag{2.37}$$

Die Beweise dieser Regeln ebenso wie Regeln für die erwähnten nicht differenzierbaren Funktionen finden sich in [112].

Für die praktische Anwendung der Steigungsarithmetik lassen sich wie bei der Taylorarithmetik *faktorisierbare* Funktionen definieren (siehe Definition 2.3.5).

Die Güte der obigen Arithmetik hängt entscheidend davon ab, wie eng die Einschließungen $E_\phi(f_x, f_c)$ von $s_\phi(f_x, f_c)$ sind, die in Satz 2.4.3 verwendet werden. Für viele Funktionen lassen sich enge Einschließungen finden (vgl. z.B. [112]).

Implementierungen der Steigungsarithmetik werden z.B. in [11, 12] angegeben.

2.5 Integralgleichungen

In diesem Abschnitt soll der in dieser Arbeit zentrale Begriff der *Integralgleichung* erklärt werden.

Eine *Integralgleichung* ist eine Gleichung, in der die Unbekannte, eine Funktion y, unter dem Integral auftritt.

Integralgleichungen können nach verschiedenen Kriterien klassifiziert werden [17, 28]. Für die nachfolgenden Definitionen seien stetige Funktionen $\alpha, g, y : [a, b] \subset \mathbb{R} \to \mathbb{R}$, $k : [a, b] \times [a, b] \to \mathbb{R}$, $\lambda \in \mathbb{R}$ gegeben (y ist die gesuchte Unbekannte).

Definition 2.5.1 *Seien α, g, y, k, λ wie oben. Die Gleichung*

$$\alpha(s)y(s) - \lambda \int_a^b k(s,t)y(t)dt = g(s) \tag{2.38}$$

heißt

- lineare Fredholmsche Integralgleichung erster Art*, wenn $\alpha = 0$ gilt.*
- lineare Fredholmsche Integralgleichung zweiter Art*, wenn $\alpha = 1$ gilt.*
- lineare Fredholmsche Integralgleichung dritter Art*, wenn $\alpha \neq 0, 1$ gilt.*

Definition 2.5.2 *Seien α, g, y, k, λ wie oben. Die Gleichung*

$$\alpha(s)y(s) - \lambda \int_a^s k(s,t)y(t)dt = g(s) \tag{2.39}$$

heißt

- lineare Volterrasche Integralgleichung erster Art*, wenn $\alpha = 0$ gilt.*

- lineare Volterrasche Integralgleichung zweiter Art*, wenn* $\alpha = 1$ *gilt.*
- lineare Volterrasche Integralgleichung dritter Art*, wenn* $\alpha \neq 0, 1$ *gilt.*

Definition 2.5.3 *Seien* α, g, y, λ *wie oben,* $k : [a, b] \times [a, b] \times \mathbb{R} \to \mathbb{R}$ *stetig. Die Gleichung*

$$\alpha(s)y(s) - \lambda \int_a^b k(s, t, y(t))dt = g(s) \tag{2.40}$$

heißt nichtlineare Fredholmsche Integralgleichung*, wobei ansonsten die gleichen Klassifikationen wie in Definition 2.5.1 gelten. Gleiches gilt für Volterrasche Integralgleichungen.*

Definition 2.5.4 *Seien* α, g, y, λ *wie oben,* $k : [a, b] \times [a, b] \times \mathbb{R} \to \mathbb{R}$ *stetig. Die Integralgleichung*

$$y(s) - \int_a^b k(s, t, y(t))dt = g(s) \tag{2.41}$$

heißt Urysohn-Gleichung.

Definition 2.5.5 *Seien* α, y, λ *wie oben,* $k : [a, b] \times [a, b] \times \mathbb{R} \to \mathbb{R}$ *und* $g : [a, b] \times \mathbb{R} \to \mathbb{R}$ *stetig. Die Integralgleichung*

$$y(s) - \int_a^b k(s, t)g(t, y(t))dt = g(s) \tag{2.42}$$

heißt Hammerstein-Gleichung.

Definition 2.5.6 *k wie oben heißt* Kern *der Integralgleichung, g wie oben heißt* rechte Seite *oder* Inhomogenität *der Integralgleichung. Für* $g = 0$ *heißt die Integralgleichung* homogen*, für* $g \neq 0$ inhomogen.

Die Integralgleichungen der vorstehenden Definitionen können noch kürzer formuliert werden, wenn die Funktionen als Elemente eines Funktionenraums und das Integral als Operator aufgefasst werden.

Definition 2.5.7 *Eine Abbildung $L : X \to Y$, X, Y Banachräume, heißt* linearer Operator*, falls gilt:*

$$L(\alpha x + \beta y) = \alpha L(x) + \beta L(y),\ x, y \in X, \alpha \in \mathbb{R}.$$

Der Operator $I : X \to X, x \mapsto x$ heißt Identitätsoperator.

Definition 2.5.8 *Die Abbildung*

$$\begin{aligned} K: \quad & C[a,b] \to C[a,b], \\ & x \mapsto Kx \end{aligned}$$

mit

$$(Kx)(s) := \int_a^b k(s,t)x(t)dt$$

heißt Fredholmscher Integraloperator.

Per definitionem ist ein Fredholmscher Integraloperator damit ein linearer Operator.

Der Raum der linearen Operatoren $L : X \to Y$, X, Y Banachräume, wird mit

$$\mathcal{L}(X, Y)$$

bezeichnet. Mit der Norm

$$\|L\| := \sup_{\|x\|=1} \|Lx\|, L \in \mathcal{L}(X, X)$$

wird $\mathcal{L}(X, X)$ für $X := C^N[a,b]$ zum Banachraum [90], da $C[a,b]$ Banachraum ist [56].

Die lineare Fredholmsche Integralgleichung zweiter Art

$$y(s) - \lambda \int_a^b k(s,t)y(t)dt = g(s) \tag{2.43}$$

lässt sich somit schreiben als

$$(I - \lambda K)y = g. \tag{2.44}$$

Zur Betrachtung der Lösbarkeit linearer Fredholmscher Integralgleichungen zweiter Art sei vorausgesetzt, dass der Kern k und die Inhomogenität g der Integralgleichung stückweise stetig über ihren Definitionsbereichen sind[6].

Für die praktische Implementierung mit Funktionen in Taylordarstellung der Ordnung n wird die n-malige stetige Differenzierbarkeit über $[a, b] \times [a, b]$ bzw. $[a, b]$ verlangt.

Im Folgenden wird die Integralgleichung (2.43) betrachtet.

2.5.1 Integralgleichungen mit entartetem Kern

Definition 2.5.9 *Hat der Kern k der Integralgleichung eine Darstellung*

$$k(s,t) = \sum_{i=1}^{N} a_i(s) b_i(t) \tag{2.45}$$

mit stetigen Funktionen $a_i, b_i : [a, b] \to \mathbb{R}, i = 1...N$, so dass die a_i und b_i für $i = 1...N$ jeweils linear unabhängig sind, so wird er als entartet *vom Grad N bezeichnet.*

Beispiel 2.5.1 *Der Kern*

$$k : [0,1] \times [0,1] \to \mathbb{R}, (s,t) \mapsto 2st + 1$$

ist entartet vom Grad 2, denn mit

$$\begin{aligned} a_1 &: [0,1] \to \mathbb{R}, s \mapsto 2s \\ b_1 &: [0,1] \to \mathbb{R}, t \mapsto t \\ a_2 &: [0,1] \to \mathbb{R}, s \mapsto 1 \\ b_2 &: [0,1] \to \mathbb{R}, t \mapsto 1 \end{aligned}$$

[6]Es werden also Integrale mit Integrationsbereichen betrachtet, auf denen die Funktion stetig ist, so wie es bei der Betrachtung von Integralgleichungssystemen der Fall ist

gilt

$$k(s,t) = 2st + 1 = \sum_{i=1}^{2} a_i(s)b_i(t)$$

und $\{a_1, a_2\}$ *und* $\{b_1, b_2\}$ *sind offensichtlich linear unabhängig in C[0,1].*

Beispiel 2.5.2 *Der Kern*

$$k : [0,1] \times [0,1] \to \mathbb{R}, (s,t) \mapsto \sin(st)$$

ist nicht entartet - es lässt sich keine Darstellung

$$k(s,t) = \sum_{i=1}^{N} a_i(s)b_i(t),\ N \in \mathbb{N}$$

finden. Stellt man $k(s,t)$ *als Taylorentwicklung der Ordnung* 2 *am Entwicklungspunkt* $(\frac{1}{2}, \frac{1}{2})$ *dar, erhält man*

$$\begin{aligned} k(s,t) \quad = \quad & \sin(\tfrac{1}{4}) \\ & + \tfrac{1}{2}\cos(\tfrac{1}{4})(s - \tfrac{1}{2}) \\ & + \tfrac{1}{2}\cos(\tfrac{1}{4})(t - \tfrac{1}{2}) \\ & - \tfrac{1}{8}\sin(\tfrac{1}{4})(s - \tfrac{1}{2})^2 \\ & + (\cos(\tfrac{1}{4}) - \tfrac{1}{4}\sin(\tfrac{1}{4}))(t - \tfrac{1}{2})(s - \tfrac{1}{2}) \\ & - \tfrac{1}{8}\sin(\tfrac{1}{4})(t - \tfrac{1}{2})^2 \\ & + R(s,t) \end{aligned}$$

mit dem Restglied $R(s,t)$*. Mit*

$$\begin{aligned} & a_1 : [0,1] \to \mathbb{R}, s \mapsto 1 \\ & b_1 : [0,1] \to \mathbb{R}, t \mapsto \sin(\tfrac{1}{4}) + \tfrac{1}{2}\cos(\tfrac{1}{4})(t - \tfrac{1}{2}) - \tfrac{1}{8}\sin(\tfrac{1}{4})(t - \tfrac{1}{2})^2 \\ & a_2 : [0,1] \to \mathbb{R}, s \mapsto s - \tfrac{1}{2} \\ & b_2 : [0,1] \to \mathbb{R}, t \mapsto \tfrac{1}{2}\cos(\tfrac{1}{4}) + (\cos(\tfrac{1}{4}) - \tfrac{1}{4}\sin(\tfrac{1}{4}))(t - \tfrac{1}{2}) \\ & a_3 : [0,1] \to \mathbb{R}, s \mapsto (s - \tfrac{1}{2})^2 \\ & b_3 : [0,1] \to \mathbb{R}, t \mapsto -\tfrac{1}{8}\sin(\tfrac{1}{4}) \end{aligned}$$

gilt aber

$$k(s,t) = k_{\mathfrak{E}}(s,t) + k_{\mathfrak{N}}(s,t)$$

mit einem entarteten Kern $k_{\mathfrak{E}}$, der gegeben ist durch[7]

$$k_{\mathfrak{E}}(s,t) := \sum_{i=1}^{3} a_i(s) b_i(t).$$

Der Kern lässt sich also darstellen durch einen entarteten Anteil und einen nicht entarteten Rest.

Hat die Integralgleichung (2.43) einen entarteten Kern vom Grad N, so lässt sie sich wie folgt darstellen:

$$y(s) - \lambda \sum_{i=1}^{N} a_i(s) \int_a^b b_i(t) y(t) dt = g(s). \tag{2.46}$$

Zum Lösungsverhalten linearer Fredholmscher Integralgleichungen zweiter Art mit entartetem Kern gilt der folgende Satz:

Satz 2.5.1 *Die Integralgleichung (2.43) habe einen entarteten Kern vom Grad N, d.h. sie lasse sich darstellen wie in (2.46), $\lambda \neq 0$. Dann hat jede Lösung der Integralgleichung die Form*

$$y(s) = g(s) + \lambda \sum_{i=1}^{N} a_i(s) \xi_i(s) \tag{2.47}$$

für geeignete $\xi_i(s), i = 1 \ldots N$, und (2.47) ist Lösung der Integralgleichung genau dann, wenn $(\xi_i(s))_{i=1\ldots N}$ Lösung des folgenden linearen Gleichungssystems ist:

$$\xi_i(s) - \lambda \sum_{j=1}^{N} \int_a^b b_i(t) a_j(t) dt \, \xi_j(s) = \int_a^b b_i(t) g(t) dt, \; i = 1 \ldots N. \tag{2.48}$$

[7] Um der üblichen Notation von Taylorkoeffizienten zu entsprechen, können die Funktionen dieses entarteten Kerns auch im Bereich 0...2 indiziert werden.

Der Beweis folgt aus der linearen Unabhängigkeit der a_i, b_i und wird in [56] angegeben.

2.5.2 Integralgleichungen mit nicht entartetem Kern

Da $C^N[a,b]$ Banachraum ist [56], können folgende Aussagen getroffen werden.

Satz 2.5.2 *Sei K ein stetiger Endomorphismus des Banachraums X mit der Norm $\|\cdot\|$. Die Neumannsche Reihe*

$$\sum_{n=0}^{\infty} K^n \tag{2.49}$$

konvergiert genau dann gleichmäßig, wenn

$$\lim_{n\to\infty} \|K^n\|^{\frac{1}{n}} < 1,$$

insbesondere also für $\|K\| < 1$.

Konvergiert die Neumannsche Reihe *gleichmäßig, so besitzt $I - K$ eine stetige Inverse auf ganz X, die gegeben ist durch*

$$(I-K)^{-1} := \sum_{n=0}^{\infty} K^n. \tag{2.50}$$

Der Beweis wird in [56] geführt.

Hieraus erhält man für die Lösung der Integralgleichung (2.43) mit nicht entartetem Kern direkt den folgenden Satz [72]:

Satz 2.5.3 *Hat der stetige Endomorphismus K die Eigenschaft*

$$\lim_{n\to\infty} \|\lambda^n K^n\|^{\frac{1}{n}} < 1, \tag{2.51}$$

dann besitzt die lineare Fredholmsche Integralgleichung (2.43) genau eine Lösung, die durch

$$y = \sum_{n=0}^{\infty} \lambda^n K^n g \tag{2.52}$$

gegeben ist.

Mit diesem Satz erhält man gleichzeitig eine konstruktive Iterationsvorschrift, um im Falle der Gültigkeit von (2.51) die Lösung (2.52) zu bestimmen[8]:

$$\begin{aligned} y^0 &:= g \\ y^n &:= g + \lambda K y^{n-1},\ n > 0. \end{aligned} \tag{2.53}$$

$$q \text{ mit } q(s,t) := \sum_{n=0}^{\infty} \lambda^n k^n(s,t) \tag{2.54}$$

heißt *lösender Kern* der Integralgleichung, der zugehörige Operator Q *lösende Transformation*. Hiermit geht (2.52) über in

$$y = g + Qg. \tag{2.55}$$

Die beiden oben angegebenen Sätze zur Lösung linearer Fredholmscher Integralgleichungen zweiter Art mit entartetem bzw. nicht entartetem Kern sind die Grundlage des von Klein in [72] beschriebenen Verfahrens, das in Abschnitt 3.1 erläutert wird.

2.5.3 Integralgleichungssysteme

Die gleichen Aussagen wie für lineare Fredholmsche Integralgleichungen können auch für Systeme linearer Fredholmscher Integralgleichungen getätigt werden.

[8] Fehler bei der praktischen Berechnung bleiben hier außer Betracht.

Ein System linearer Fredholmscher Integralgleichungen zweiter Art ist gegeben durch

$$y^i(s) - \lambda \sum_{j=1}^{N} \int_{\alpha_j}^{\beta_j} k^{ij}(s,t) y^j(t) dt = g^i(s), \alpha_i \leq s \leq \beta_i, i = 1...N. \tag{2.56}$$

Dabei werden stetige Kerne k^{ij} über $D^{ij} := [\alpha_i, \beta_i] \times [\alpha_j, \beta_j]$ sowie stetige rechte Seiten g^i und Unbekannte y^i über $[\alpha_i, \beta_i]$ betrachtet, wobei

$$\alpha_i := a + \frac{i-1}{N}(b-a), \beta_i := a + \frac{i}{N}(b-a),\ i = 1...N,\ D := [a,b] \in \mathbb{R},$$

d.h. der Definitionsbereich wird äquidistant unterteilt.

Michlin zeigt in [95], dass sich das obige Integralgleichungssystem äquivalent auch in eine einzelne Fredholmsche Integralgleichung umwandeln lässt, so dass beim Übergang von der Integralgleichung (2.43) zum Integralgleichungssystem (2.56) alle relevanten mathematischen Eigenschaften erhalten bleiben. Durch geeignete Definitionen wird der Übergang von den einzelnen Definitionsbereichen der Kerne auf den Gesamtdefinitionsbereich vorgenommen. Diese Transformation lässt sich sowohl für den obigen Fall des unterteilten Gesamt-Definitionsbereiches als auch in ähnlicher Weise für den Fall, dass alle Kerne auf dem gesamten Intervall $[a, b]$ definiert sind, vornehmen. Für die oben angegebene Unterteilung des Definitionsbereichs wird wie folgt vorgegangen:

Auf dem Intervall [a,b] definiert man für $1 \leq i \leq N$ und $\alpha_i \leq s < \beta_i$

$$\begin{aligned} \Phi(s) &:= y^i(s) \quad \text{und} \\ G(s) &:= g^i(s). \end{aligned}$$

Weiter definiert man

$$K(s,t) := k^{ij}(s,t)$$

mit $1 \leq i, j \leq N$ derart, dass $\alpha_i \leq s < \beta_i$ und $\alpha_j \leq t < \beta_j$.

Sind g^i, y^i, k^{ij} stetig und integrierbar über den jeweiligen Definitionsbereichen, so sind die entstehenden Funktionen Φ, G, K stückweise stetig und integrierbar, und es gilt für $\alpha_i \leq s < \beta_i$ und $i = 1...N$:

$$\begin{aligned} & \int_a^b K(s,t)\Phi(t)dt \\ = & \sum_{j=1}^{N} \int_{\alpha_j}^{\beta_j} K(s,t)\Phi(t)dt \\ = & \sum_{j=1}^{N} \int_{\alpha_j}^{\beta_j} k^{ij}(s,t)y^j(t)dt. \end{aligned}$$

Damit geht insgesamt das Integralgleichungssystem (2.56) über in

$$\Phi(s) - \lambda \int_a^b K(s,t)\Phi(t)dt = G(s). \tag{2.57}$$

In den nachfolgenden Betrachtungen sollen Fredholmsche Integralgleichungssysteme in der folgenden Matrix-Vektor-Schreibweise dargestellt werden.

Für Fredholmsche Integralgleichungssysteme definieren wir

$$\begin{aligned} \underset{\sim}{y} &:= (y^1, ..., y^N)^T \\ \underset{\sim}{k} &:= (k^{ij})_{i,j=1...N} \\ \underset{\sim}{g} &:= (g^1, ..., g^N)^T. \end{aligned}$$

So erhalten wir für das Integralgleichungssystem (2.56) die Darstellung

$$\underset{\sim}{y}(s) - \lambda \int_a^b \underset{\sim}{k}(s,t)\underset{\sim}{y}(t)dt = \underset{\sim}{g}(s) \tag{2.58}$$

in Matrix-Vektor-Schreibweise. Dabei werden die Einträge der Matrix bzw. der Vektoren immer auf ihre Definitionsbereiche D^{ij} bzw. $[a_i, b_i]$ bezo-

gen, insbesondere wird das Integral ausgewertet als

$$\int_a^b \underset{\sim}{k}(s,t)\underset{\sim}{y}(t)dt := \begin{pmatrix} \sum_{j=1}^{N} \int_{\alpha_j}^{\beta_j} k^{1j}(s,t)y^j(t)dt \\ \vdots \\ \sum_{j=1}^{N} \int_{\alpha_j}^{\beta_j} k^{Nj}(s,t)y^j(t)dt \end{pmatrix}. \tag{2.59}$$

Hierdurch ist auch ein Matrix-Kernoperator $\underset{\sim}{K} := (K^{ij})_{i,j=1...N}$ zu $\underset{\sim}{k}$, bestehend aus den Operatoren K^{ij} zu den Kernen $k^{ij}, i,j = 1...N$, erklärt.

Beispiel 2.5.3 *Die obige Notation soll an einem Beispiel verdeutlicht werden. Hierfür wird die Zerlegung einer einzelnen Integralgleichung in ein System von Integralgleichungen gewählt, so wie sie in den späteren Tests betrachtet wird.*

Sei die Integralgleichung

$$y(s) - \int_0^1 k(s,t)y(t)dt = g(s)$$

mit

$$k : [0,1] \times [0,1] \to \mathbb{R}, (s,t) \mapsto \exp(s+t)$$

und

$$g : [0,1] \to \mathbb{R}, s \mapsto s - \exp(s)$$

gegeben. Dann seien

$$\alpha_1 := 0, \alpha_2 := \tfrac{1}{2}, \beta_1 := \tfrac{1}{2}, \beta_2 := 1,$$

d.h.

$$[\alpha_1, \beta_1] = [0, \tfrac{1}{2}], [\alpha_2, \beta_2] = [\tfrac{1}{2}, 1]$$

und

$$
\begin{aligned}
k^{11} &: [0, \tfrac{1}{2}] \times [0, \tfrac{1}{2}] \to \mathbb{R}, (s,t) \mapsto \exp(s+t) \\
k^{12} &: [0, \tfrac{1}{2}] \times [\tfrac{1}{2}, 1] \to \mathbb{R}, (s,t) \mapsto \exp(s+t) \\
k^{21} &: [\tfrac{1}{2}, 1] \times [0, \tfrac{1}{2}] \to \mathbb{R}, (s,t) \mapsto \exp(s+t) \\
k^{22} &: [\tfrac{1}{2}, 1] \times [\tfrac{1}{2}, 1] \to \mathbb{R}, (s,t) \mapsto \exp(s+t) \\
g^1 &: [0, \tfrac{1}{2}] \to \mathbb{R}, s \mapsto s - \exp(s) \\
g^2 &: [\tfrac{1}{2}, 1] \to \mathbb{R}, s \mapsto s - \exp(s).
\end{aligned}
$$

Damit sind

$$
\underset{\sim}{k} := \begin{pmatrix} k^{11} & k^{12} \\ k^{21} & k^{22} \end{pmatrix}
$$

und

$$
\underset{\sim}{g} := \begin{pmatrix} g_1 \\ g_2 \end{pmatrix}
$$

definiert, und das Integral hat die Gestalt

$$
\int_0^1 \underset{\sim}{k}(s,t)y(t)dt := \begin{pmatrix} \int_0^{\frac{1}{2}} \exp(s+t)y^1(t)dt + \int_{\frac{1}{2}}^1 \exp(s+t)y^2(t)dt \\ \int_0^{\frac{1}{2}} \exp(s+t)y^1(t)dt + \int_{\frac{1}{2}}^1 \exp(s+t)y^2(t)dt \end{pmatrix},
$$

wobei die Einträge bezüglich der Zuordnungsvorschrift, nicht aber bezüglich der Definitionsbereiche gleich sind. Es ist also zu sehen, dass es sich auch bei der Darstellung in Matrix-Vektor-Schreibweise mittels verschiedener Definitionsbereiche immer noch um eine einzelne Funktion handelt.

Bei der Darstellung von Intervall-Funktionen mittels Taylorarithmetik wie in den später praktisch durchgeführten Berechnungen werden die oben gezeigten Einträge aufgrund der Taylorentwicklung um unterschiedliche Entwicklungspunkte nicht mehr gleich sein.

Im Raum der wie oben definierten Vektoren kann eine Norm

$$
\|x\| := \sum_{i=1}^{N} \|x^i\| = \sum_{i=1}^{N} \max_{\alpha_i \le s \le \beta_i} |x^i(s)| \tag{2.60}
$$

definiert werden, wobei für die Einträge x^i die Maximumsnorm angewandt wird.

Weitere Untersuchungen zu Integralgleichungssystemen werden auch in [134] angestellt.

Integralgleichungssysteme mit entartetem Kern

Zur Lösung des Integralgleichungssystems mit entarteten Kernen seien alle Kerne k^{ij} entartet, d.h. es gebe für jeden Kern k^{ij} stetige Funktionen $a_m^{ij} : [\alpha_i, \beta_i] \to \mathbb{R}$ und $b_m^{ij} : [\alpha_j, \beta_j] \to \mathbb{R}$, $m = 1...T^{ij} \in \mathbb{N}$, so dass die a_m^{ij} und b_m^{ij} für $m = 1...T^{ij}$ linear unabhängig sind, und so dass gilt:

$$k^{ij}(s,t) = \sum_{m=0}^{T^{ij}} a_m^{ij}(s) b_m^{ij}(t), i,j = 1...N. \tag{2.61}$$

Zur Vereinfachung soll vorausgesetzt werden, dass der Grad T^{ij} der Entartung der Kerne k^{ij} für alle $i,j = 1...N$ gleich ist:

$$T^{ij} = T \in \mathbb{N}, i,j = 1...N.$$

In der praktischen Anwendung der später beschriebenen Lösungsverfahren stellt dies keine Einschränkung dar, da bei der Darstellung der Kerne durch Taylorentwicklung der Grad der Entartung durch die Ordnung der Taylorentwicklung gewählt werden kann.

Für die a_m^{ij} wie oben definiert soll zusätzlich vorausgesetzt werden, dass diese nur vom Index i abhängen, d.h.

$$a_m^{i1} = ... = a_m^{iN}, i = 1...N, m = 1...T.$$

Diese Voraussetzung ist bei Verwendung der Taylorarithmetik zur praktischen Durchführung von Lösungsverfahren erfüllt [72]. Bei der Darstellung der Kerne k^{ij} durch Taylorentwicklung der Ordnung T haben die k^{ij} nämlich die folgende Form:

$$\begin{aligned} k^{ij}(s,t) &= \sum_{m=0}^{T}\sum_{n=0}^{T-m}\frac{1}{m!n!}\frac{\partial^{m+n}k^{ij}}{\partial s^m \partial t^n}(s_0^{ij},t_0^{ij})(s-s_0^{ij})^m(t-t_0^{ij})^n \\ &= \sum_{m=0}^{T}\underbrace{(s-s_0^{ij})^m}_{=:a_m^{ij}(s)}\underbrace{\sum_{n=0}^{T-m}\frac{1}{m!n!}\frac{\partial^{m+n}k^{ij}}{\partial s^m \partial t^n}(s_0^{ij},t_0^{ij})(t-t_0^{ij})^n}_{=:b_m^{ij}(t)}. \end{aligned}$$

Der Term

$$a_m^{ij}(s) := (s-s_0^{ij})^m$$

ist aber nur von der ersten Variablen s abhängig und über

$$D^i := [\alpha_i,\beta_i], 1 \le i \le N$$

definiert. Der Entwicklungspunkt s_0^{ij} wird daher unabhängig von j als

$$mid(D^i) =: s_0^i =: s_0^{ij}, j = 1...N$$

gewählt. Somit gilt

$$a_{i1} = ... = a_{iN}, i = 1...N.$$

Damit lässt sich die Kernmatrix $\mathcal{k}$ schreiben als

$$\mathcal{k}(s,t) = \sum_{m=0}^{T}\mathcal{a}_m(s)\mathcal{b}_m(t) = \left(\sum_{m=0}^{T}a_m^i(s)b_m^{ij}(t)\right)_{i,j=1...N} \tag{2.62}$$

mit Matrizen

$$\mathcal{a}_m := \begin{pmatrix} a_m^1 & 0 & \cdots & 0 \\ 0 & a_m^2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & a_m^N \end{pmatrix} \tag{2.63}$$

und

$$\mathfrak{b}_m := \begin{pmatrix} b_m^{11} & \cdots & b_m^{1N} \\ \vdots & & \vdots \\ b_m^{N1} & \cdots & b_m^{NN} \end{pmatrix}. \tag{2.64}$$

Im Fall entarteter Kerne lässt sich also auch das Integralgleichungssystem mit der entarteten Kerndarstellung in Matrixschreibweise äquivalent zum Fall der einzelnen Integralgleichung formulieren.

Unter Verwendung des dortigen Lösungsansatzes

$$\mathfrak{y}(s) = \mathfrak{g}(s) + \lambda \sum_{m=0}^{T} \mathfrak{a}_m(s)\mathfrak{x}_m(s)$$

für einen Vektor $x_m(s)$ (unter Anwendung der jeweils gegebenen Definitionsbereiche) ergibt sich durch Einsetzen in das Ursprungssystem wieder ein lineares Gleichungssystem der entsprechend höheren Dimension $N \cdot (T+1) \times N \cdot (T+1)$ und es lässt sich der entsprechende Satz formulieren:

Satz 2.5.4 *Seien $\mathfrak{k}, \mathfrak{g}, \mathfrak{y}$ wie oben und $\mathfrak{k}$ entartet mit der Darstellung (2.62). Dann besitzt jede Lösung $\mathfrak{y}$ des linearen Fredholmschen Integralgleichungssystems (2.56) die Darstellung*

$$\mathfrak{y}(s) = \mathfrak{g}(s) + \lambda \sum_{m=0}^{T} \mathfrak{a}_m(s)\mathfrak{x}_m(s) \tag{2.65}$$

und (2.65) löst das Integralgleichungssystem genau dann, wenn die Vektoren $\mathfrak{x}_m(s)$, $m = 0...T$ das lineare Gleichungssystem

$$\mathfrak{x}_m(s) - \lambda \sum_{n=0}^{T} \mathfrak{b}_m(t)\mathfrak{a}_n(t)dt\,\mathfrak{x}_n(s) = \int_a^b \mathfrak{b}_m(t)\mathfrak{g}(t)dt \tag{2.66}$$

lösen.

Das obige lineare Gleichungssystem enthält insbesondere Blöcke

$$Q_{mn} := \int_a^b \mathcal{b}_m(t)\mathcal{a}_n(t)dt,\ m,n = 0...T,$$

die komponentenweise aus genau einem Summanden mit Integral

$$Q_{mn}^{ij} = \sum_{k=1}^{N} \int_{\alpha_k}^{\beta_k} b_m^{ik}(t)a_n^j(t)dt \qquad (2.67)$$

$$\overset{(2.63),(2.64)}{=} \int_{\alpha_j}^{\beta_j} b_m^{ij}(t)a_n^j(t)dt \qquad (2.68)$$

bestehen mit $i,j \in \{1,...,N\}$.

Integralgleichungssysteme mit nicht entartetem Kern

Wegen

$$\|\mathcal{K}x\| = \sum_{i=1}^{N} \| \sum_{j=1}^{N} K^{ij}x^j \| \le \sum_{j=1}^{N} \|x^j\| \sum_{i=1}^{N} \|K^{ij}\| \qquad (2.69)$$

kann die Bedingung aus Satz 2.5.2 ebenso für den nunmehr betrachteten Raum formuliert werden als

$$|\lambda| \sum_{i,j=1}^{N} \|K^{ij}\| < 1,$$

so dass der betreffende Lösungssatz auf Integralgleichungssysteme übertragen werden kann.

Es kann also die Vorgehensweise für einzelne Operatoren übernommen werden und eine Iteration in der Form

$$y = g + \lambda\mathcal{K}y =: \mathcal{T}y \qquad (2.70)$$

mit einem zugehörigen Operator $\mathcal{T}$ formuliert werden.

2.6 Paralleles Rechnen

In den vorangehenden Abschnitten wurden die mathematischen Grundlagen angegeben, die den in Kapitel 3 erläuterten Verfahren zu Grunde liegen. In diesem Abschnitt sollen diesen noch Grundlagen des parallelen Rechnens hinzugefügt werden. Dabei sind die zwei Aspekte

- parallele Hardware und
- Software für parallele Hardware-Umgebungen

zu betrachten.

Paralleles Rechnen kann definiert werden als die *Benutzung mehrerer Recheneinheiten zur Lösung eines einzelnen Problems* [117]. Unter einer Recheneinheit soll hier ein *Prozessor (CPU)* als physikalische Recheneinheit verstanden werden. Als Abstraktion des Prozessors von der physikalischen Umgebung wird der Begriff des *Prozesses* verwendet. Mehrere Prozesse können so physikalisch demselben Prozessor zugeordnet sein. Umgebungen, in denen ausschließlich abstrakte *Prozesse* vorliegen, nicht jedoch eine physikalisch parallele Umgebung, sollen hier allerdings nicht näher betrachtet werden, d.h. in der vorliegenden Arbeit werden Umgebungen betrachtet, in denen jeder Prozess auch hardwareseitig auf einem separaten Prozessor ausgeführt wird.

Parallele Hardware kann auf verschiedene Weise charakterisiert bzw. klassifiziert werden. Betrachtet man die parallele Arbeitsweise, so lassen sich

- *Pipelining* und
- *Nebenläufigkeit*

unterscheiden. Von Pipelining spricht man, wenn eine *einzelne Operation* in mehrere Takte zerfällt, so dass dieselbe Operation für unterschiedliche Operanden gleichzeitig überlappend durchgeführt werden kann. Dieses Prinzip wird in *Vektorrechnern* angewandt, die *Vektoren* von Operanden für eine Operation gleichzeitig verarbeiten. Bei Rechnern, die dieselbe oder unterschiedliche Operationen gleichzeitig ausführen, ohne dabei einzelne Operationen zu zerlegen, spricht man von nebenläufiger Arbeitsweise [8]. Häufig wird unter einem *Parallelrechner* ausschließlich ein

Rechner verstanden, der Parallelität in Form der Nebenläufigkeit verwirklicht.

Eine wichtige und oft verwendete Klassifikation von Hardware ist die Klassifizierung nach Flynn [31], die Rechner anhand der verarbeiteten Programme und Daten einteilt:

- **SISD** (single instruction, single data)- Rechner sind klassische serielle Rechner.
- **SIMD** (single instruction, multiple data)- Rechner führen jeweils dieselben Operationen für unterschiedliche Daten auf den unterschiedlichen Prozessoren aus; meist werden auch Vektorrechner dieser Kategorie zugeordnet.
- **MIMD** (multiple instruction, multiple data)- Rechner haben dagegen unterschiedliche Eingabeströme bzw. -programme für jeden Prozessor.

Ferner werden **SPMD** (single program, multiple data)- Rechner gesondert bezeichnet, die zwar auf den einzelnen Prozessoren asynchron unterschiedliche Operationen gleichzeitig ausführen können, bei denen jedoch jeder Prozessor das gleiche Gesamtprogramm ausführt. Diese Rechner können, insbesondere aus Hardware-Sicht, als Spezialfall von MIMD-Rechnern angesehen werden, etwa wird in [63] der Begriff des *SPMD-Modus* eines MIMD-Systems benutzt.

Eine weitere Ebene der Klassifizierung liegt in der Speicherverwaltung eines Parallelrechners. Hier wird unterschieden zwischen

- *gemeinsamem Speicher (shared memory)* und
- *verteiltem Speicher (distributed memory).*

Systeme mit gemeinsamem Speicher haben einen zentral verwalteten Speicher, auf den alle Prozessoren über das Verbindungsnetzwerk zugreifen können. Unterschiedliche Prozesse auf unterschiedlichen Prozessoren können somit über den Speicher miteinander kommunizieren. Bei Systemen mit verteiltem Speicher besitzt jeder Prozessor lokal verwalteten Speicher. Es existieren ferner kombinierte Lösungen, bei denen je eine Teilmenge von Prozessoren Zugriff auf einen gemeinsamen Speicher hat, der für die

betrachtete Teilmenge von Prozessoren lokal ist; hier spricht man von *distributed shared memory*. Zusätzlich betrachtet man die Zugriffsmöglichkeiten und -zeiten auf den Speicher.

- Von *uniform memory access (UMA)* spricht man, wenn alle Prozessoren in einheitlicher Zeit auf den gemeinsamen Speicher zugreifen können.
- Falls verschiedene Prozessoren auf denselben Speicher zugreifen können, aber nicht in einheitlicher Zeit, wird dies als *non-uniform memory access (NUMA)* bezeichnet.
- Architekturen mit verteiltem Speicher, bei denen jeweils nur auf den lokalen Speicher Zugriff besteht, werden *no remote access (NORMA)*-Architekturen genannt.

Des Weiteren ist die *Netzwerktopologie* der Prozessoren als Unterscheidungsmerkmal zu betrachten, d.h. die Verknüpfungsstruktur der Prozessoren eines Parallelrechners untereinander. Typische Beispiele hierfür sind

- Ring
- Gitter
- Baum
- Torus

Diese Strukturen sind in Abbildung 2.1 dargestellt. Bei ihnen handelt es sich um *statische*, d.h. fest geschaltete Verbindungsnetzwerke.

Bei *dynamischen* Verbindungsnetzwerken dagegen werden die Verbindungen während der Programmverarbeitung aktiv geschaltet. Ein Beispiel eines *dynamischen* Verbindungsnetzwerks ist der *Crossbar Switch*, der mittels Schaltern je zwei Prozessoren untereinander oder Prozessoren mit Speicherorten verbinden kann. Ein *Crossbar Switch* ist in Abbildung 2.2 dargestellt.

Neben der Hardware ist auch die Softwareumgebung für Parallelrechner von besonderer Bedeutung. Zunächst sind

- *explizite Parallelität* und
- *implizite Parallelität*

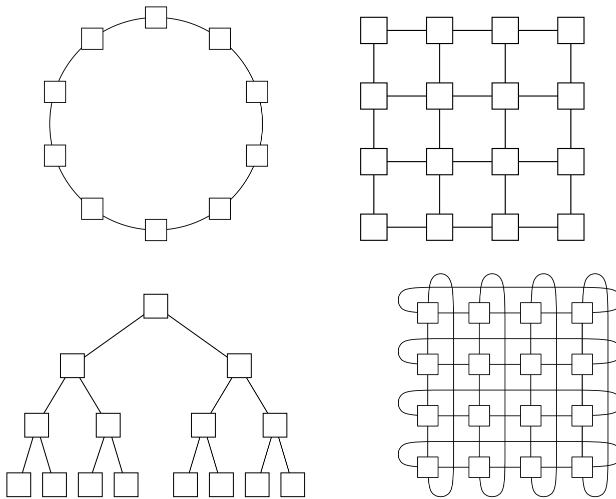

Abbildung 2.1: Statische Prozessortopologien: Ring, zweidimensionales Gitter, Baum und 2D-Torus

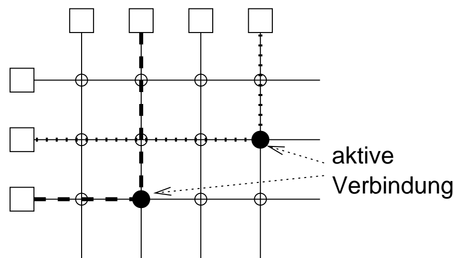

Abbildung 2.2: Dynamische Prozessortopologien: Crossbar Switch

voneinander abzugrenzen. Implizite Parallelität kann z.B. durch Compiler verwirklicht werden, wie in [8] oder [135] dargestellt wird. Sie bezeichnet die automatische Erzeugung parallelen Codes aus seriellen Programmen. Explizite Parallelität dagegen verlangt, dem Wortsinn entsprechend, die explizite Behandlung paralleler Abläufe und Kommunikation bei der Erstellung der Software zur Lösung des betrachteten Problems. Zur Verwirklichung expliziter Parallelität können parallele Programmiersprachen oder Erweiterungen existierender Sprachen verwendet werden, letztere entweder in Form integrierter Erweiterungen zur parallelen Programmierung oder in Form von Bibliotheken für die existierende Sprache.

Die zu verwendenden Sprachen oder Bibliotheken müssen dabei auf die Ziel-Kategorie von Parallelrechnern angepasst sein, d.h. Werkzeuge zur Verfügung stellen, die die Fähigkeiten des parallelen Systems (SIMD oder MIMD, gemeinsamer oder verteilter Speicher etc.) in angemessener Weise nutzen können.

Kommunikation

Bei Rechnern mit verteiltem Speicher und *NORMA*-Architektur muss ein Mechanismus der Nachrichtenübermittlung *(Message Passing)* zwischen den einzelnen Prozessoren (bzw. zwischen den einzelnen Prozessen) geschaffen werden, um Kommunikation zu betreiben. Ein weit verbreiteter Standard hierzu ist das *Message Passing Interface (MPI)*, das in Abschnitt 5.1.3 vorgestellt wird.

Dem Terminus *Nachrichtenübermittlung* entsprechend, hat jede zu übermittelnde Nachricht Absender *(Sender)* und Empfänger *(Receiver)*. Je nach Art des Kommunikationsvorganges sind dabei jeweils ein oder mehrere Absender und Empfänger beteiligt.

Bei der Kommunikation mit Message Passing unterscheidet man zwischen

- *Point-to-point Communication* und
- *Collective Communication*.

Im ersten Fall ist jeweils genau ein Prozess als Absender und Empfänger am Kommunikationsvorgang beteiligt, im zweiten Fall gibt es mehrere Absender und/oder Empfänger.

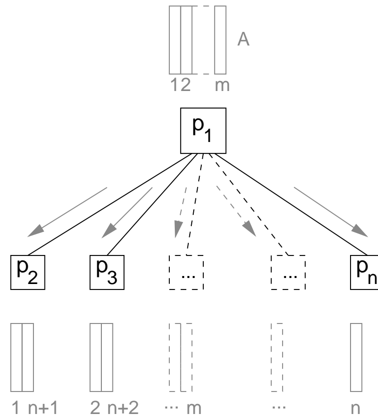

Abbildung 2.3: Master-Worker-Modell: Verteilung von Matrixdaten vom Masterprozess an die Worker

Typische Operationen der *collective communication* sind

- *Broadcast:* Ein Datenobjekt wird von einem Absender an alle anderen Prozesse verschickt.
- *Scatter:* Ein Datenvektor (oder entsprechend partitionierbares Objekt) wird stückweise auf alle Prozesse verteilt.
- *Gather:* Ein Datenvektor (oder entsprechend partitionierbares Objekt) wird aus Teilstücken gebildet, wobei der (einzelne) Empfänger von jedem Prozess ein Teilstück empfängt.

Kommunikationsmodelle

Auf Parallelrechnern können verschiedene Kommunikationsmodelle implementiert werden, die mit einer unterschiedlichen Datenhaltung und einer unterschiedlichen Verwaltung der zu bearbeitenden und zu verteilenden Teilaufgaben einhergehen.

Als verbreitetes Modell, das auch in den in dieser Arbeit betrachteten Verfahren zum Einsatz kommt, ist das *Master-Slave-Modell* bzw. *Master-*

Worker-Modell zu nennen. Dieses Modell zeichnet einen Prozess als *Master* aus, der den *Slaves* bzw. *Workern* Aufgaben zuweist. Die Worker bearbeiten die zugewiesenen Aufgaben und schicken deren Lösungen am Schluss zurück an den Master. Dabei kann der Master sowohl in erster Linie die Verteilung von Daten vornehmen (siehe Abbildung 2.3: Verteilung von Matrixspalten an die Worker) als auch Informationen zu bereits verteilt vorliegenden Daten verteilen. Die Arbeit des Masters kann auf die Verwaltung von Teilaufgaben und Daten beschränkt sein, der Master kann aber auch gleichzeitig die Aufgaben eines zusätzlichen Workers übernehmen und eigene Teilaufgaben bearbeiten, während die Worker ihrerseits Teilaufgaben erledigen.

Das Modell sieht in der Grundform keine Kommunikation zwischen den Workern vor. Allerdings können abgewandelte Modelle dies zur Anpassung auf das zu behandelnde Problem vorsehen. Insbesondere bei Problemen mit adaptiven Unterteilungsstrategien kommen oft individuell angepasste Kommunikationsmodelle zum Einsatz. Schematische Darstellungen von Kommunikationsmodellen bei Problemen der verifizierten globalen Optimierung, in der derartige Strategien zum Einsatz kommen, finden sich z.B. in [124].

3 Verfahren

In Kapitel 2 wurden die allgemeinen Grundlagen erläutert, auf denen die in dieser Arbeit betrachteten Probleme basieren. In diesem Kapitel sollen nun die Verfahren zur Lösung der unterschiedlichen Probleme vorgestellt werden, auf denen insbesondere die in Kapitel 4 entwickelten parallelen Verfahren beruhen. Dabei handelt es sich zum einen um „Gesamtverfahren“ zur Lösung spezieller Probleme, insbesondere Verfahren zur Lösung Fredholmscher Integralgleichungen zweiter Art. Zum anderen handelt es sich um Teilverfahren, die im Rahmen der erstgenannten Verfahren Teilprobleme lösen. Auch wenn sie in diesem Kontext als Teilverfahren aufgefasst werden, so stellen diese Verfahren dennoch eigenständige Problemstellungen dar und werden teilweise in eigenen Abschnitten erläutert. Je nach Bedeutungsgrad im Kontext dieser Arbeit werden die Verfahren in unterschiedlicher Ausführlichkeit beleuchtet.

Das Hauptaugenmerk liegt auf den Verfahren zur Lösung linearer Fredholmscher Integralgleichungen und Integralgleichungssysteme zweiter Art, die von *Klein* in [72] und [73] vorgestellt wurden. Die Eigenschaften dieser Verfahren sollen in den nachfolgenden Kapiteln mit neuen Implementierungen getestet werden. Darüber hinaus wird ein paralleles Verfahren entwickelt (siehe Kapitel 4) und untersucht, das die Vorteile der Aufsplittung einzelner Integralgleichungen in Systeme von Integralgleichungen nutzbar macht, indem es den Rechenaufwand für das Systemverfahren verteilt und so die Rechenzeiten reduziert.

Als Vergleichsverfahren wird das Verfahren zur Lösung nichtlinearer Fredholmscher Integralgleichungen zweiter Art vom Urysohn-Typ beschrieben, das *Obermaier* in [106] angibt. Für dieses Verfahren gilt insbesondere, dass es die Lösung linearer Fredholmscher Integralgleichungen zweiter Art als Spezialfall enthält und somit den Vergleich von speziell auf lineare Integralgleichungen ausgerichteten Verfahren mit allgemeinen

Verfahren, angewandt auf lineare Integralgleichungen, erlaubt. Überdies liegt eine parallele Implementierung des Verfahrens vor, die nach Erstellung einer modifizierten Version in der gleichen Umgebung wie die Implementierungen der erstgenannten Verfahren verwendet werden konnte.

Als Teilverfahren innerhalb der erstgenannten Verfahren treten die folgenden Verfahren auf:

- *Ein-und zweidimensionale Taylorentwicklung von Funktionen*: Die theoretischen Aspekte der Taylorentwicklung wurden bereits in Kapitel 2 angegeben. Es verbleiben nur noch Betrachtungen der vorliegenden Implementierungen, die in Kapitel 5 erfolgen.
- *Verifizierte Lösung linearer Gleichungssysteme*: Lösungsverfahren für lineare Gleichungssysteme gehören zu den grundlegenden Verfahren der Numerik. Als wichtiges Verfahren zur *verifizierten* Lösung von *Intervall*-Gleichungssystemen wird das Verfahren von Rump [113] angegeben, für das in Kapitel 4 eine neue parallele Version und in Kapitel 5 eine neue parallele Implementierung in C-XSC [71] vorgestellt wird.
- *Verfahren zur Matrixinversion* als Teilproblem bei der verifizierten Lösung von linearen Intervall-Gleichungssystemen.

Das Vergleichsverfahren von Obermaier bedient sich ebenso mehrerer Teilverfahren, die nur in Kurzform beschrieben werden sollen; u.a. ist dies das *Nyström*-Verfahren zur Berechnung einer Näherungslösung linearer Fredholmscher Integralgleichungen zweiter Art [103].

Bei den obigen Verfahren handelt es sich um Verfahren mathematischer Natur. Weitere Verfahren, die hauptsächlich softwaretechnischer Natur sind, wie z.B. Vorgehensweisen zur Behandlung benutzerdefinierter Datentypen mit MPI, werden in Kapitel 5 vorgestellt. Dort werden auch die Implementierungen der hier vorgestellten Verfahren diskutiert.

Zunächst sollen also die genannten Verfahren zur Lösung Fredholmscher Integralgleichungen zweiter Art erläutert werden. Dazu sollen zur Übersicht zuerst mögliche Arten von Lösungsverfahren genannt werden.

Verfahren zur Lösung linearer Fredholmscher Integralgleichungen zweiter Art können verschiedene Ansätze verfolgen:

- *Zerlegung des Kerns in einen entarteten und einen nicht entarteten Anteil*: Die spezielle Gestalt des entarteten Anteils wird hier ausgenutzt, um für diesen Anteil ein Lösungsverfahren anzugeben. Für den nicht entarteten Anteil werden gleichzeitig zusätzliche Bedingungen formuliert, die die Lösung für diesen Anteil ermöglichen. Schließlich wird gezeigt, dass die geforderte Zerlegung so gewählt werden kann, dass die entsprechenden Bedingungen erfüllt sind. Das nachfolgend vorgestellte Verfahren von Klein bedient sich dieses Ansatzes.

- *Projektionsverfahren*: In diesen Verfahren wird die Gleichung nicht in dem zu Grunde liegenden, unendlich dimensionalen Raum X, sondern in einem endlich dimensionalen Teilraum $X_n, n \in \mathbb{N}$ (bzw. einer Folge solcher Räume für variierende Dimensionen $n \in \mathbb{N}$) betrachtet. Zur Übertragung der Funktionen bzw. Funktionswerte in diese Teilräume werden *Projektionsoperatoren*[1] $P_n, n \in \mathbb{N}$ verwendet, die die Elemente von X auf Elemente von X_n abbilden. Es wird dann gezeigt, dass die Differenz des ursprünglichen und des projizierten Kernoperators der Norm nach gegen 0 konvergiert für $n \to \infty$ und somit durch die Betrachtung der Lösung auf den endlich dimensionalen Räumen (mit endlicher Basis) geeignete Näherungen für die tatsächliche Lösung gefunden werden können. Zu diesen Verfahren zählt etwa die *Kollokationsmethode*, die z.B. in [7] beschrieben wird.

- *Nyströmverfahren*: In diesem Verfahren werden die Integrale in der Gleichung durch die Verwendung von Quadraturformeln diskretisiert. Statt der Integralgleichung selbst werden dann die enstehenden *Nyström-Gleichungen* gelöst[2] und die Lösung für das Ursprungsintervall interpoliert. Für den verifizierten Fall wurde das Nyström-Verfahren von *Gienger* [36] beschrieben.

- *Lösung durch Übergang zu Integralgleichungssystemen*: Auch dieser Ansatz findet sich in den nachfolgend geschilderten Verfahren; er wird insbesondere im Hinblick auf die Parallelisierbarkeit der Verfahren ausgenutzt.

[1] Ein Projektionsoperator P von einem Raum X auf einen Teilraum $\tilde{X}$ ist ein beschränkter linearer Operator, der die Elemente des Raums $\tilde{X}$ auf sich selbst abbildet.

[2] benannt nach E. Nyström, der das Verfahren zuerst vorstellte [103].

- *Lösung mit Methoden für nichtlineare Integralgleichungen für den Spezialfall der linearen Integralgleichung*: In diesem Sinne wird in dieser Arbeit das Verfahren von Obermaier betrachtet.

Die verschiedenen Arten von Lösungsansätzen werden z.B. in [7] beschrieben. Mit verifizierten Verfahren setzt sich *Dobner* in [25] genauer auseinander.

3.1 Kleins Verfahren für lineare Fredholmsche Integralgleichungen und Integralgleichungssysteme zweiter Art

Das von Klein in [72] vorgestellte Verfahren zur Lösung linearer Fredholmscher Integralgleichungen zweiter Art beruht auf Satz 2.5.1 und Satz 2.5.3. Sollen allgemeine lineare Fredholmsche Integralgleichungen zweiter Art gelöst werden, die weder entartet sind noch den Voraussetzungen von Satz 2.5.1 genügen, kann man die Integralgleichung wie im Folgenden beschrieben zerlegen.

Satz 3.1.1 *Jeder stetige Kern einer Integralgleichung gemäß (2.43) lässt sich gleichmäßig durch eine Folge* $(K_{\mathfrak{E}\ n})_{n\in\mathbb{N}}$ *von entarteten Kernen der Form (2.45) approximieren.*

Dies ist eine Anwendung des Approximationssatzes von Weierstrass (siehe z.B. [130]), die z.B. in [55] erwähnt wird. Der Satz lässt sich insbesondere auf entartete Kerne in Taylor-Darstellung als spezielle Polynome anwenden.

Damit ist jeder stetige Kern k von der Form

$$k = k_{\mathfrak{E}} + k_{\mathfrak{N}} \tag{3.1}$$

mit entartetem Kern $k_{\mathfrak{E}}$ und nicht entartetem Kern $k_{\mathfrak{N}}$ und zugehörigen Operatoren $K_{\mathfrak{E}}$ und $K_{\mathfrak{N}}$.

Bemerkung 3.1.1 *Die wichtige Eigenschaft von $k_{\mathfrak{N}}$ ist in diesem Falle, dass $k_{\mathfrak{N}}$ so gewählt werden kann, dass $|\lambda|\|K_{\mathfrak{N}}\| < 1$ gilt, und der Lösungssatz für Integralgleichungen mit nicht entartetem Kern auf $k_{\mathfrak{N}}$ angewandt werden kann.*

Damit kann die Integralgleichung in Operatorschreibweise (2.44) notiert werden als

$$(I - \lambda K_{\mathfrak{N}})\, y - \lambda K_{\mathfrak{E}} y = g. \tag{3.2}$$

Mit dieser Darstellung können nun die Sätze 2.5.1 und 2.5.3 zusammengeführt werden in eine Formulierung des Satzes über die *Fredholmsche Alternative*:

Satz 3.1.2 *Sei $k : [a,b] \times [a,b] \to \mathbb{R}$ stetig, $\lambda \neq 0$. Dann ist entweder die inhomogene Fredholmsche Integralgleichung*

$$y(s) - \lambda \int_a^b k(s,t)y(t)dt = g(s) \tag{3.3}$$

für jede rechte Seite $g \in C[a,b]$ eindeutig lösbar und die zugehörige homogene Integralgleichung

$$y(s) - \lambda \int_a^b k(s,t)y(t)dt = 0 \tag{3.4}$$

besitzt in $C[a,b]$ nur die triviale Lösung, oder die homogene Integralgleichung (3.4) besitzt mindestens eine nicht-triviale Lösung und die inhomogene Integralgleichung (3.3) ist nicht für jede rechte Seite $g \in C[a,b]$ eindeutig lösbar.

Beweis: Der Beweis ist [56] zu entnehmen und wird in [72] wiedergegeben. Da er von Bedeutung für die Konstruktion der Lösung ist, soll er auch hier angegeben werden.

Sei k von der Form (3.1), so dass $K_{\mathfrak{N}}$ der Eigenschaft aus Bemerkung 3.1.1 genügt, und sei

$$R := I - \lambda K_{\mathfrak{N}}.$$

Dann existiert R^{-1} (mit zugehörigem Kern r^{-1}) und mit

$$S := R^{-1} K_{\mathfrak{E}}$$

gilt:

$$\begin{aligned} & (I - \lambda K_{\mathfrak{N}})\, y - \lambda K_{\mathfrak{E}} y = g & (3.5) \\ \Leftrightarrow \quad & (R - \lambda K_{\mathfrak{E}})\, y = g & (3.6) \\ \Leftrightarrow \quad & (I - \lambda S)\, y = R^{-1} g. & (3.7) \end{aligned}$$

Die entsprechende homogene Gleichung lautet

$$(I - \lambda S)\, y = 0. \tag{3.8}$$

S ist aber wieder Operator für einen entarteten Kern

$$\tilde{k} = \sum_{i=1}^{N} \alpha_i b_i$$

wegen

$$\begin{aligned} (Sy)(s) &= \int_a^b r^{-1}(s,u) \sum_{i=1}^{N} a_i(u) \int_a^b b_i(t) y(t)\, dt\, du & (3.9) \\ &= \sum_{i=1}^{N} \underbrace{\int_a^b r^{-1}(s,u) a_i(u) du}_{=:\alpha_i(s)} \int_a^b b_i(t) y(t) dt & (3.10) \end{aligned}$$

und weil die α_i wieder linear unabhängig sind nach [95].

Damit ist Satz 2.5.1 anwendbar und die Lösung hat die Form

$$y(s) = \int_a^b r^{-1}(s,t) g(t) dt + \lambda \sum_{i=1}^{N} \alpha_i(s) \xi_i(s), \tag{3.11}$$

falls $\xi_i(s), i = 1...N$ existieren, die das Gleichungssystem

$$\xi_i(s) - \lambda \sum_{j=1}^{N} \int_a^b b_i(t)\alpha_j(t)dt\, \xi_j(s) = \int_a^b b_i(t)f(t)\, dt \tag{3.12}$$

mit

$$f(t) := \int_a^b r^{-1}(t,u)g(u)\, du$$

lösen.

Ebenso hat das zugehörige homogene Gleichungssystem die Lösung

$$y(s) = \lambda \sum_{i=1}^{N} \alpha_i(s)\xi_i(s),$$

falls die $\xi_i(s), i = 1...N$ das Gleichungssystem

$$\xi_i(s) - \lambda \sum_{j=1}^{N} \int_a^b b_i(t)\alpha_j(t)dt\, \xi_j(s) = 0$$

lösen.

Da R bijektiv, d.h. insbesondere injektiv ist, ist, falls (3.4) nur die triviale Lösung besitzt, auch (3.8) nur trivial lösbar. Mit Satz 2.5.1 ist aber dann die zugehörige inhomogene Gleichung (3.7) eindeutig lösbar sowie wegen der Bijektivität von R dann auch (3.3). □

Eine weitergehende Darstellung der *Fredholmschen Alternative* findet sich auch in [109].

Dieser Beweis liefert das Lösungsverfahren für die allgemeine lineare Fredholmsche Integralgleichung zweiter Art. Zur Durchführbarkeit des Verfahrens ist es erforderlich, dass für den Kern k eine Darstellung

$$k = k_{\mathfrak{E}} + k_{\mathfrak{N}}$$

praktisch ermittelt werden kann.

Die Taylorentwicklung gemäß Abschnitt 2.3 liefert, falls durchführbar, eine solche Darstellung, wobei der entartete Kern $k_{\mathfrak{E}}$ gerade aus dem eigentlichen Taylorpolynom besteht und der nichtentartete Kern $k_{\mathfrak{N}}$ aus den Restgliedtermen.

3.1.1 Das praktische Verfahren

Im Folgenden wird das praktische Vorgehen zur Berechnung der Lösung für allgemeine Integralgleichungen gemäß Satz 3.1.2 dargestellt.

Ausgangspunkt ist wieder die folgende Darstellung in Operatorschreibweise:

$$y - \lambda K_{\mathfrak{N}} y - \lambda K_{\mathfrak{E}} y = g. \tag{3.13}$$

Umformuliert erhält man hieraus:

$$y - \lambda K_{\mathfrak{N}} y = g + \lambda K_{\mathfrak{E}} y. \tag{3.14}$$

(3.14) kann mit dem iterativen Verfahren (2.70) gelöst werden und hat somit eine lösende Transformation Q, mit der gilt:

$$y = g + \lambda K_{\mathfrak{E}} y + Q(g + \lambda K_{\mathfrak{E}} y) \tag{3.15}$$

$$= (g + Qg) + \lambda(K_{\mathfrak{E}} y + Q K_{\mathfrak{E}} y). \tag{3.16}$$

Somit erhält man

$$\begin{aligned} y(s) &= \overbrace{g(s) + \int_a^b q(s,t) g(t) dt}^{=:f(s)} \\ &\quad + \lambda \sum_{m=0}^{N} \underbrace{\left(a_m(s) + \int_a^b q(s,u) a_m(u) du \right)}_{=:\alpha_m(s)} \\ &\quad \cdot \int_a^b b_m(t) y(t) dt. \end{aligned} \tag{3.17}$$

Dies stellt eine neue Integralgleichung mit Inhomogenität f und entartetem Kern, der durch die α_m und b_m gegeben wird, dar.

Es lässt sich also das Lösungsverfahren für Integralgleichungen mit entartetem Kern anwenden.

Sowohl f als auch $\alpha_m, m = 0...N$, sind dabei Lösungen von Integralgleichungen nach dem iterativen Verfahren mit der lösenden Transformation q bzw. dem zugehörigen Operator Q.

Bevor der Algorithmus für das allgemeine Einzelverfahren angegeben werden kann, sollen noch einige Bemerkungen zur Durchführung des Verfahrens mit Hilfe der Taylorentwicklung und der Anwendung des Verfahrens zur Verifikation, d.h. für Intervalle, formuliert werden.

3.1.2 Bemerkungen zur Verwendung der Taylorentwicklung

Die folgenden Hinweise sind bei der praktischen Durchführung des Verfahrens mit Hilfe der Taylorentwicklung von Bedeutung.

Bemerkung 3.1.2 (Taylordarstellung des Kerns) *Ist die Taylordarstellung des Kerns gegeben durch*

$$k(s,t) = \sum_{m=0}^{N} (s-s_0)^m \left(\sum_{n=0}^{N-m} \frac{1}{m!\,n!} \frac{\partial^{m+n} k}{\partial s^m \partial t^n}(s_0,t_0)(t-t_0)^n \right) \tag{3.18}$$

$$+ \sum_{n=0}^{N} (s-s_0)^n \frac{1}{(N+1-n!)\,n!} \frac{\partial^{N+1} k}{\partial s^n \partial t^{N+1-n}}(\xi,\eta)(t-t_0)^{N+1-n} \tag{3.19}$$

(vgl. Abschnitt 2.3), so können die Funktionen des entarteten Kerns für

$m = 0 ... N$ gewählt werden durch[3]

$$a_m(s) \quad := \quad (s - s_0)^m \tag{3.20}$$

$$b_m(t) \quad := \quad \sum_{n=0}^{N-m} \underbrace{\frac{1}{m!\,n!} \frac{\partial^{m+n} k}{\partial s^m \partial t^n}(s_0, t_0)}_{=:\beta_{mn}} (t - t_0)^n. \tag{3.21}$$

Die verbleibende Summe (3.19) bildet die Restglieddarstellung für den nicht entarteten Kern.

Bemerkung 3.1.3 *(Berechnung der Integrale für das lineare Gleichungssystem)*

Die Matrix und die rechte Seite des linearen Gleichungssystems (3.12) für die Lösung der Integralgleichung sind gegeben durch

$$\begin{pmatrix} \mathbf{1} - \lambda \int_a^b B_1(t)\alpha_1(t)\,dt & \cdots & -\lambda \int_a^b B_1(t)\alpha_N(t)\,dt \\ \vdots & \ddots & \vdots \\ -\lambda \int_a^b B_N(t)\alpha_1(t)\,dt & \cdots & \mathbf{1} - \lambda \int_a^b B_N(t)\alpha_N(t)\,dt \end{pmatrix}$$

und

$$\begin{pmatrix} \int_a^b B_1(t)F(t)\,dt \\ \vdots \\ \vdots \\ \int_a^b B_N(t)F(t)\,dt \end{pmatrix}.$$

In den einzelnen Elementen treten Integrale von Produkten der Form

$$B_i(t)\alpha_j(t), 1 \leq i, j \leq N$$

[3] Hier wird ein Indexbereich von 0 bis N gewählt, da dies der üblichen Indizierung von Taylorkoeffizienten entspricht. Es handelt sich also im Hinblick auf die Darstellung entarteter Kerne um eine Entartung vom Grad $N + 1$.

bzw.

$$B_i(t)F(t), 1 \leq i \leq N$$

auf. Jeder Faktor ist dabei ein Polynom der Ordnung $\leq N$, d.h. das Produkt ist maximal ein Polynom der Ordnung $2N$. Für die Integration dieser Produkte genügt es damit bei Verwendung von Taylorpolynomen, die Integrale der Monome

$$(t - t_0)^n,\ n = 1...2N \tag{3.22}$$

mit Integrationsvariable t und Entwicklungspunkt t_0 der Taylorentwicklung zu kennen, um die vorkommenden Integrale zu berechnen.

Die Integrale der Monome können aber für alle Berechnungen bereits im Vorfeld bereitgestellt werden. Dies gilt analog für die im Systemverfahren auftretenden Produktausdrücke.

Beispiel 3.1.1 *Sind die Funktionen*[4] b_i, f *durch die Taylordarstellungen*

$$b_1 : [0,2] \to \mathbb{R}, t \mapsto 1 + 2(t-1)$$
$$f : [0,2] \to \mathbb{R}, t \mapsto 2 + 3(t-1)$$

gegeben, so ist gemäß Bemerkung 3.1.3 folgendes Integral zu berechnen:

$$\begin{aligned} & \int_0^2 (1 + 2(t-1)) \cdot (2 + 3(t-1))dt \\ = & \int_0^2 2 + 7(t-1) + 6(t-1)^2 dt. \end{aligned}$$

Um dieses Integral berechnen zu können, genügt es offensichtlich, die

[4] hier der Übersicht halber mit reellen Koeffizienten notiert

Stammfunktionen

$$\begin{aligned}\int c\,dt &= ct, c \in \mathbb{R}\\ \int (t-1)\,dt &= \frac{1}{2}(t-1)^2\\ \int (t-1)^2\,dt &= \frac{1}{3}(t-1)^3\end{aligned}$$

zu kennen. Werden diese im Vorfeld bestimmt, so lassen sich alle Integrale $\int_0^2 G(t)\cdot H(t)dt$ *von Produkten ähnlicher Funktionen in Taylordarstellung der Ordnung* 1 *mit Entwicklungspunkt* 1 *ebenfalls bestimmen.*

3.1.3 Bemerkungen zum verifizierten Verfahren

In den bisherigen Ausführungen wurden noch keine speziellen Aussagen zur Anwendung der Verfahren auf Intervalle getroffen. Zur Einschließung der Lösungen gemäß der angegebenen Verfahren auf Intervallbasis müssen noch folgende Sachverhalte betrachtet werden:

- Bei der Berechnung der Integrale von Monomen (3.22) erhöht sich die Dimension der integrierten Polynome.
- Bei der Anwendung der Iteration auf Intervalle muss die Konvergenz des Verfahrens sichergestellt werden.

Hierzu gelten die folgenden Betrachtungen.

Bemerkung 3.1.4 *Bei der Wahl des Lagrangeschen Restglieds der Taylorentwicklung als nicht entarteter Kern ist zu beachten, dass die Restgliedterme der Taylorentwicklung ein Polynom vom Grad* $N+1$ *darstellen. Im weiteren Verfahrensverlauf soll jedoch weiter mit Polynomen vom Grad* N *gerechnet werden. Um diese Darstellung zu erreichen, wird der Summand der Ordnung* $N+1$ *über dem ganzen Definitionsintervall ausgewertet, wodurch ein Polynom vom Grad* N *entsteht, das das Ursprungpolynom einschließt. (vgl. [72]).*

Es soll nun noch die Konvergenz des durch (2.53) gegebenen Iterationsverfahrens geprüft werden. Bei der praktischen Einschließung durch Intervalle ist die Konvergenz des Verfahrens zunächst nicht sichergestellt. Die benötigte Eigenschaft wird jedoch gesichert, da der untenstehende Fixpunktsatz anwendbar ist.

Hierzu ist die folgende Definition vorauszuschicken:

Definition 3.1.1 *Ein Operator $K \in L(X, Y)$, X, Y Banachräume, heißt* kompakt*, wenn das Bild $K(B)$ der abgeschlossenen Einheitskugel*

$$\{x \in X | \|x\| \leq 1\}$$

kompakt ist, d.h. falls die abgeschlossene Hülle von $K(B)$ kompakt ist.

Der Begriff der Kompaktheit wird z.B. in [90] detaillierter betrachtet.

Integralgleichungen sind Probleme auf (unendlich dimensionalen) Funktionenräumen. Bei den oben geschilderten Verfahren werden Lösungen $y \in C^N[a, b], n \in \mathbb{N}$, gesucht. Es kann also nur ein Fixpunktsatz angewandt werden, dessen Aussage auch in unendlich dimensionalen Räumen gilt:

Satz 3.1.3 (Schauderscher Fixpunktsatz) *Sei E ein normierter Raum, $U \subseteq E$ nichtleer, beschränkt und konvex, $A : U \to E$ stetig und kompakt. Gilt dann für den Abschluss $\overline{A(U)}$ des Bildes A(U)*

$$\overline{A(U)} \subseteq U,$$

so hat A einen Fixpunkt in U.

Der Beweis ist etwa in [6] zu finden.

Um zu zeigen, dass die Voraussetzung der Kompaktheit für Fredholmsche Integraloperatoren erfüllt ist, soll der folgende Satz angegeben werden, dessen Beweis in [51] ausgeführt wird.

Satz 3.1.4 *Ist D ein kompakter Definitionsbereich und gilt für den Kern der Integralgleichung (2.43) $k \in C(D \times D)$, so ist der zugehörige Integraloperator K kompakt auf $X = C(D)$.*

Verfahren 3.1.1 : Einzelverfahren

Für $n := 0...2N$:
  Berechne die Integrale der Basismonome $([t] - t_0)^n$
  für $[t] := [a, b]$ und $t_0 := mid([t])$ mit Verfahren 3.1.2.
Führe die Iteration nach Verfahren 3.1.3
  mit Startwert $F_0 := G$ und Ergebnis F durch.
Für $m := 0...N$:
  Führe die Iteration nach Verfahren 3.1.3
    mit Startwert $\alpha_m^0 := A_m$ und Ergebnis α_m durch.
Berechne die Einträge des (Intervall-)Gleichungssystems
  mit Verfahren 3.1.4.
Löse das (Intervall-)Gleichungssystem
  mit dem Verfahren von Rump (Verfahren 3.2.1).
Berechne die Lösungsfunktion mit Verfahren 3.1.5.

Somit ist der Schaudersche Fixpunktsatz in dem hier betrachteten Zusammenhang anwendbar und es existiert ein Fixpunkt in dem in Satz (2.5.3) betrachteten Iterationsverfahren, angewandt auf Funktionen über Intervallen, falls für zwei Iterierte Y^i, Y^{i+1} (als Einschließung der eigentlichen Iterierten y^i, y^{i+1}) gilt:

$$Y^{i+1} \subseteq Y^i.$$

3.1.4 Der Algorithmus für das Einzelverfahren

Es kann nunmehr der Algorithmus für das vorstehend beschriebene Verfahren angegeben werden.

Das Rahmenverfahren zur Lösung der linearen Fredholmschen Integralgleichung zweiter Art (2.43) ist in Verfahren 3.1.1 dargestellt. In den Verfahren 3.1.2, 3.1.3, 3.1.4 und 3.1.5 werden die folgenden im Rahmenverfahren referenzierten Verfahrensteile angegeben:

Verfahren 3.1.2 : Monomintegration

Gegeben: Intervall $[t] := [a, b]$, Entwicklungspunkt $t_0 := mid([t])$.
Ergebnis: Vektoren Ia, Ib der Integrale, ausgewertet an den Intervallgrenzen.

Für $i := 0...2N + 1$:
$pa_i := (a - t_0)^i$
$pb_i := (b - t_0)^i$
Für $i := 0...2N$:
$Ia_i := \frac{1}{i+1}\, pa_{i+1}$
$Ib_i := \frac{1}{i+1}\, pb_{i+1}$

Verfahren 3.1.3 : Iteration

Gegeben: Einschließung der Startfunktion $\mathfrak{Y}_0$.
$\epsilon > 0$ (für ϵ-Aufblähung $\bowtie$)
Abbruchindex $i_{max} \in \mathbb{N}$.
Ergebnis: Einschließung des Fixpunkts Q.

Setze $i := 0$.
Wiederhole
$\mathfrak{Y}_i := \mathfrak{Y}_i \bowtie \epsilon$
$\mathfrak{Y}_{i+1} := \mathfrak{Y}_0 + K_{\mathfrak{N}}(\mathfrak{Y}_i)$
$i := i + 1$
bis $\mathfrak{Y}_{i+1} \subseteq \mathfrak{Y}_i$ oder $i > i_{max}$.
Falls $i \leq i_{max}$
$Q := \mathfrak{Y}_{i+1}$
sonst Abbruch.

Verfahren 3.1.4 : Aufbau des Gleichungssystems

Gegeben: Iterationsergebnisse F und $\alpha_m, m = 0...N$.
Intervallauswertungen Ia, Ib, der Monomintegrale (aus Verfahren 3.1.2).
Die auftretenden Funktionen seien als Taylorpolynome gegeben, wobei folgende Notationen gelten:
$B_m(t), \alpha_m(t), 0 \leq m \leq N$, sowie $F(t)$: Anwendung von B_m, α_m, F, aufgefasst als Funktion, auf t.
$(B_m)_\mu, (\alpha_m)_\mu, (F)_\mu, 0 \leq m, \mu \leq N,$: μ-te Komponente von B_m, α_m bzw. F, aufgefasst als Vektor von Taylorkoeffizienten.

Ergebnis: Einträge der Matrix M und der rechten Seite R des Intervall-Gleichungssystems

$$(I - \lambda M)\Xi = R$$

mit

$$M = \begin{pmatrix} \int_a^b B_0(t)\alpha_0(t)\ dt & \cdots & \int_a^b B_0(t)\alpha_N(t)\ dt \\ \vdots & \ddots & \vdots \\ \int_a^b B_N(t)\alpha_0(t)\ dt & \cdots & \int_a^b B_N(t)\alpha_N(t)\ dt \end{pmatrix},\ R = \begin{pmatrix} \int_a^b B_0(t)F(t)\ dt \\ \vdots \\ \int_a^b B_N(t)F(t)\ dt \end{pmatrix}$$

Für $m := 0...N$:
  Für $n := 0...N$:
    // Berechnung des Elementes (m,n) der Matrix
    $M_{mn} := \sum_{\mu=0}^{N} \sum_{\nu=0}^{N} (B_m)_\mu \cdot (\alpha_n)_\nu (Ib_{\mu+\nu} - Ia_{\mu+\nu})$
  $R_m := \sum_{\mu=0}^{N} \sum_{\nu=0}^{N} (B_m)_\mu \cdot (F)_\nu (Ib_{\mu+\nu} - Ia_{\mu+\nu})$

Verfahren 3.1.5 : Berechnung der Lösungsfunktion

Gegeben: Lösung $\Xi := (\Xi_0, ..., \Xi_N)^T \in \mathbb{IR}^{N+1}$ des Intervall-Gleichungssystems.
Iterationsergebnisse F und $\alpha_m, m = 0...N$.
Die auftretenden Funktionen seien als Taylorausdrücke gegeben, wobei folgende Notationen gelten:
$(B_m)_n, (\alpha_m)_n, (F)_n, 0 \leq m, n \leq N,$: n-te Komponente von B_m, α_m bzw. F, aufgefasst als Vektor von Taylorkoeffizienten.

Ergebnis: Lösung Y der Integralgleichung als Taylorausdruck, mit $(Y)_m$ aufgefasst analog obiger Notation.

Für $m := 0...N$:
$(Y)_m := (F)_m + \sum_{n=0}^{N} \lambda \cdot (\alpha_m)_n \cdot \Xi_n$

- Berechnung der Integrale der Monome $(t - t_0)^n,\ n = 0...2N$ (Verfahren 3.1.2)
- Iteratives Verfahren (für nicht entartete Kerne) (Verfahren 3.1.3)
- Aufbau des linearen Gleichungssystems (Verfahren 3.1.4)
- Berechnung der Lösung der Integralgleichung aus der Lösung des linearen Gleichungssystems (Verfahren 3.1.5)

Die folgenden schon aus den bisherigen Beschreibungen bekannten Notationen werden in den Verfahren benutzt:

- $k = k_{\mathfrak{E}} + k_{\mathfrak{N}}$: Zerlegung des Kerns in einen entarteten und einen nicht entarteten Kern. (Bei Verwendung der Taylorentwicklung: $k_{\mathfrak{E}}$ enthält die Summanden der (Intervall-)Taylorentwicklung, $k_{\mathfrak{N}}$ die Restgliedterme)
- $K = K_{\mathfrak{E}} + K_{\mathfrak{N}}$: Operatoren zu k, $k_{\mathfrak{E}}$, $k_{\mathfrak{N}}$.

- $A_m, B_m, m = 0...N$: Einschließung der Funktionen des entarteten Kerns $k_{\mathfrak{E}}(s,t) = \sum_{i=0}^{N} a_i(s)b_i(t)$ [5]
- λ: Parameter λ der Integralgleichung $(I - \lambda K)Y = G$.
- G: Einschließung der rechten Seite

Gemäß den Bemerkungen in den vorstehenden Abschnitten, insbesondere 3.1.3, kann das Verfahren zur verifizierten Einschließung der Lösung verwendet werden, wenn alle Funktionen (gegeben in Taylordarstellung) als Funktionen über Intervallen aufgefasst werden.

3.1.5 Das Verfahren für Integralgleichungssysteme

Analog zum Verfahren für die einzelne Integralgleichung stellt sich auch das Verfahren für Integralgleichungssysteme dar. In Abschnitt 2.5.3 wurden bereits die Grundlagen zur Lösung von Integralgleichungssystemen angegeben. Der Begriff der Entartung wird dabei für ein Integralgleichungssystem komponentenweise auf die Elemente der Kernmatrix zurückgeführt. Damit wird auch die Darstellung

$$k = k_{\mathfrak{E}} + k_{\mathfrak{N}}$$

aus (3.1) auf Systeme übetragen, indem eine Darstellung der Kerne k^{ij} der Kernmatrix als

$$k^{ij} = k_{\mathfrak{E}}^{ij} + k_{\mathfrak{N}}^{ij},\ i,j = 1...N$$

möglich ist.

Zu den Kernen k^{ij}, $k_{\mathfrak{E}}^{ij}$, $k_{\mathfrak{N}}^{ij}$ werden durch

$$K^{ij}, K_{\mathfrak{E}}^{ij}, K_{\mathfrak{N}}^{ij} \tag{3.23}$$

wieder die zugehörigen Operatoren definiert und durch

$$\mathcal{K}, \mathcal{K}_{\mathfrak{E}}, \mathcal{K}_{\mathfrak{N}} \tag{3.24}$$

[5] Hier wird ein Indexbereich von 0 bis N gewählt, da dies der üblichen Indizierung von Taylorkoeffizienten entspricht.

werden Matrizen definiert, deren Elemente die $K^{ij}, K^{ij}_{\mathfrak{E}}, K^{ij}_{\mathfrak{N}},\ i,j = 1...N$ sind. Dann können, entsprechend der Aussagen für einzelne Integralgleichungen, die nicht entarteten Kerne des Kernsystems so gewählt werden, dass

$$|\lambda| \sum_{i,j=1}^{N} \|k^{ij}_{\mathfrak{N}}\| < 1 \tag{3.25}$$

gilt und somit

$$|\lambda| \|k^{ij}_{\mathfrak{N}}\| < 1,\ \ i,j = 1...N, \tag{3.26}$$

so dass für alle $k^{ij}_{\mathfrak{N}}$ lösende Transformationen

$$q^{ij} : (s,t) \mapsto q^{ij}(s,t),\ i,j = 1...N \tag{3.27}$$

mit zugehörigen Operatoren Q^{ij} im Sinne von (2.54) existieren. Analog zu den bisherigen Festlegungen werden dann auch hier die zugehörigen Matrizen

$$\mathcal{q} := (q^{ij})_{i,j=1...N}, \mathcal{Q} := (Q^{ij})_{i,j=1...N} \tag{3.28}$$

definiert.

Auch für das allgemeine Integralgleichungssystem erhalten wir damit eine Darstellung

$$\begin{aligned} \mathcal{y} - \lambda \mathcal{K}_{\mathfrak{N}} \mathcal{y} - \lambda \mathcal{K}_{\mathfrak{E}} \mathcal{y} &= \mathcal{g} & (3.29) \\ \Leftrightarrow \qquad \mathcal{y} - \lambda \mathcal{K}_{\mathfrak{N}} \mathcal{y} &= \mathcal{g} + \lambda \mathcal{K}_{\mathfrak{E}} \mathcal{y}. & (3.30) \end{aligned}$$

Mit den lösenden Transformationen (3.28) ergibt sich

$$\begin{aligned} \mathcal{y} &= \mathcal{g} + \lambda \mathcal{K}_{\mathfrak{E}} \mathcal{y} + \mathcal{Q}(\mathcal{g} + \lambda \mathcal{K}_{\mathfrak{E}} \mathcal{y}) & (3.31) \\ &= (\mathcal{g} + \mathcal{Q}\mathcal{g}) + \lambda(\mathcal{K}_{\mathfrak{E}} \mathcal{y} + \mathcal{Q}\mathcal{K}_{\mathfrak{E}} \mathcal{y}). & (3.32) \end{aligned}$$

Wir erhalten somit[6]

$$\begin{aligned} y(s) &= \overbrace{g(s) + \int_a^b q(s,t)g(t)dt}^{=:f(s)} \\ &\quad +\lambda \sum_{m=0}^{T} \underbrace{\left(a_m(s) + \int_a^b q(s,u)a_m(u)du \right)}_{=:c_m(s)} \\ &\quad \cdot \int_a^b b_m(t)y(t)dt. \end{aligned} \tag{3.33}$$

Dies stellt wieder ein Integralgleichungssystem mit einem *Vektor* von Inhomogenitäten f und einer Matrix entarteter Kerne dar, die durch die Einträge von c_m und b_m dargestellt werden. Es lässt sich also das Lösungsverfahren für Integralgleichungsysteme mit entarteten Kernen anwenden. Sowohl f als auch $c_m, m = 0...T$, sind dabei Lösungen von Integralgleichungsystemen nach dem iterativen Verfahren mit der lösenden Transformation q. Dabei ist zu beachten, dass f wie gefordert ein Vektor von Funktionen ist, bei $c_m, m = 0...T$ handelt es sich jedoch entsprechend der Definition der a_m, b_m (siehe (2.63), (2.64)) um Matrizen. Damit zerfällt für jedes $m = 0...T$ die Berechnung der Matrix c_m spaltenweise in $i = 1...N$ Integralgleichungssysteme mit den Lösungen

$$c_m^i = a_m^i(s) + \int_a^b q(s,t)a_m^i(t)dt. \tag{3.34}$$

3.1.6 Der Algorithmus für das Systemverfahren

Es kann nun der Algorithmus für das Systemverfahren angegeben werden, so wie es in Abschnitt 3.1.5 beschrieben wird.

Das Rahmenverfahren für Integralgleichungssysteme ist in Verfahren 3.1.6 dargestellt. In den Verfahren 3.1.7, 3.1.8, 3.1.9 und 3.1.10 werden, analog zum Einzelverfahren, die folgenden Verfahrensteile angegeben:

[6] T ist wie in (2.61) gegeben.

Verfahren 3.1.6 : Systemverfahren

Für $j := 1...N$
  Für $n := 0...2T$:
    Berechne die Integrale der Basismonome[a] $([t]^j - t_0^j)^n$
    für $[t]^j := [\alpha_j, \beta_j]$ und $t_0^j := mid([t]^j)$ mit Verfahren 3.1.7.
Führe die Iteration nach Verfahren 3.1.8
  mit Startwert $\mathcal{F}^0 := \mathcal{G}$ und Ergebnis $\mathcal{F}$ durch.
Für $m := 0...T$:
  Für $j := 1...N$:
    Führe die Iteration nach Verfahren 3.1.8
      mit Startwert $\mathcal{C}_m^{j\mathbf{0}} := \mathcal{A}_m^j$ und Ergebnis $\mathcal{C}_m^j$ durch.
Berechne die Einträge des (Intervall-)Gleichungssystems
  mit Verfahren 3.1.9.
Löse das (Intervall-)Gleichungssystem
  mit dem Verfahren von Rump (Verfahren 3.2.1).
Berechne die Lösung des Integralgleichungssystems mit Verfahren 3.1.10.

[a] Es wird nur nach t integriert.

- Berechnung der benötigten Monom-Integrale (Verfahren 3.1.7)
- Iteratives Systemverfahren (für nicht entartete Kerne) (Verfahren 3.1.8)
- Aufbau des linearen Gleichungssystems (Verfahren 3.1.9)
- Berechnung der Lösung des Integralgleichungssystems aus der Lösung des linearen Gleichungssystems (Verfahren 3.1.10)

Im Folgenden sollen noch einmal die wichtigsten Bezeichnungen für das Verfahren genannt werden:

- $N \in \mathbb{N}$: Ordnung des Integralgleichungssystems

Verfahren 3.1.7 : Monomintegration

Gegeben:	Intervalle $[t]^j := [\alpha_j, \beta_j]$, Entwicklungspunkte $t_0^j := mid([t]^j)$, $j = 1...N$.
Ergebnis:	Vektoren Ia^j, Ib^j, $j = 1...N$ der Integrale, ausgewertet an den Intervallgrenzen.

Für $j := 1...N$:
  Für $i := 0...2T+1$:
    $pa_i^j := (\alpha_j - t_0^j)^i, pb_i^j := (\beta_j - t_0^j)^i$
Für $j := 1...N$:
  Für $i := 0...2T$:
    $Ia_i^j := \frac{1}{i+1}\, pa_{i+1}^j, Ib_i^j := \frac{1}{i+1}\, pb_{i+1}^j$

Verfahren 3.1.8 : Iteration

Gegeben:	Einschließung des Startvektors $\mathfrak{Y}^0 := (\mathfrak{Y}_1^0, ..., \mathfrak{Y}_N^0)^T$ $\epsilon > 0$ (für ϵ-Aufblähung $\bowtie$) Abbruchindex $i_{max} \in \mathbb{N}$
Ergebnis:	Einschließung des Fixpunktvektors $Q := (Q_1, ..., Q_N)^T$

Setze $i := 0$.
Wiederhole
  $\mathfrak{Y}^i := \mathfrak{Y}^i \bowtie \epsilon$
  $\mathfrak{Y}^{i+1} := \mathfrak{Y}^0 + \mathcal{K}\mathfrak{n}(\mathfrak{Y}^i)$
  $i := i + 1$
bis $\mathfrak{Y}^{i+1} \subseteq \mathfrak{Y}^i$ oder $i > i_{max}$.
Falls $i \leq i_{max}$
  $Q := \mathfrak{Y}^{i+1}$
sonst Abbruch.

Verfahren 3.1.9 : Aufbau des Gleichungssystems

Gegeben: Iterationsergebnisse $\mathcal{F}$, $\mathcal{C}_m := (C_m^{ij})_{i,j=1\ldots N}, m = 0...T$. Auswertungen Ia^j, Ib^j, $j = 1...N$, der Monomintegrale. Notation der Taylorausdrücke analog zu Verfahren 3.1.4.

Ergebnis: Einträge der $N \cdot (T+1) \times N \cdot (T+1)$-Matrix $\mathcal{M}$ und der rechten Seite $\mathcal{R}$ des Intervall-Gleichungssystems

$$(I - \lambda\mathcal{M})\mathcal{X} = \mathcal{R}$$

mit

$$\mathcal{M} = \begin{pmatrix} \mathcal{M}_{00} & \cdots & \mathcal{M}_{0T} \\ \vdots & & \vdots \\ \mathcal{M}_{T0} & \cdots & \mathcal{M}_{TT} \end{pmatrix}, \mathcal{R} = \begin{pmatrix} \int_a^b \mathcal{B}_0(t)\mathcal{F}(t)\,dt \\ \vdots \\ \int_a^b \mathcal{B}_T(t)\mathcal{F}(t)\,dt \end{pmatrix}$$

und Teilmatrizen $\mathcal{M}_{mn}, 0 \leq m, n \leq T$ gegeben durch

$$\begin{aligned} \mathcal{M}_{mn} &:= \int_a^b \mathcal{B}_m(t)\mathcal{C}_n(t)\,dt \\ &= \begin{pmatrix} \sum_{k=1}^N \int_{\alpha_k}^{\beta_k} B_m^{1k}(t)C_n^{k1}(t)dt & \cdots & \sum_{k=1}^N \int_{\alpha_k}^{\beta_k} B_m^{1k}(t)C_m^{kN}(t)dt \\ \vdots & & \vdots \\ \sum_{k=1}^N \int_{\alpha_k}^{\beta_k} B_m^{Nk}(t)C_n^{k1}(t)dt & \cdots & \sum_{k=1}^N \int_{\alpha_k}^{\beta_k} B_m^{Nk}(t)C_m^{kN}(t)dt \end{pmatrix} \end{aligned}$$

Für $m := 0...T$:
  Für $n := 0...T$: *// Teilmatrix M_{mn}*
    Für $i := 1...N$:
      Für $j := 1...N$:
        $M_{mn}^{ij} := \sum_{k=1}^N \sum_{\mu,\nu=0}^T (B_m^{ik})_\mu \cdot (C_n^{kj})_\nu (Ib_{\mu+\nu}^j - Ia_{\mu+\nu}^j)$
  Für $i := 1...N$:
    $R_m^i := \sum_{k=1}^N \sum_{\mu,\nu=0}^T (B_m^{ik})_\mu \cdot (F^k)_\nu (Ib_{\mu+\nu}^i - Ia_{\mu+\nu}^i)$

Verfahren 3.1.10 : Lösung des Integralgleichungssystems

Gegeben:	Lösung $\mathcal{X} := (\Xi^1, ..., \Xi^N)^T$ des Intervall-Gleichungssystems mit $\Xi^i := (\Xi^i_0, ..., \Xi^i_T),\ i = 1...N$. Iterationsergebnisse $\mathcal{F}$, $\mathcal{C}_m := (C^{ij}_m)_{i,j=1...N}, m = 0...T$. Notation der Taylorausdrücke analog zu Verfahren 3.1.5.
Ergebnis:	Lösung $\mathcal{Y} := (Y_1, ..., Y_N)^T$ mit Taylorausdrücken $Y_i, i = 1...N$.

Für $m := 0...T$:
  Für $i := 1...N$:
    $(Y^i)_m := (F_i)_m + \sum_{j=1}^{N} \sum_{n=0}^{T} \lambda \cdot (C^{ij}_n)_m \cdot \Xi^j_n$

- $T \in \mathbb{N}$: Ordnung der Taylorentwicklung bei Darstellung der beteiligten Funktionen in Taylorform
- $\mathcal{k} := (k^{ij})_{i,j=1...N}$: Kernmatrix
- $\mathcal{K} := (K^{ij})_{i,j=1...N}$: Matrix der Kernoperatoren
- $k^{ij} = k^{ij}_{\mathfrak{E}} + k^{ij}_{\mathfrak{N}}$: Zerlegung der Kerne k^{ij} in einen entarteten und einen nicht entarteten Anteil. (Bei Verwendung der Taylorentwicklung: $k^{ij}_{\mathfrak{E}}$ enthält die Summanden der (Intervall-)Taylorentwicklung, $k^{ij}_{\mathfrak{N}}$ die Restgliedterme)
- $K^{ij} = K^{ij}_{\mathfrak{E}} + K^{ij}_{\mathfrak{N}}$: Operatoren zu k^{ij}, $k^{ij}_{\mathfrak{E}}$, $k^{ij}_{\mathfrak{N}}$.
- $\mathcal{K}_{\mathfrak{E}} := (K^{ij}_{\mathfrak{E}})_{i,j=1...N}$, $\mathcal{K}_{\mathfrak{N}} := (K^{ij}_{\mathfrak{N}})_{i,j=1...N}$: Matrizen der entarteten und nicht entarteten Kernoperatoren
- $[\alpha_i, \beta_i] \times [\alpha_j, \beta_j], i, j = 1...N$: Definitionsbereiche der Kerne k^{ij}
- $[\alpha_i, \beta_i], i = 1...N$: Definitionsbereiche der rechten Seiten und der Unbekannten.
- $A^i_m, B^{ij}_m, m = 0...T,\ i, j = 1...N$: Einschließung der Funktionen a^i_m, b^{ij}_m des entarteten Kerns $k^{ij}_{\mathfrak{E}}(s,t) = \sum_{m=0}^{T} a^i_m(s) b^{ij}_m(t)$

- $\mathcal{A}_m := \begin{pmatrix} A_m^1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & A_m^N \end{pmatrix}$, $\mathcal{B}_m := \begin{pmatrix} B_m^{11} & \cdots & B_m^{1N} \\ \vdots & \ddots & \vdots \\ B_m^{N1} & \cdots & B_m^{NN} \end{pmatrix}$.

  $\mathcal{A}_m^j, j = 1...N$, bezeichne dabei die j-te Spalte von A_m.
- $G^i, i = 1...N$: Einschließungen der rechten Seiten g^i
- $\mathcal{G} := \begin{pmatrix} G^1 \\ \vdots \\ G^N \end{pmatrix}$
- λ: Parameter λ des Integralgleichungssystems.

Gemäß der Bemerkungen in den vorstehenden Abschnitten, insbesondere 3.1.3, kann auch dieses Verfahren zur verifizierten Einschließung der Lösung verwendet werden, wenn alle Funktionen (als Elemente der betreffenden Vektoren und Matrizen, jeweils gegeben in Taylordarstellung) wie angegeben als Funktionen über Intervallen aufgefasst werden.

3.1.7 Lösung von Integralgleichungssystemen zur Verbesserung der Lösung von Integralgleichungen

Da der Zusammenhang zwischen der Lösung von Integralgleichungen und der Lösung von Integralgleichungssystemen, so wie sie in den vorstehenden Abschnitten beschrieben werden, für die in dieser Arbeit vorgenommenen Untersuchungen von entscheidender Bedeutung ist, soll dieser in diesem Abschnitt noch einmal explizit genannt werden.

Eine Integralgleichung lässt sich in ein Integralgleichungssystem zerlegen, indem der Kern als Matrix gleicher Kerne dargestellt wird, die über jeweils anderen Definitionsbereichen betrachtet werden. Dafür wird der Ursprungsdefinitionsbereich $[a, b]$ in N Definitionsbereiche $[\alpha_i, \beta_i]$ zerlegt. Entscheidend bei dieser Zerlegung ist die Eigenschaft, dass diese neuen Definitionsbereiche deutlich kleiner sind als der Ursprungsdefinitionsbereich. Damit ist zu erwarten, dass die Ergebnisse der Berechnungen, bei denen zur Einschließung der Lösung das gesamte Definitionsintervall eingesetzt wird, an Genauigkeit gewinnen, d.h. eine engere

Einschließung des Ergebnisses erzielt werden kann. Überdies liefern insbesondere die Taylorentwicklungen der auftretenden Funktionen engere Funktionswerteinschließungen, da nur Werte der jeweiligen Funktion nahe am Entwicklungspunkt der Taylorentwicklung untersucht werden und die Werte der Taylorentwicklung diejenigen der entwickelten Funktion nur in einer kleinen Umgebung des Entwicklungspunktes zufriedenstellend einschließen. Dies ist umso wichtiger, je geringer die gewählte Taylorordnung ist. Somit werden bereits durch den gewählten Berechnungsansatz die Überschätzungen des eingeschlossenen Ergebnisses bei großen Definitionsbereichen deutlich höher ausfallen als bei kleinen Definitionsbereichen.

Durch die Zerlegung einer Integralgleichung in ein Integralgleichungssystem ist es also möglich, eine genauere Einschließung der Lösung der Gleichung zu erhalten.

Gleichzeitig ist es möglich, die verwendete Ordnung der Taylorapproximation zu reduzieren, ohne dass damit die Lösung der Integralgleichung notwendigerweise unbrauchbar wird oder die Einschließung signifikant schlechter wird. Dies ist besonders im Hinblick auf die praktische Umsetzung der Verfahren von Interesse, da die Erhöhung der Taylorordnung mit stark erhöhtem Rechenaufwand einhergeht. Untersuchungen hierzu werden etwa in [68] angestellt.

Im Rahmen dieser Arbeit soll besonders untersucht werden, welches die praktischen Auswirkungen dieses Übergangs von der Integralgleichung zum Integralgleichungssystem sind: In welchem Umfang wird der Genauigkeitsgewinn durch den Übergang zum System in der Praxis sichtbar? „Lohnt“ sich der (bei gleicher Taylorordnung) erhöhte Aufwand beim Übergang zum System? Insbesondere wird untersucht, inwiefern bei einer Formulierung des Verfahrens für Parallelrechner die Rechenzeiten des Systemverfahrens reduziert und somit die Dimensionen der berechenbaren Systeme erhöht werden können. Daneben wird verglichen, ob sich das durch den Übergang zum System aufwändiger scheinende Verfahren in der Praxis gegen allgemeinere Verfahren für nichtlineare Integralgleichungen behaupten kann, sofern diese ebenfalls in parallelisierter Fassung verwendet werden können.

3.2 Lösungsverfahren für Teilprobleme

Ein entscheidendes Teilproblem der in den vorangehenden Abschnitten erläuterten Verfahren ist die verifizierte Lösung linearer Gleichungssysteme, die auch als unabhängige Problemstellung sowie in verschiedenen anderen Anwendungen der numerischen Mathematik eine zentrale Rolle spielt.

Es existieren verschiedene Methoden zur verifizierten Lösung linearer Gleichungssysteme. Im Zusammenhang dieser Arbeit soll jedoch nur das hier für die verifizierte Lösung linearer Gleichungssysteme verwendete Verfahren von *Rump* [113] angegeben werden. Eine detaillierte Auseinandersetzung mit Verfahren zur verfizierten Lösung linearer Gleichungssysteme ist etwa in [102] zu finden.

Im Anschluss an die Darstellung des Verfahrens von Rump wird noch ein Vorgehen zur Matrixinversion angegeben, ohne auf die Berechnung von *verifizierten* Lösungen einzugehen. Solche reellen Verfahren können verwendet werden, um in Rumps Verfahren Startwerte für die durchzuführende Iteration zu berechnen.

Bei der Betrachtung linearer Gleichungssysteme gehen wir hier von einem *Intervall*-Gleichungssystem aus, das wie folgt gegeben ist:

$$Ax = b, A \in [A], b \in [b],\ [A] \in \mathbb{IR}^{n\times n}, [x], [b] \in \mathbb{IR}^n, n \in \mathbb{N} \tag{3.35}$$

bzw. mit denselben Bezeichnungen

$$[A][x] = [b]. \tag{3.36}$$

Zu diesem Gleichungssystem ist eine Einschließung der Lösungsmenge

$$\Sigma([A],[b]) := \{x \in \mathbb{R}^n | Ax = b \text{ für ein } A \in [A],\ b \in [b]\} \tag{3.37}$$

gesucht.

Im Allgemeinen ist die exakte Lösungsmenge eines linearen Gleichungssystems kein Intervallvektor und kann unterschiedliche Gestalt annehmen (Beispiele finden sich z.B. in [102]). Daher wird eine Einschließung gesucht, die die eigentliche Lösungsmenge möglichst wenig überschätzt.

3.2.1 Rumps Verfahren zur verifizierten Lösung linearer Gleichungssysteme

Wie bei vielen anderen Problemstellungen ist auch im Hinblick auf lineare Gleichungssysteme einerseits die Existenz, andererseits aber auch die Eindeutigkeit einer Lösung von Interesse. Rump zeigt in [113], dass mit dem nachfolgend vorgestellten Verfahren erzielte Einschließungen der Lösungsmenge eines Intervall-Gleichungssystems diese Eigenschaften erzielt werden können.

Das iterative Verfahren von Rump ist eine dem Newton-Verfahren (vgl. z.B. [115] sowie Abschnitt 3.3) verwandte iterative Methode, deren Grundlage bereits von *Krawczyk* [75] angegeben wurde. Dabei entspricht die Suche nach der Lösung eines linearen Gleichungssystems $Ax = b$ der Suche nach einer Nullstelle der Funktion

$$f(x) = Ax - b.$$

Die zugehörige Iterationsvorschrift lautet dann zunächst

$$\begin{aligned} x_{k+1} &= x_k - R(Ax_k - b) \\ &= Rb + (I - RA)x_k, k = 0, 1, ... \end{aligned} \tag{3.38}$$

mit der Einheitsmatrix I und einer Näherungsinversen R zu A, da A^{-1} selbst im Allgemeinen nicht exakt bekannt ist. Der folgende Satz wurde von Rump [113] bewiesen:

Satz 3.2.1 *Seien $A \in [A] \in \mathbb{IR}^{n\times n}, b \in [b] \in \mathbb{IR}^n, R \in \mathbb{R}^{n\times n}$. Existiert für die Iteration aus (3.38) (verifiziert durchgeführt mit Intervallen $[x]_k \in \mathbb{IR}^n$) ein $k \in \mathbb{N}$ mit*

$$[x]_{k+1} \overset{\circ}{\subset} [x]_k,$$

so sind A und R regulär und es gibt eine eindeutig bestimmte Lösung x von $Ax = b$, die in $[x]_{k+1}$ liegt.

Es handelt sich um eine weitere Anwendung eines Fixpunktsatzes. Da hier aber endlichdimensionale reelle Zahlenräume und keine Funktionenräume

betrachtet werden, wird hier im Gegensatz zu Abschnitt 3.1.3 der Fixpunktsatz von *Brouwer* angewendet, der im Folgenden angegeben wird:

Satz 3.2.2 *Sei $f : \mathbb{R}^n \to \mathbb{R}^n$ stetig und X eine nichtleere, abgeschlossene, konvexe und beschränkte Menge. Gilt dann*

$$f(X) \subseteq X,$$

so hat f mindestens einen Fixpunkt in X.

Ist eine Näherungslösung $\tilde{x}$ des Gleichungssystems gegeben, so kann eine Verbesserung des Verfahrens durch den Übergang zum Residuum erreicht werden. Dazu bildet man $d = b - A\tilde{x}$ und löst dann das System $Ay = d$. Damit erhält man aus der anfänglichen Iteration die Iteration

$$y_{k+1} = \underbrace{R(b - A\tilde{x})}_{=:z} + \underbrace{(I - RA)}_{=:C} y_k, k = 0, 1, ... \tag{3.39}$$

bzw. mit Intervall-Daten

$$y_{k+1} = \underbrace{R([b] - [A]\tilde{x})}_{=:[z]} + \underbrace{(I - R[A])}_{=:[C]} y_k, k = 0, 1, ... \tag{3.40}$$

für den Fehler der Näherungslösung.

Eine erweiterte Version dieses Grund-Verfahrens für schlecht konditionierte Matrizen R ergibt sich, wenn man eine neue Näherungsinverse S zu RA berechnet, so dass wegen $A^{-1} = (RA)^{-1}R$ durch SR eine neue Näherungsinverse zu A mit vermutlich besserer Kondition gegeben wird, mit der die Iteration wiederholt werden kann. Diese kann zudem in der praktischen Implementierung mit einfachen Mitteln besonders genau als Summe zweier Matrizen $R_1 + R_2$ dargestellt werden. Diese Darstellung wird in der in Kapitel 5 vorgestellten Implementierung des Verfahrens gewählt. Der zweite Teil des Verfahrens wird somit in einer dieser Summe angepassten Verfahrensweise durchgeführt. Im Folgenden soll das Verfahren allerdings ohne explizite Angabe der hierfür notwendigen Modifikationen angegeben werden.

Verfahren 3.2.1 : Intervall-LGS-Löser

Seien $[A] \in \mathbb{IR}^{n \times n}$, $[b] \in \mathbb{IR}^n$ gegeben, $A := mid([A])$, $b := mid([b])$.

Berechne eine Näherungsinverse R von A.

Setze $\tilde{x} := R \cdot b$. **(*)**

Führe die Residuumsiteration

$d := b - A\tilde{x}$.

$\tilde{x} := \tilde{x} + R \cdot d$.

aus bis $\tilde{x}$ genau genug ist (oder Abbruch).

Berechne Einschließungen $[C], [z]$ für C und z aus (3.40).

Setze $[x]^{(0)} := [z]$. Führe die verifizierte Iteration

$[x]^{(i)} := [x]^{(i)} \bowtie \epsilon$ (ϵ-Aufblähung)

$[x]^{(i+1)} := [z] + [C][x]^{(i)}, i := 0, 1, ...$

aus bis $[x] := [x]^{(i+1)} \overset{\circ}{\subset} [x]^{(i)}$ (oder Abbruch).

Falls erfolgreich:

Setze $[x] := \tilde{x} + [x]$.

Sonst:

Berechne RA.

Berechne eine Näherungsinverse S von RA.

Berechne $R := SR$.

Wiederhole **(*)**.

Falls erfolgreich:

Setze $[x] := \tilde{x} + [x]$.

Sonst: Fehlgeschlagen.

Das Verfahren nimmt somit die in Verfahren 3.2.1 dargestellte Gestalt an.

Bemerkung 3.2.1 *Das Verfahren kann sowohl für reelle als auch für In-*

tervall-Eingabedaten ausgeführt werden. Im Falle von Intervall-Eingabedaten wird zur Berechnung der Näherungsinversen üblicherweise die reelle Mittelpunktsmatrix als Ausgangsmatrix gewählt, also $R \approx (mid[A])^{-1}$.

Ein paralleles Verfahren zur verifizierten Lösung von linearen Intervall-Gleichungssystemen, das auf der hier erläuterten Iteration basiert, wird in Kapitel 4 angegeben, seine Implementierung in Kapitel 5 vorgestellt.

3.2.2 Verfahren zur Matrixinversion

Bei der Durchführung des Verfahrens von Rump (siehe Abschnitt 3.2.1) werden Näherungsinverse von Punktmatrizen berechnet. Die Berechnung der Inversen einer $(n \times n)$-Matrix kann als lineares Gleichungssystem mit n rechten Seiten aufgefasst werden.

Ein mögliches Vorgehen zur Berechnung der Inversen einer Matrix ist die *LU-Zerlegung* mit anschließender *Vorwärts-Rückwärts-Substitution*, d.h. dem Lösen zweier Dreieckssysteme für n rechte Seiten. Dieses Vorgehen soll hier kurz geschildert werden. Da die Berechnung einer Näherungsinversen nicht den Hauptteil des Verfahrens von Rump darstellt, wird auf die Diskussion unterschiedlicher Verfahrensweisen und die Angabe von weiterführenden Details hier verzichtet. Einige Überlegungen zur Durchführung werden jedoch in Kapitel 4 angestellt, wo ein paralleler Algorithmus zur Matrixinversion angegeben wird.

Kennt man für $A := (a_{ij})_{i,j=1\ldots n} \in \mathbb{R}^{n \times n}$ eine Zerlegung

$$A = LU$$

mit einer oberen Dreiecksmatrix U und einer unteren Dreiecksmatrix L, so lässt sich ein System $Ax = LUx = b$ lösen, indem man nacheinander die beiden Dreieckssysteme

$$Ly = b \quad \text{und} \tag{3.41}$$

$$Ux = y \tag{3.42}$$

löst. Diese Systeme sind aber einfach durch direktes Auflösen nach der jeweils betrachteten Komponente der Unbekannten y bzw. x lösbar.

Verfahren 3.2.2 : LU-Zerlegung

Für $k := 1...n-1$:
  Für $s := k+1...n$:
    $a_{sk} = a_{sk}/a_{kk}$
  Für $j := k+1...n$:
    Für $i := k+1...n$:
      $a_{ij} := a_{ij} - a_{ik}a_{kj}$

Man gewinnt die LU-Zerlegung, indem man A durch Zeilen- oder Spaltenoperationen (Bilden von Linearkombinationen) in bekannter Weise auf obere Dreiecksgestalt bringt. Dabei lassen sich die Operationen ausdrücken durch Transformationsmatrizen $M_i, i = 1...n$, so dass gilt:

$$M_n \cdots M_1 A = U. \tag{3.43}$$

Die angegebene Multiplikation der Matrix A mit den Matrizen M_i von links entspricht dabei Zeilenumformungen, für Spaltenumformungen ergeben sich analoge Aussagen (vgl. [29]).

Aus dieser Darstellung erhält man die gesuchte Matrix L mit $A = LU$ offensichtlich durch

$$L = M_1^{-1} \cdots M_n^{-1}. \tag{3.44}$$

Das Entscheidende hieran ist, dass die Inversen M_i^{-1} sofort an den M_i ablesbar sind. Diese Eigenschaft wird z.B. in [38] dargestellt.

Eine algorithmische Beschreibung der LU-Zerlegung, bei der alle Berechnungen mit den Einträgen der Matrix A durchgeführt werden, so dass als Ergebnis die Matrix A zum Schluss eine untere Dreiecksmatrix L und eine obere Dreiecksmatrix U enthält, findet sich in Verfahren 3.2.2.

Hat man die Zerlegung $A = LU$ berechnet, so kann A^{-1} durch Vorwärts-Rückwärts-Subsitution bestimmt werden, indem man die Systeme

$$LUx = e_i, i = 1...n \tag{3.45}$$

für die Einheitsvektoren $e_i, i = 1...n$, aus denen sich die Einheitsmatrix zusammensetzt, löst.

Zusätzlich sind oft *Pivotstrategien* nötig, um die Durchführbarkeit des Verfahrens zu gewährleisten, so dass zusätzlich zu Linearkombinationen der Zeilen einer Matrix Zeilenvertauschungen vorgenommen werden. Auch diese lassen sich aber als Matrixmultiplikation mit der Ursprungsmatrix darstellen, und auch diese Matrizen haben direkt ablesbare Inverse, so dass das Ergebnis des so permutierten Systems schnell zurückgewonnen werden kann.

Im Kapitel 4 wird, wie oben bereits erwähnt, ein konkreter Algorithmus zur parallelen Berechnung der LU-Zerlegung angegeben.

3.3 Obermaiers Verfahren für nichtlineare Fredholmsche Integralgleichungen zweiter Art

Obermaier stellt in [106] ein Verfahren zur verifizierten Lösung nichtlinearer Fredholmscher Integralgleichungen zweiter Art vom Urysohntyp vor (siehe Definition 2.5.4). Hierbei wird die lineare Fredholmsche Integralgleichung zweiter Art (2.43) als Spezialfall mit erfasst. Das Verfahren ist daher von Interesse, um zu untersuchen, ob es als Beispiel eines Verfahrens für einen allgemeineren Typ von Integralgleichungen in der Praxis vergleichbare Ergebnisse liefert und insbesondere ähnlich effizient ist wie die spezialisierteren Verfahren für lineare Fredholmsche Integralgleichungen zweiter Art, denen das Hauptaugenmerk dieser Arbeit gilt.

Da das Verfahren nur zu Vergleichszwecken herangezogen wird, werden die theoretischen Ausführungen zu den einzelnen Teilen des Verfahrens nur grob gehalten. Details der jeweils nicht näher beleuchteten Punkte sind [106] zu entnehmen.

Das Verfahren besteht aus einer Anzahl von separaten Schritten, die der Übersicht halber zunächst insgesamt angegeben werden.

1. Berechnung einer Näherungslösung ω der gegebenen Fredholmschen Integralgleichung vom Urysohntyp. Dieser Schritt findet ohne Verifikation statt, so dass in nachfolgenden Schritten eine Einschließung des Fehlers vorgenommen werden muss.

2. Berechnung einer Konstanten δ, die den Approximationsfehler der in 1. berechneten Näherungslösung ω nach oben abschätzt. Dieser Schritt liefert zwar bereits eine Einschließung, jedoch ist die *Existenz* der Lösung noch nicht bekannt. Um mit Hilfe des Schauderschen Fixpunktsatzes 3.1.3 die Existenz der Lösung zu beweisen, muss daher ein geeigneter Operator konstruiert werden, der die Voraussetzungen des Satzes erfüllt. Dies geschieht in den nachfolgenden Schritten.

3. Bestimmung eines linearisierten Operators $I - \mathcal{K}$ am Punkt ω und Prüfung seiner Invertierbarkeit sowie Abschätzung der Norm der Inversen dieses Operators nach oben durch eine Konstante K. Die Inverse dieses Operators stellt, falls existent, den Operator dar, auf den der Schaudersche Fixpunktsatz angewendet wird.

4. Bestimmung einer monoton wachsenden Funktion G, für die für einen unten näher definierten Operator $\mathcal{G}$ die Abschätzung

   $$\|\mathcal{G}x\| \leq G(\|x\|)$$

   gilt.

5. Bestimmung einer Konstanten $\alpha \geq 0$ mit

   $$\delta \leq \frac{\alpha}{K} - G(\alpha).$$

Obermaier zeigt in [106], dass bei Durchführbarkeit dieses Vefahrens eine Lösung der betrachteten Integralgleichung existiert und die berechnete Konstante α die Norm der Differenz zwischen approximierter Lösung und exakter Lösung nach oben abschätzt.

Die oben genannten Schritte sollen nun beschrieben werden.

3.3.1 Die Näherungslösung

Zu Beginn des Verfahrens wird eine Näherungslösung für die nichtlineare Fredholmsche Integralgleichung vom Urysohntyp

$$y(s) - \int_a^b k(s,t,y(t))dt = g(s) \tag{3.46}$$

gesucht. Als Verfahren zur Gewinnung der Näherungslösung wird das Newton-Verfahren verwendet. Die Bestimmung einer Näherungslösung von (3.46) ist nämlich äquivalent zur Bestimmung einer Nullstelle für den Operator $\mathcal{F}$, der durch

$$(\mathcal{F}x)(s) := x(s) - \int_a^b k(s,t,x(t))dt - g(s) \tag{3.47}$$

gegeben ist.

Es wird keine Einschließung angestrebt, sondern lediglich eine Approximation der Nullstelle, deren Güte für den weiteren Verlauf des Verfahrens ausreichend ist.

Im allgemeinen Fall der nichtlinearen Integralgleichung (3.46) zerfällt die Berechnung einer Näherungslösung der Gleichung wiederum in folgende einzelne Schritte:

1. Bestimmung einer Startnäherung für das Newton-Verfahren, das bei beliebigen Startwerten möglicherweise nicht konvergiert.

2. Durchführung je eines Newton-Schrittes, der eine lineare Fredholmsche Integralgleichung liefert.

3. Approximation der Lösung dieser linearen Integralgleichung. Hierzu wird das *Nyström-Verfahren* verwendet.

Für Operatoren hat die Iteration des Newton-Verfahrens die Form

$$x_{k+1} = x_k - \mathcal{F}'(x_k)^{-1}\mathcal{F}(x_k). \tag{3.48}$$

Da sich ein derartiger Schritt aus der Vereinfachung der Darstellung

$$\mathcal{F}(x) = \mathcal{F}(x_0) + \mathcal{F}'(x_0)(x - x_0) + \mathcal{R}(x) \tag{3.49}$$

zu

$$\mathcal{F}(x) = \mathcal{F}(x_0) + \mathcal{F}'(x_0)(x - x_0) \tag{3.50}$$

ergibt, wobei $\mathcal{R}(x)$ für nahe an x_0 liegende x klein wird, kann man diesen Prozess auch als *Linearisierung* des Operators bezeichnen, bei dem aus einem nichtlinearen Operator ein diesen approximierender linearer Operator gewonnen wird.

Im Falle linearer Integralgleichungen liegt das Ergebnis des Newton-Verfahrens direkt vor, und das Vorgehen reduziert sich auf das Nyström-Verfahren.

Das Nyström-Verfahren

Das *Nyström-Verfahren* ist ein Verfahren zur Bestimmung einer Approximationslösung einer linearen Fredholmschen Integralgleichung zweiter Art (2.43) [103]. Das Verfahren basiert auf der Approximation des Integrals durch numerische Quadratur (Quadraturverfahren werden z.B. in [122] dargestellt). Dadurch erhält das Integral die diskretisierte Darstellung

$$\int_a^b k(s,t)y(t)dt \approx \sum_{j=1}^{n} w_{nj} k(s, t_{nj}) y(t_{nj}),\ n \in \mathbb{N} \tag{3.51}$$

mit *Quadraturknoten* t_{nj} und *Quadraturgewichten* $w_{nj},\ j = 1...n$.

Die Integralgleichung des Typs (2.43)[7]

$$y(s) - \int_a^b k(s,t)y(t)dt = g(s)$$

[7] Hierbei handelt es sich allgemein um das Ergebnis des Newton-Schrittes und im Hinblick auf den nichtlinearen Fall nicht notwendigerweise um die Funktionen g, k der Ausgangsintegralgleichung.

geht also über in

$$y(s) - \sum_{j=1}^{n} w_{nj} k(s, t_{nj}) y(t_{nj}) = g(s),$$

in Operatorschreibweise geht

$$(I - K)y = g$$

über in

$$(I - \mathcal{K}_n)y_n = g \tag{3.52}$$

mit dem *Nyström-Operator* $\mathcal{K}_n, n \in \mathbb{N}$, der definiert ist durch

$$(K_n y)(s) := \sum_{j=1}^{n} w_{nj} k(s, t_{nj}) y(t_{nj}).$$

Das Nyström-Verfahren besteht nun in der Lösung von (3.52) für wachsende Werte von $n \in \mathbb{N}$.

Gienger beweist dazu in [36] den folgenden Satz.

Satz 3.3.1 *Gegeben sei die Fredholmsche Integralgleichung (2.43). Existiert $(I - K)^{-1}$, so existiert ein $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ die Gleichung (3.52) nach y_n aufgelöst werden kann, und es gilt:*

$$\lim_{n \to \infty} y_n = y.$$

Die Berechnung der Lösung erfolgt durch Transformation der Gleichung (3.52) in ein lineares Gleichungssystem. Dafür definiert man eine Matrix

$$\mathbf{K}_n := w_{nj} k(t_{ni}, t_{nj})_{i,j=1\ldots n} \tag{3.53}$$

und einen Vektor

$$\mathbf{g}_n := (g(t_{ni}))_{i=1\ldots n} \tag{3.54}$$

und erhält so das lineare Gleichungssystem

$$(\mathbf{I} - \mathbf{K}_n)\mathbf{y}_n = \mathbf{g}_n \tag{3.55}$$

mit einem unbekannten Vektor $\mathbf{y}_n$.

Hierzu führt Gienger an:

Satz 3.3.2 *Sei* $\mathbf{K}_n$ *wie in (3.53) definiert.* $(\mathbf{I} - \mathbf{K}_n)$ *sei regulär und* z *sei Lösung von (3.55). Weiter sei*

$$Z(s) := g(s) + \sum_{j=1}^{n} w_{nj} k(s, t_{nj}) z_j.$$

Dann ist die Nyström-Gleichung

$$(I - \mathcal{K}_n) y_n = g$$

eindeutig lösbar und es gilt $y_n = Z$.

Der Beweis findet sich in [51].

[106] schildert zusätzlich Vorgehensweisen, um die Berechnung der Startnäherung mit dem oben beschriebenen Verfahren zu beschleunigen.

3.3.2 Die weiteren Teilschritte

Hat man eine Näherungslösung der nichtlinearen Integralgleichung berechnet, können die weiteren Teilschritte des Verfahrens durchgeführt werden.

Abschätzung des Approximationsfehlers

Die berechnete Näherungslösung der nichtlinearen Integralgleichung kann nun als

$$\omega(s) = g(s) + \sum_{j=1}^{n} w_{nj} k(s, t_{nj}, \omega_{nj}) \tag{3.56}$$

geschrieben werden (vgl. [106]). Damit hat der gesuchte Defekt die Darstellung

$$d(\omega)(s) = \sum_{j=1}^{n} w_{nj} k(s, t_{nj}, \omega_{nj}) - \int_a^b k(s, t, \omega(t)) dt. \tag{3.57}$$

Die Bestandteile dieser Darstellung werden nun in Steigungsarithmetik (siehe Abschnitt 2.4) ausgewertet, indem das Definitionsintervall in $m \in \mathbb{N}$ Teilintervalle unterteilt wird und auf diesen Intervallen Steigungstripel berechnet werden. Aus den Steigungstripeln werden Maxima und Minima und somit jeweils eine Majorante und eine Minorante für die vorkommenden Ausdrücke

$$\sum_{j=1}^{n} w_{nj} k(s, t_{nj}, \omega_{nj}) \quad \text{und} \tag{3.58}$$

$$k(s, t, \omega(t)) \tag{3.59}$$

bestimmt. Zur Berechnung des Integrals aus (3.57) werden dann für das unterteilte Integrationsintervall durch Abschätzung ebenso Majorante und Minorante bestimmt. Schließlich erhält man hieraus eine Minorante bzw. Majorante für den Gesamtausdruck (3.57), die auf den betrachteten Teilintervallen linear ist und somit nur an den Intervallgrenzen betrachtet werden muss, um Abschätzungen zu erhalten, die schließlich die Gesamtabschätzung für den Defekt der Näherungslösung ergeben.

Berechnung der Konstanten K

Es wird nun eine Konstante K gesucht, so dass der durch

$$\mathcal{L}x := (I - \mathcal{K})'(\omega)x =: (I - \tilde{\mathcal{K}})x \tag{3.60}$$

gegebene Operator ($\mathcal{K}$ als zu k gehöriger Kernoperator) mit [8]

$$\mathcal{L}x(s) = x - \int_a^b \frac{\partial k}{\partial \omega}(s, t, \omega(t))x(t)dt \tag{3.61}$$

eine durch K nach oben beschränkte Inverse besitzt. Diese Inverse wird dann als Operator verwendet, um den Fixpunktsatz von Schauder anwenden zu können. Ein Satz von Rall [107] liefert eine solche Abschätzung, wenn statt des eigentlichen Operators ein beschränkter linearer Operator $\tilde{\mathcal{L}}$ mit

$$\|\tilde{\mathcal{L}} - \mathcal{L}\| < \frac{1}{\|\tilde{\mathcal{L}}^{-1}\|}$$

[8] $\frac{\partial k}{\partial \omega}$ bezeichnet hier die partielle Ableitung nach der dritten Variablen.

gefunden werden kann. Obermaier approximiert hierzu den Kern $\tilde{k}$ des Operators $\tilde{\mathcal{K}}$ durch bilineare sog. „Hütchenfunktionen“ [51]. Für den approximierenden Operator müssen dann folgende Schritte unternommen werden:

1. Abschätzung der Norm des Approximationsfehlers $\|\tilde{\mathcal{L}} - \mathcal{L}\|$ durch eine Konstante K_1.

2. Nachweis, dass der approximierende Operator invertierbar ist und Bestimmung einer Konstanten K_2, die die Norm des invertierten Operators beschränkt.

Der erste Punkt wird wieder mit Hilfe einer Unterteilung des Definitionsbereichs und der Auswertung mittels Steigungsarithmetik auf den Teilintervallen des Definitionsbereiches untersucht. Auch hier können mit dieser Methode wieder Majoranten und Minoranten für die auftretenden Terme bestimmt werden.

Zur Bestimmung der Konstanten des zweiten Punktes wird die bilineare Approximation $\tilde{\mathcal{L}}$ in die Integralgleichung eingesetzt, so dass ein lineares Gleichungssystem entsteht, womit die Frage der Invertierbarkeit des Operators $\tilde{\mathcal{L}}$ auf die Frage nach der Regularität der Matrix des linearen Gleichungssystems übergeht.

Das Gleichungssystem kann etwa mit dem Verfahren von Rump (siehe Abschnitt 3.2.1) intervallarithmetisch gelöst werden, wobei man bei erfolgreich durchgeführter Iteration gleichzeitig den Beweis der Regularität der betrachteten Matrix erhält.

Unter Verwendung der so berechneten Inversen und der Komponenten des approximierenden Operators $\tilde{\mathcal{L}}$ kann schließlich eine Abschätzung K_2 und somit eine Gesamtabschätzung K berechnet werden.

Bestimmung einer Funktion G und einer Konstanten α

Im allgemeinen Fall der nichtlinearen Integralgleichung wird nun eine Funktion

$$G : \mathbb{R}_+ \to \mathbb{R}_+ \text{ mit } G(\alpha) \to 0 \ (\alpha \to 0)$$

gesucht, die den Operator $\mathcal{G}$, der durch

$$(\mathcal{G}x)(s) \quad := \quad \int_a^b \frac{\partial k}{\partial \omega}(s,t,\omega(t))x(t) \\ -\Big(k(s,t,\omega(t)+x(t)) - k(s,t,\omega(t))\Big)dt \quad (3.62)$$

gegeben ist, durch

$$\|\mathcal{G}x\| \le G(\|x\|), \;\; x \in C([a,b]) \quad (3.63)$$

abschätzt.

Für den allgemeinen Fall wird hier der Mittelwertsatz in Integralform angewendet, um wieder einen Ausdruck zu gewinnen, der mit Hilfe der Steigungsarithmetik geeignet ausgewertet werden kann, um eine Oberschranke zu erhalten.

Für den Fall linearer Integralgleichungen gilt jedoch $\mathcal{G}x = 0$, wie aus der Darstellung (3.62) ablesbar ist, wenn anstatt eines allgemeinen nichtlinearen Kernes der Kern $\tilde{k} := \tilde{k}(s,t)$ der linearen Integralgleichung eingesetzt wird:

$$\begin{aligned}(\mathcal{G}x)(s) \quad &= \quad \int_a^b \tilde{k}(s,t)x(t) \\ &\quad -\Big(\tilde{k}(s,t)(\omega(t)+x(t)) - \tilde{k}(s,t)\omega(t)\Big)dt \\ &= \quad 0.\end{aligned}$$

Damit kann sofort die Funktion $G \equiv 0$ als Abschätzung gewählt werden.

Nun wird eine Konstante α mit

$$\delta \le \frac{\alpha}{K} - G(\alpha) \quad (3.64)$$

gesucht. Für den linearen Fall erhält man mit $G \equiv 0$ sofort

$$\alpha := \delta\, K, \quad (3.65)$$

so dass Gleichung (3.64) erfüllt ist.

Zum Schluss soll noch einmal der Satz angegeben werden, der die Teilschritte für den allgemeinen Fall zusammenführt [106]:

Satz 3.3.3 *Sei $k \in C([a,b] \times [a,b] \times \mathbb{R})$, $\frac{\partial k}{\partial y}$ und $\frac{\partial^2 k}{\partial s\, \partial y}$ existent und stetig und $g \in C^1([a,b])$ sowie ω Näherungslösung der Fredholmschen Integralgleichung vom Urysohntyp*

$$y(s) - \int_a^b k(s,t,y(t))dt = g(s). \tag{3.66}$$

Sind dann die Voraussetzungen der vorstehend beschriebenen Schritte gegeben und lässt sich ein α nach dem beschriebenen Verfahren berechnen, so besitzt die Fredholmsche Integralgleichung vom Urysohntyp (3.66) eine Lösung in C([a,b]) und es gilt

$$\|\omega - x\| \leq \alpha. \tag{3.67}$$

4 Parallele Verfahren

In diesem Kapitel soll ein paralleles Verfahren zur Lösung linearer Fredholmscher Integralgleichungen und Integralgleichungssysteme vorgestellt werden, das auf den in Kapitel 3 angegebenen Verfahren basiert. Durch die Formulierung eines Verfahrens für Parallelrechner sollen die mathematischen Vorteile des Verfahrens für Systeme nutzbar gemacht werden, indem der Zeitaufwand für die Berechnung der Systemlösungen reduziert wird. Dabei werden die in Abschnitt 3.2 genannten Teilprobleme ebenso für Parallelrechner formuliert. Anschließend wird das von Obermaier formulierte parallele Verfahren in Kurzform angegeben.

Grundlagen des parallelen Rechnens bzw. paralleler Hard- und Software wurden in Abschnitt 2.6 bereits angegeben. Die Umgebung für die Umsetzung der Verfahren wird in Kapitel 5 erläutert. Dort werden auch die konkreten Implementierungen der hier beschriebenen Verfahren vorgestellt.

4.1 Aufwandsabschätzungen

Um die zu parallelisierenden Teile eines Verfahrens zu bestimmen, ist zunächst die Frage des Aufwands einzelner Programmteile zu klären. Hierzu bedarf es einer kurzen Erklärung des Begriff *Aufwand*. Da hier der relative Aufwand eines Algorithmus im Verhältnis zur Problemgröße $n \in \mathbb{N}$ (also etwa der Dimension der zu betrachtenden Matrizen oder anderer Eingabedaten), nicht jedoch der absolute Aufwand, der je nach verwendeter Hardware oder Größe der Eingabedaten höchst unterschiedlich ausfallen kann, betrachtet wird, ist es sinnvoll, eine entsprechende Notation zu wählen. Die aus der folgenden Definition hervorgehende Notation für den Aufwand von Algorithmen ist allgemein verbreitet (vgl. z.B. [118]):

Definition 4.1.1 *Sei $f : \mathbb{N} \to \mathbb{R}_+$ gegeben. Dann bezeichnet*

$$O(f(n)) \tag{4.1}$$

die Klasse der Funktionen $g : \mathbb{N} \to \mathbb{R}_+$, für die gilt:

$$\exists c \in \mathbb{R}\ \exists n_0 \in \mathbb{N} : \forall n > n_0 : g(n) < cf(n). \tag{4.2}$$

Die so definierten Klassen von Funktionen gestatten die Betrachtung typischer Klassen von (Maximal-) Aufwänden im Verhältnis zur Problemgröße n. Dabei wird der Zeitaufwand mit der Anzahl auszuführender Operationen identifiziert. Die Aussage *Der Algorithmus hat eine Komplexität von $O(n^2)$* bedeutet also, dass der Zeitaufwand des betrachteten Algorithmus maximal proportional zum Zeitaufwand eines Algorithmus ist, der bei Problemgröße n n^2 Operationen ausführt.

Bemerkung 4.1.1 *Es ist explizit zu erwähnen, dass sich die obige Definition nur auf das asymptotische Verhalten einer Funktion bzw. eines Algorithmus für $n \to \infty$ bezieht. Bei der Betrachtung kleiner Problemgrößen sind die mit dieser Notation getroffenen Aussagen oft unbrauchbar, da konstante Faktoren grundsätzlich außer Betracht gelassen werden.*

Bemerkung 4.1.2 *Die Identifikation des Zeitaufwands mit der Anzahl auszuführender Operationen ist eine stark vereinfachende Annahme. Die Benutzung obiger Notation ist also bei der Betrachtung von Problemen, für die Details einzelner Operationen von Bedeutung sind, nicht anwendbar.*

Folgende Beispiele sind typische Vertreter der Aufwandsklassen $O(n)$, $O(n^2)$ und $O(n^3)$.

Beispiel 4.1.1 Seien $A := (a_{ij})_{i,j=1\ldots n}$, $B := (b_{ij})_{i,j=1\ldots n} \in \mathbb{R}^{n\times n}$ und $v := (v_i)_{i=1\ldots n}$, $w := (w_i)_{i=1\ldots n} \in \mathbb{R}^n$.

1. Skalarprodukt: Komplexität $O(n)$

$$v^T w := \sum_{i=1}^{n} v_i w_i$$

2. Matrix-Vektor-Multiplikation: Komplexität $O(n^2)$

$$Av := (\sum_{j=1}^{n} a_{ij} v_j)_{i=1\dots n}$$

3. Matrix-Matrix-Multiplikation: Komplexität $O(n^3)$.

$$AB := (\sum_{j=1}^{n} a_{ij} b_{jk})_{i,k=1\dots n}$$

Das Gegenstück zur Beschreibung des Maximalaufwands $O(f(n))$ bildet der *Minimalaufwand* $\Omega(f(n))$.

Definition 4.1.2 *Sei* $f : \mathbb{N} \to \mathbb{R}_+$ *gegeben. Dann bezeichnet*

$$\Omega(f(n)) \tag{4.3}$$

die Klasse der Funktionen $g : \mathbb{N} \to \mathbb{R}_+$, *für die gilt:*

$$\exists c > 0, c \in \mathbb{R},\ \exists n_0 \in \mathbb{N} : \forall n > n_0 : g(n) > cf(n). \tag{4.4}$$

Bemerkung 4.1.3 *Im Allgemeinen wird die Bezeichnung* $O(f(n))$ *nur dann verwendet, wenn der entsprechende Maximal-Aufwand auch tatsächlich erreicht wird, auch wenn nach Definition beispielsweise die Menge der* $O(1)$*-Funktionen Teilmenge der* $O(n)$*-Funktionen ist etc. Entsprechendes gilt für die Verwendung der Bezeichnung* $\Omega(f(n))$*. Auf die Einführung der im Zusammenhang der Aufwandsabschätzung geläufigen Bezeichnung* $\Theta(f(n))$ *wird hier verzichtet.*

Bemerkung 4.1.4 *In ähnlicher Weise können Maximalaufwand und Minimalaufwand auch für mehrere Eingangsgrößen definiert werden. Allerdings werden damit Aussagen über den Aufwand uneindeutiger, da zusätzlich zur Abhängigkeit von den Problemgrößen das Verhältnis der Problemgrößen untereinander eine Rolle spielt und möglicherweise keine eindeutige Beziehung zwischen verschiedenen Klassen hergestellt werden*

kann. Die nachfolgenden Aufwandsbetrachtungen können also bei Vorliegen mehrerer Eingangsparameter zum Teil nur eingeschränkte Aussagen liefern.

4.2 Paralleles Verfahren zur Lösung linearer Fredholmscher Integralgleichungssysteme zweiter Art

Die Idee des Verfahrens von Klein [72] zur Lösung linearer Fredholmscher Integralgleichungssysteme ist die genauere Lösbarkeit von Integralgleichungen, wenn man den Definitionsbereich in Teilbereiche zerlegt und die Integralgleichung als System von Integralgleichungen löst. Gleichzeitig hofft man auf eine effizientere Ausführung des Verfahrens, wenn auf eine starke Erhöhung der Taylorordnung verzichtet werden kann. Dennoch bedeutet die Ausführung des Verfahrens mit entsprechenden Parametern eine hohe Rechenlast, so dass der Erfolg des Systemverfahrens in [72] nur mit kleinen Parameterwerten getestet werden konnte. Erst mit einem parallelen Verfahren ist man in der Lage, den Nutzen des Systemverfahrens auch für größere Parameterwerte beurteilen zu können.

Es sollen zunächst die Teile des Verfahrens bestimmt werden, die

a) sinnvoll parallelisiert werden können und

b) deren Aufwand einen relevanten Anteil am Gesamtverfahren ausmacht

Nur für Verfahrensteile, die beide Bedingungen erfüllen, ist eine Parallelisierung sinnvoll und möglich.

Hierzu soll Verfahren 3.1.6 noch einmal angegeben werden, wobei alle Programmteile markiert sind, deren Aufwand besonders relevant ist (siehe Verfahren 4.2.1).

Verfahren 4.2.1 : Systemverfahren

Für $j := 1...N$
  Für $n := 0...2T$:
    Berechne die Integrale der Basismonome[a] $([t]^j - t_0^j)^n$
    für $[t]^j := [\alpha_j, \beta_j]$ und $t_0^j := mid([t])$ mit Verfahren 3.1.7.
Führe die Iteration nach Verfahren 3.1.8
  mit Startwert $\mathcal{F}^0 := \mathcal{G}$ und Ergebnis $\mathcal{F}$ durch.

Für $m := 0...T$: (A)
  Für $j := 1...N$:
    Führe die Iteration nach Verfahren 3.1.8
      mit Startwert $\mathcal{C}_m^{j\mathbf{0}} := \mathcal{A}_m^j$ und Ergebnis $\mathcal{C}_m^j$ durch.

Berechne die Einträge des (Intervall-)Gleichungssystems (B)
  mit Verfahren 3.1.9.

Löse das (Intervall-)Gleichungssystem (C)
  mit dem Verfahren von Rump (Verfahren 3.2.1).

Berechne die Lösung des Integralgleichungssystems mit Verfahren 3.1.10.

[a] Es wird nur nach t integriert.

Der Aufwand in Teil (A) (Iterationen) hängt (wie auch der Aufwand der weiteren Teile) vom Verhältnis zwischen den Parametern T und N sowie von der Anzahl der nötigen Iterationsschritte ab. Da aber für beide Parameter (Taylorordnung und Systemordnung) Tests durchgeführt werden sollen, gleichzeitig jedoch $T < N$ angestrebt wird, kann der Aufwand grob als $O(TN) \approx O(N^2)$, mutlipliziert mit dem Aufwand für die Durchführung einer Iteration, angesehen werden. Bei der Iteration wird in jedem Schritt der Integraloperator des Integralgleichungssystems angewendet, so dass die Iteration zumindest den Aufwand $\Omega(N^2)$ besitzt. Die Parallelisierung dieses Teils erscheint somit angebracht.

Beim Aufbau des Gleichungssystems (incl. Berechnung der angegebenen

Integrale) in Teil (B) werden $(T+1)^2 N^2$ Elemente einer Matrix berechnet, die ihrerseits Berechnungen mit Aufwand $O(T^2 N)$ erfordern. Auch dieser Aufwand rechtfertigt die Parallelisierung.

Schließlich wird in Teil (C) das lineare Gleichungssystem mit der obigen Matrix mit dem Verfahren von Rump gelöst, das zunächst die Berechnung einer reellen approximierten Inversen und dann die Intervall-Newton-Iteration umfasst und damit ohne Einbezug der benötigten Zahl von Iterationsschritten bereits mehrere Teilschritte des Aufwands $O((TN)^3)$ beinhaltet. Dies ist ein signifikanter Teil des Aufwands des Gesamtverfahrens und somit in eine effiziente Parallelisierung einzubeziehen.

Die Parallelisierung der folgenden Teile dagegen erscheint im Vergleich zu den oben angegebenen Teilen nachrangig:

- Berechnung der Integrale der Basismonome $(t - t_0^j)^n$
- Berechnung des Vektors der Lösungsfunktionen
- Taylorentwicklungen der zu Grunde liegenden Funktionen

Die ersten beiden dieser Teile erfordern lediglich quadratischen Aufwand, während für die potentiell aufwändigeren Taylorentwicklungen die praktischen Tests zeigten, dass auch hier der tatsächliche Aufwand weit hinter den genannten Verfahrensteilschritten zurückbleibt[1].

Im Folgenden sollen Parallelisierungen der oben identifizierten Verfahrensteile (A) und (B) angegeben werden. Das Verfahren zu Teil (C), d.h. der Lösung des linearen Gleichungssystems, ist als Teilverfahren in Abschnitt 4.3 zu finden.

Parallelisierung der Iterationen

Da die Iterationen

$$\mathcal{C}_m^{j,0} := \mathcal{A}_m^j; \;\; \mathcal{C}_m^{j,i+1} := \mathcal{A}_m^j + \mathcal{K}\mathfrak{n}\,\mathcal{C}_m^{j,i}, \;\; i = 0, 1, ... \tag{4.5}$$

für $m = 0...T$ und $j = 0...N$ eigenständig sind, können die $T \cdot N$ Iterationsverfahren auf die zur Verfügung stehenden Prozesse verteilt werden

[1] Da in den durchgeführten Tests in Kapitel 6 kein signifikanter zeitlicher Aufwand durch die Taylorentwicklungen festgestellt werden konnte, werden keine diesbezüglichen Testwerte angegeben.

und ohne zwischengelagerte Kommunikation dort eigenständig beendet werden. Für die Iterationen erscheint dies als einfacher aber sinnvoller Ansatz zur Durchführung, da hiermit zusätzliche Kommunikation vermieden werden kann. Die Aufteilung einzelner Iterationen auf mehrere Prozesse hätte nach jedem Iterationsschritt das Versenden von Ergebnissen bzw. Teilergebnissen zur Folge.

Bei dieser Aufteilung wird für jeden Prozess die vollständige Information über den zugehörigen Kernoperator $\mathcal{K}_{\mathfrak{N}}$ benötigt. In Kapitel 6 wird näher untersucht, inwiefern das Kriterium Speicherplatzverbrauch bei den Praxistests der vorgestellten Implementierungen zum Tragen kommt.

Bei der Verteilung der Iterationen auf die Prozesse kann entweder gleichmäßig vorgegangen werden oder die Verteilung kann aktiv erfolgen, indem jeder Prozess nach Beendigung der aktuellen Iteration vom Masterprozess die nächste Iteration zugewiesen bekommt. Dieses Verfahren würde jedoch ein aktives Management der zu bearbeitenden Iterationen erfordern, was einer Zunahme des Verwaltungs- und Kommunikationsaufwands gleichkäme. Daher ist diese Verteilung nur dann vorzuziehen, wenn nachweislich starke Differenzen in der Rechenlast bestehen, d.h. wenn durch eine angepasste Verteilung der Berechnungen eine hohe Aufwandsreduktion zu erwarten ist. Dies kann der Fall sein, wenn Iterationen mit einer sehr hohen Zahl von Iterationsschritten durchgeführt werden und gleichzeitig parallel durchgeführte Iterationen nach einer sehr geringen Zahl von Schritten abbrechen, oder wenn die Struktur der gleichzeitig zu bearbeitenden Aufgaben sehr heterogen ist. Im Falle der hier betrachteten Iterationen ist letzteres nicht gegeben, und die Zahl der Iterationen wird auf eine kleine Maximalzahl begrenzt.

Parallelisierung der Berechnung der Matrix des Gleichungssystems

Auch die Erstellung der Matrix für das zu lösende lineare Gleichungssystem ist gut parallelisierbar, indem die zu bestimmenden Einträge jeweils von unterschiedlichen Prozessen berechnet werden. Durch Umordnen der Schleifen bei der Berechnung kann erreicht werden, dass jeweils nur eine Zeile der Matrix $\mathcal{C}$ der Iterationsergebnisse gleichzeitig für die Berechnung benötigt wird. Die anderen Datenobjekte müssen jeweils vollständig

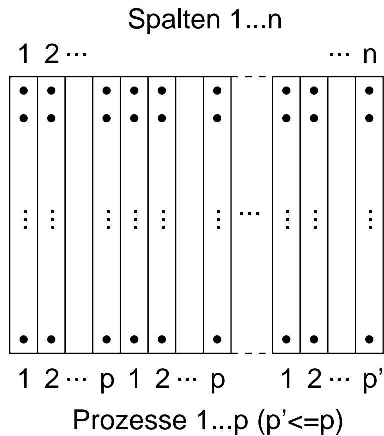

Abbildung 4.1: Spaltenzyklische Verteilung

vorliegen, wenn angestrebt wird, die Berechnung eines einzelnen Elements der Matrix jeweils vollständig einem einzelnen Prozess zuzuordnen.

Da im nachfolgenden Schritt das lineare Gleichungssystem mit der zu erstellenden Matrix gelöst wird, stellt sich insbesondere die Frage, ob die Verteilung der zu berechnenden Elemente auf die Prozesse für die nachfolgenden Berechnungen vorteilhaft erfolgen kann. Da das Verfahren von Rump mit der Inversion der betrachteten Matrix startet, sollte die Verteilung bei der Erstellung der Matrix also so erfolgen, dass eine erneute Verteilung der Matrixspalten nicht mehr nötig ist.

Wird also etwa eine spaltenzyklische Verteilung der Spalten für die Matrixinversion angestrebt, so wie es in der vorliegenden Implementierung der Fall ist, so sollte die Verteilung der zu berechnenden Matrixelemente ebenso spaltenzyklisch erfolgen (siehe Abbildung 4.1).

Damit kann man die parallele Verfahrensbeschreibung angeben. Das Rahmenverfahren zur parallelen Lösung eines linearen Fredholmschen Integralgleichungssystems zweiter Art ist in Verfahren 4.2.2 dargestellt.

Verfahren 4.2.2 : Paralleles Systemverfahren

Sei p_0 der Masterprozess, p_{akt} der aktuelle Prozess.

Für $j := 1...N$
  Für $n := 0...2T$:
    Berechne die Integrale der Basismonome $([t]^j - t_0^j)^n$
    für $[t]^j := [\alpha_j, \beta_j]$ und $t_0^j := mid([t])$ mit Verfahren 3.1.7.
Führe die Iteration nach Verfahren 3.1.8
  mit Startwert $\mathcal{F}^0 := \mathcal{G}$ und Ergebnis $\mathcal{F}$ durch.

Bestimme die von p_{akt} auszuführende Menge von Iterationen MyI. (A)
Für $m := 0...T$:
  Für $j := 1...N$:
    Falls die Iteration zu MyI gehört:
      Führe die Iteration nach Verfahren 3.1.8
        mit Startwert $\mathcal{C}_m^{j\mathbf{0}} := \mathcal{A}_m^j$ und Ergebnis $\mathcal{C}_m^j$ durch.
      Sende das Ergebnis an p_0.

Berechne die Einträge des (Intervall-)Gleichungssystems (B)
  mit Verfahren 4.2.3.

Löse das (Intervall-)Gleichungssystem (C)
  mit Verfahren 4.3.2.

Berechne die Lösung des Integralgleichungssystems mit Verfahren 3.1.10.

Die Verfahren zur Berechnung der benötigten Monomintegrale (Verfahren 3.1.7), zur Ausführung einer einzelnen Iteration (Verfahren 3.1.8) und zur Berechnung der Lösung des Integralgleichungssystems aus der Lösung des linearen Gleichungssystems (Verfahren 3.1.10) bleiben unverändert aufgrund ihres geringen Anteils an der Gesamtkomplexität des Programms und werden in Verfahren 4.2.2 erneut referenziert.

Verfahren 4.2.3 : Paralleler Aufbau des Gleichungssystems

Sei p_0 der Masterprozess, p_{akt} der aktuelle Prozess.

Gegeben: Iterationsergebnisse $\mathcal{F}$, $\mathcal{C}_m := (C_m^{ij})_{i,j=1...N}, m = 0...T$. Auswertungen Ia^j, Ib^j, $j = 1...N$, der Monomintegrale. Notation der Taylorausdrücke analog zu Verfahren 3.1.4.

Ergebnis: Einträge der $N \cdot (T+1) \times N \cdot (T+1)$-Matrix $\mathcal{M}$ und der rechten Seite $\mathcal{R}$ des Intervall-Gleichungssystems

$$(I - \lambda\mathcal{M})\mathcal{X} = \mathcal{R}$$

mit $\lambda, \mathcal{M}, \mathcal{R}$ und Teilmatrizen $\mathcal{M}_{mn}, 0 \leq m, n \leq T$ wie in Verfahren 3.1.9.

Bestimme die Menge der von p_{akt} zu berechnenden Spalten MyC
 der Ergebnismatrix.
Für $m := 0...T$:
 Für $n := 0...T$:
  Für $i := 1...N$:
   Für $j := 1...N$:
    Falls $(n \cdot N + j) \in MyC$:
     // Spalte j von Teilmatrix $\mathcal{M}_{mn}$
     // ist Spalte $(n \cdot N + j)$ der Gesamtmatrix
     $\mathcal{M}_{mn}^{ij} := \sum_{k=1}^{N} \sum_{\mu,\nu=0}^{T} (B_m^{ik})_\mu \cdot (C_n^{kj})_\nu$
      $\cdot(Ib_{\mu+\nu}^j - Ia_{\mu+\nu}^j)$
 Für $i := 1...N$:
  $R_m^i := \sum_{k=1}^{N} \sum_{\mu,\nu=0}^{T} (B_m^{ik})_\mu \cdot (F^k)_\nu (Ib_{\mu+\nu}^i - Ia_{\mu+\nu}^i)$
$Füri := 0...(T+1) \cdot N :$
 Falls $i \in MyC$:
  Sende Spalte $\mathcal{M}_i$ an p_0.

Die folgenden Verfahrensteile werden für das parallele Systemverfahren neu formuliert:

- Paralleler Aufbau des linearen Gleichungssystems (Verfahren 4.2.3)
- Paralleler (Intervall-)Löser für lineare Gleichungssysteme (Verfahren 4.3.2, siehe Abschnitt 4.3.1)

Zu Verfahren 4.2.2 sollen noch die nachfolgenden Bemerkungen hinzugefügt werden.

Bemerkung 4.2.1 *Die Verteilung der von den Prozessen benötigten Daten wird in den Algorithmen nicht explizit aufgeführt.*

Bemerkung 4.2.2 *Nach der Berechnung der Spalten $\mathcal{M}_j$ der Gesamtmatrix liegen die Spalten zyklisch verteilt vor, so dass das Verfahren zur Lösung des linearen Gleichungssystems die Spalten nicht noch einmal für den ersten Verfahrensschritt verteilen muss.*

Bemerkung 4.2.3 *Die Bestimmung der zu berechnenden Spalten bzw. Elemente kann vom aktuellen Prozess selbst vorgenommen werden, wenn die Zuordnung zum aktuellen Prozess eindeutig in Abhängigkeit von den vorliegenden Informationen darstellbar ist (z.B. als $j\%p = 0$ bei p Prozessen)*

Bemerkung 4.2.4 *Die in Verfahren 4.2.2 dunkel hinterlegten Bereiche sind die oben beschriebenen parallelisierten Verfahrensteile; die hell hinterlegten Verfahrensteile können von allen Prozessen ausgeführt werden. Damit wird die Versendung dieser Daten überflüssig. Dies ist bei für das Gesamtverfahren unerheblichem Aufwand möglich.*

4.3 Parallele Teilverfahren

In diesem Abschnitt sollen nun die im vorangehenden Abschnitt noch außen vor gelassenen parallelen Verfahrensteile vorgestellt werden. Dabei handelt es sich um die parallele Lösung linearer Gleichungssysteme aufbauend auf dem Verfahren von Rump, und um die parallele Matrixinversion und Matrixmultiplikation als Teilschritte des Verfahrens von Rump bzw. im Verfahren auftretende Operationen.

4.3.1 Paralleles Verfahren zur verifizierten Lösung linearer Gleichungssysteme

Das Verfahren von Rump wurde bereits in Abschnitt 3.2.1 erläutert. Da das Verfahren vor allem Matrix- und Vektor-Operationen ausführt, handelt es sich bei den einzelnen ausgeführten Operationen meist um Operationen mit Aufwand $O(n)$, $O(n^2)$ oder $O(n^3)$, gemessen an der Matrixdimension $n \in \mathbb{N}$. Für große Werte von n sind die Operationen mit dem Aufwand $O(n)$ und $O(n^2)$ gegenüber den Operationen mit Aufwand $O(n^3)$ vernachlässigbar. Daher sollen hier Parallelisierungen für diejenigen Verfahrensteile angegeben werden, die Operationen mit Aufwand $O(n^3)$ beinhalten.

Für die im Verfahren enthaltenen Iterationen (Residuumsiteration und verifizierte Iteration) gilt, dass sie

- zwar mit Intervall-Werten rechnen, aber nur höchstens Operationen mit Aufwand $O(n^2)$ ausführen.
- nur eingeschränkt parallelisierbar sind, da in jedem Schritt das Ergebnis des vorangehenden Schrittes benötigt wird.

Daher ist eine Parallelisierung der Iterationen nur dann sinnvoll, wenn besondere Gründe hierfür vorliegen. Die später dokumentierten Tests zeigen, dass das parallele Verfahren auch bei serieller Ausführung dieser Verfahrensteile gute Ergebnisse erbringen.

In Verfahren 4.3.1 sind die zu parallelisierenden Teile mit Operationen des Aufwands $O(n^3)$ dunkel hinterlegt dargestellt.

Bei den zu parallelisierenden Operationen handelt es sich im Einzelnen um die Folgenden:

1. Inversion einer reellen Matrix.
2. Matrixmultiplikation AB zweier Matrizen A, B.
3. Berechnung von Ausdrücken $(I - AB)$ für Matrizen A, B und die Einheitsmatrix I.

Zusätzlich werden in der praktischen Implementierung auch Ausdrücke der folgenden Form von Interesse sein:

Verfahren 4.3.1 : Intervall-LGS-Löser

Seien $[A] \in \mathbb{IR}^{n\times n}$, $[b] \in \mathbb{IR}^{n}$ gegeben, $A := mid([A])$, $b := mid([b])$.

Berechne eine Näherungsinverse R von A.

Setze $\tilde{x} := R \cdot b$. **(*)**

Führe die Residuumsiteration

$d := b - A\tilde{x}$.

$\tilde{x} := \tilde{x} + R \cdot d$.

aus bis $\tilde{x}$ genau genug ist (oder Abbruch).

Berechne Einschließungen $[C], [z]$ für C und z aus (3.40).

Setze $[x]^{(0)} := [z]$. Führe die verifizierte Iteration

$[x]^{(i)} := [x]^{(i)} \bowtie \epsilon$ (ϵ-Aufblähung)

$[x]^{(i+1)} := [z] + [C][x]^{(i)}, i := 0, 1, ...$

aus bis $[x] := [x]^{(i+1)} \overset{\circ}{\subset} [x]^{(i)}$ (oder Abbruch).

Falls erfolgreich:

Setze $[x] := \tilde{x} + [x]$.

Sonst:

Berechne RA.

Berechne eine Näherungsinverse S von RA.

Berechne $R := SR$.

Wiederhole **(*)**.

Falls erfolgreich:

Setze $[x] := \tilde{x} + [x]$.

Sonst: Fehlgeschlagen.

4. Berechnung von Ausdrücken $(I - A_1B_1 - A_2B_2)$ für Matrizen A_1, A_2, B_1 und B_2 und die Einheitsmatrix I.

5. Berechnung von Ausdrücken $(A_1B_1 - A_2B_2)$ für Matrizen A_1, A_2, B_1 und B_2.

Die entsprechenden Betrachtungen werden in Kapitel 5 angestellt.

Alle oben genannten multiplikativen Ausdrücke können in verschiedenen Varianten für unterschiedliche Datentypen, d.h. insbesondere für reelle Matrizen und Intervallmatrizen in unterschiedlichen Kombinationen auftreten.

Die Ausdrücke 3.-5. sind Abwandlungen der Matrixmultiplikation, die zwar aus Effizienzgründen getrennt implementiert werden können, für die es aber zur Parallelisierung keiner zusätzlichen Verfahrensangabe bedarf.

Parallele Verfahren für Matrixinversion und Matrixmultiplikation werden in den nachfolgenden Abschnitten 4.3.2 und 4.3.3 erläutert.

Da das Verfahren neben den zu parallelisierenden Teilen auch Teile enthält, für die oben keine Parallelisierung vorgesehen wurde, ist zu beachten, dass die parallel auszuführenden Operationen nicht von den unmittelbar zuvor seriell ausgeführten Verfahrensteilen abhängen:

- Die Ausdrücke RA (mit Intervallmatrizen für die verifizierte Iteration und reell zur Bestimmung einer Näherungsinversen von RA) hängen nicht vom Ergebnis der Residuumsiteration bzw. vom Ergebnis der Fixpunktiteration als jeweils vorangehende Verfahrensteile ab.

- Die Näherungsinversion von RA (für die zweite Anwendung von `(*)` in Verfahren 4.3.1) hängt (bis auf die Feststellung, ob das Verfahren fortgesetzt wird oder vorzeitig erfolgreich beendet werden kann) nicht vom Ergebnis der vorausgehenden Fixpunktiteration ab.

Es ist also möglich, diese Verfahrensteile bzw. Operationen bereits vor Beendigung der jeweiligen Iteration zu beginnen. Damit kann die zur Verfügung stehende Menge von Prozessen aufgeteilt werden:

- Ein Prozess p_0 (vorzugsweise bei Verwendung eines Master-Worker-Modells der Master-Prozess) führt die seriellen Iterationen aus.

Verfahren 4.3.2 : Paralleler Intervall-LGS-Löser [a]

Seien $[A] \in \mathbb{IR}^{n \times n}$, $[b] \in \mathbb{IR}^n$ gegeben, $A := mid([A])$, $b := mid([b])$.
Seien $p_0, ..., p_q$ die beteiligten Prozesse.

Berechne eine Näherungsinverse R von A.

p_0:	$p_1, ..., p_q$: **(*)**
Setze $\tilde{x} := R \cdot b$. Führe die Residuumsiteration $d := b - A\tilde{x}$ $\tilde{x} := \tilde{x} + R \cdot d$ aus bis $\tilde{x}$ genau genug ist. Berechne $[z]$ aus (3.40).	Berechne $[C]$ aus (3.40).
Setze $[x]^{(0)} := [z]$. Führe die verifizierte Iteration $[x]^{(i)} := [x]^{(i)} \bowtie \epsilon$ $[x]^{(i+1)} := [z] + [C][x]^{(i)}$, $i{:=}0,1,\ldots$ aus bis $[x] := [x]^{(i+1)} \overset{\circ}{\subset} [x]^{(i)}$.	Berechne RA.

Falls erfolgreich:
  Setze $[x] := \tilde{x} + [x]$.
Sonst:
  Berechne eine Näherungsinverse S von RA.
  Berechne $R := SR$.

  Wiederhole **(*)**.
  Falls erfolgreich:
    Setze $[x] := \tilde{x} + [x]$.
  Sonst: Fehlgeschlagen.

---

[a] Aus Platzgründen wurden Hinweise auf alternative Abbruchkriterien ausgelassen.

Verfahren 4.3.3 : Matrizenmultiplikation

Sei anfänglich $c_{ij} = 0$ für $i, j := 1...n$.

Für $i := 1...n$:
  Für $j := 1...n$:
    Für $k := 1...n$:
      $c_{ij} := c_{ij} + a_{ik}b_{kj}$

- Die restlichen Prozesse $p_1, ..., p_q$ (bei $q \in \mathbb{N}$ Prozessen) berechnen die in den nachfolgenden Teilen benötigten Ausdrücke im Voraus.

Damit erhalten wir das in Verfahren 4.3.2 dargestellte parallele Verfahren, wobei die dunkel hinterlegten Bereiche parallel entweder von allen Prozessen oder den in der zugehörigen Spalte genannten Prozessen ausgeführt werden. Die hell hinterlegten Bereiche werden gleichzeitig seriell von einem Prozess ausgeführt.

4.3.2 Parallele Matrixmultiplikation

Die Berechnung des Produkts $C := (c_{ij})$ zweier Matrizen $A := (a_{ij})$, $B := (b_{ij})$ für $i, j = 1...n$ gehört zu den Grundaufgaben des parallelen Rechnens. Im Algorithmus in Verfahren 4.3.3 lassen sich die Schleifen untereinander vertauschen, ohne das Ergebnis zu verändern. Während diese Anordnung der Schleifen besonders für Vektorrechner interessant ist (vgl. z.B. [34]), ist bei der Berechnung auf Parallelrechnern die Frage nach der vorteilhaften Verteilung der Daten von besonderer Bedeutung.

Auf jedem Prozess können entweder bestimmte Zwischenergebnisse oder das Gesamtergebnis für einen bestimmten Teil der Ergebnismatrix bestimmt werden. Aus Implementierungsgründen, die in Kapitel 5 diskutiert werden, ist es im Rahmen dieser Arbeit von besonderem Interesse, dass während der Berechnung eines Elements der Ergebnismatrix keine zwischengelagerte Kommunikation auftritt, d.h. dass jeder beteiligte Prozess

$$Z_p\left\{\begin{pmatrix} A \\ \hline \\ \hline \end{pmatrix}\right. \begin{pmatrix} B & \overbrace{\quad}^{S_p} & \end{pmatrix} = \begin{pmatrix} C & & \end{pmatrix}$$

Abbildung 4.2: Blockweise Matrixmultiplikation

bereits endgültige Ergebnisse für die zu berechnenden Matrixelemente bestimmt. Dies wird erreicht, wenn jeder Prozess p jeweils auf eine Menge vollständiger Zeilen Z_p und eine Menge vollständiger Spalten S_p zugreifen kann (siehe Abbildung 4.2), so dass er einen Block der Ergebnismatrix allein berechnen kann[2].

Jeder Prozess wendet dann Verfahren 4.3.3 mit angepassten Indexgrenzen für den von ihm zu berechnenden Block der Ergebnismatrix an. In der Praxis der eingesetzten Verfahren liegen Teile der Daten bereits vor, Teile der Daten werden neu verteilt, um die beschriebene blockweise Verteilung zu erhalten.

4.3.3 Paralleles Verfahren zur Matrixinversion

Auch die numerische Lösung linearer Gleichungssysteme und damit einhergehend die numerische Inversion von Matrizen gehört zu den verbreiteten Problemen des parallelen Rechnens. Ein Ansatz zur Berechnung wurde in Abschnitt 3.2.2 bereits angegeben.

Ausgehend von Verfahren 3.2.2 ist in Verfahren 4.3.4 eine algorithmische Beschreibung der parallelen LU-Zerlegung angegeben.

Bemerkung 4.3.1 *Für die Kommunikation werden in Verfahren 4.3.4 analog zu den Bezeichnungen in* MPI *(siehe auch Abschnitt 5.1.3) die Bezeichnungen Send, Receive* und *Broadcast benutzt.*

[2] Eine formalere Definition mit Hilfe des Begriffs der *Partition* findet sich z.B. in [34]

Bemerkung 4.3.2 *Bei der Matrixinversion werden Strategien zur Bestimmung von* Pivotelementen *angewandt. Diese sollen hier nicht näher beschrieben werden. Bei spaltenzyklischer Verteilung einer Matrix empfiehlt sich jedoch, ebenfalls eine* Spalten-*Pivotsuche durchzuführen, da nur diese ohne Kommunikationsaufwand realisierbar ist. Zur Berücksichtigung der Umordnung durch die Pivotelemente werden in Verfahren 4.3.4 die Elemente der Matrix L explizit als solche bezeichnet; trotzdem kann die spätere Speicherung in der Matrix A erfolgen. Bemerkungen zu Pivotstrategien finden sich z.B. in [34].*

Bemerkung 4.3.3 *Das Send-Ahead-Prinzip* ermöglicht *es, Leerlaufzeiten bei den Prozessen zu vermeiden, indem die neu berechneten Elemente einer Pivotspalte sobald wie möglich an die anderen Prozesse verschickt werden, d.h. insbesondere, bevor die restlichen in einem Schritt der äußeren Schleife zu berechnenden Elemente bestimmt werden [34].*

Nach der Ausführung der LU-Zerlegung gemäß Verfahren 4.3.4 müssen die in Abschnitt 3.2.2 beschriebenen gestaffelten Systeme gelöst werden. Dies kann einerseits passieren, indem jeder Prozess die Systeme für bestimmte rechte Seiten vollständig löst (die Ergebnismatrix steht ohne relevanten Mehraufwand bei der Kommunikation auf jedem Prozess zur Verfügung). Andererseits können die Systeme partiell mit der beibehaltenen zyklischen Verteilung der Spalten erfolgen. Letzteres jedoch erzeugt zusätzlichen Kommunikationsaufwand, da im ungünstigsten Fall jeder Prozess an der Berechnung der anfallenden Summen für *ein* System beteiligt wird. Die später vorgestellte Softwarekomponente löst die gestaffelten Systeme parallel, indem die rechten Seiten auf die Prozesse verteilt werden und jeder Prozess die ihm zugewiesenen Systeme löst.

Für die gestaffelten Systeme soll an dieser Stelle kein Algorithmus angegeben werden, da sich die zu berechnenden Werte nacheinander jeweils direkt durch Einsetzen ergeben.

4.4 Obermaiers paralleles Verfahren

Das Verfahren von Obermaier, das in Abschnitt 3.3 erläutert wird, wird in [106] auch als paralleles Verfahren angegeben. Dabei werden für je-

Verfahren 4.3.4 : LU-Zerlegung parallel

Sei A spaltenzyklisch auf die Prozesse verteilt.
MyC sei die Menge der zum aktuellen Prozess p_{akt} gehörigen Spalten.
Die Pivotisierung sei in jedem Schritt implizit durch eine Permutation
$p := (p_i), i = 1...n$ gegeben.

Falls $1 \in MyC$:
  Bestimme Pivotindex s.
  Für $m := 1...n$:
    $l_{p_m 1} = a_{p_m 1}/a_{s1}$
  Broadcast s, $l_{p_1 1}, ..., l_{p_n 1}$.
Für $k := 1...n-1$:
  Falls $k \notin MyC$:
    Receive s, $l_{p_1 k}, ..., l_{p_n k}$
  Vertausche p_k und p_s
  Für $j := k+1...n$:
    Falls $j \in MyC$:
      Für $i := k+1...n$:
        $a_{p_i j} := a_{P_i j} - l_{p_i k} a_{p_k j}$
      Falls $j = k+1$ und $j \neq n$:
        Bestimme Pivotindex s.
        Für $m := k+1...n$:
          $l_{p_m k+1} = a_{p_m k+1}/a_{p_s k+1}$
        Broadcast s, $l_{p_1 k+1}, ..., l_{p_n k+1}$

den Verfahrensschritt (zur Einteilung siehe ebenfalls 3.3) jeweils separat Parallelisierungen angegeben. Überdies ist auch die konkrete Implementierung der Software so gestaltet, dass von einer separaten Ausführung der einzelnen Schritte ausgegangen wird (siehe hierzu die Darstellung der Implementierung in Kapitel 5).

Im Folgenden soll daher für die genannten Verfahrensschritte deren Parallelisierung beschrieben werden.

Parallele Berechnung der Näherungslösung

Der wesentliche Bestandteil bei der Bestimmung der Näherungslösung der Integralgleichung in Bezug auf den Aufwand des Verfahrens ist die näherungsweise Lösung linearer Gleichungssysteme. Diese wird parallel formuliert unter Benutzung der *LU-Zerlegung* (siehe auch Abschnitt 3.2.2). Dabei wird ein ähnlicher wie der in Abschnitt 3.2.2 beschriebene Algorithmus verwendet, bei dem statt der *Spaltenpivotsuche* die analoge Strategie der *Zeilenpivotsuche* verwendet wird. Das beschleunigende *Send-Ahead-Prinzip*, auf das in Bemerkung 4.3.3 hingewiesen wird, wird bei Obermaier jedoch nicht angewandt.

Zusätzlich werden in diesem Schritt einige Matrix-Vektor-Produkte parallel berechnet.

Abschätzung des Approximationsfehlers

Dieser Verfahrensschritt wird charakterisiert durch die Darstellung des Integrals mit Hilfe von Quadraturformeln. Dabei wird das Definitionsintervall in eine genügend große Anzahl von Teilintervallen zerlegt und die Kernfunktion der Integralgleichung bzw. deren Werte für die diskretisierte Problemstellung mittels Steigungsarithmetik ausgewertet. Durch die Zerlegung entstehen voneinander unabhängige Teilprobleme, die jeweils vollständig von einem Prozess bearbeitet werden können, während andere Prozesse andere Teilaufgaben bearbeiten.

Zur Verteilung der Aufgaben werden zwei unterschiedliche Verfahren eingesetzt:

- „Klassisches" Master-Slave-Verfahren: Hier verteilt der Master die Teilprobleme auf die Knoten, wobei jeder Knoten zunächst nur eine

einzelne Teilaufgabe bearbeitet und nachfolgend durch den Master eine neue Aufgabe zugewisesen bekommt.

- Lokales Master-Slave-Verfahren bezüglich einer mittels Prioritäten vorgegebenen Ordnung der Prozesse: Jeder Prozess fungiert als Master für diejenigen Prozesse, die niedrigere Priorität besitzen. Dabei wird jeweils die bestehende Menge an Aufgaben auf die verwalteten Prozesse aufgeteilt. Prozesse niedrigerer Priorität können bei Prozessen höherer Priorität nach Bearbeitung ihrer Aufgaben anfragen, um neue Arbeit zu erhalten oder bis zum Erhalt weiterer Arbeit zu warten.

Berechnung der Konstanten K, der Funktion G und der Konstanten α

Auch bei der Berechnung der Konstanten K werden die oben beschriebenen Verfahren zur Bearbeitung unabhängiger Teilaufgaben eingesetzt.

Zusätzlich wird eine verifizierte Inversen einer Matrix berechnet. Hierzu wird das Verfahren von Rump in seiner Grundformulierung verwendet, d.h. nur der erste Teil des Verfahrens kommt zum Einsatz. In dem implementierten Verfahrensteil werden wie in dem in Abschnitt 4.3.2 angegebenen Verfahren die Näherungsinverse bestimmt und die einzige vorkommende Matrixmultiplikation parallel durchgeführt. Die Matrixmultiplikation erfolgt blockspaltenorientiert, d.h. nur die Spalten des zweiten Multiplikanden werden auf die Prozesse verteilt, während der erste Multiplikand auf allen Prozessen vollständig vorliegt, so dass jeweils vollständige Spalten der Ergebnismatrix berechnet werden.

Die folgenden Verfahrensteile und Strategien werden nicht einbezogen[3]:

- *Vorausberechnung* des Ausdrucks $I - RA$ während der Durchführung der Residuumsiteration.
- Erneute Bestimmung der numerisch stabileren Näherungsinversen S zu RA (letzteres Produkt ebenfalls vorausberechnet).
- Erneute Durchführung des Verfahrens mit erhöhter Genauigkeit unter Verwendung der Summe $R = R_1 + R_2$.

[3] Zu den Bezeichnungen der Matrizen siehe Abschnitt 3.2.1

Hinweise zur konkreten Implementierung, zu Unterschieden bei der Verwendung verschiedener Intervallbibliotheken sowie im Hinblick auf Fragestellungen bei der Datenkommunikation mit *MPI* finden sich in Kapitel 5.

Die Bestimmung der Funktion G (siehe Abschnitt 3.3.2) erfolgt nach dem gleichen Verteilungsprinzip für unabhängig zu berechnende Teilaufgaben wie für die Abschätzung des Approximationsfehlers beschieben (s.o.).

Die Bestimmung der Konstanten α wird nicht gesondert parallelisiert; der Aufwand hierfür ist vergleichsweise klein.

5 Softwarekomponenten - Erweiterungen und Neuentwicklungen

In Kapitel 3 wurden Verfahren zur verifizierten Lösung Fredholmscher Integralgleichungen zweiter Art und ihrer Teilprobleme angegeben, darunter insbesondere die folgenden Verfahren:

- Verifizierte Lösung linearer Fredholmscher Integralgleichungen zweiter Art (Verfahren von Klein)
- Verifizierte Lösung linearer Fredholmscher Integralgleichungssysteme zweiter Art (Systemverfahren von Klein)
- Verifizierte Lösung nichtlinearer Fredholmscher Integralgleichungen vom Urysohntyp (Verfahren von Obermaier)
- Verifizierte Lösung linearer Intervall-Gleichungssysteme (Erweitertes Verfahren von Rump)

In Kapitel 4 wurden parallele Verfahren angegeben, die die effiziente parallele Implementierung der oben genannten Verfahren ermöglichen.

In diesem Kapitel wird nun die Software vorgestellt, die diese Verfahren umsetzt. Dabei werden jeweils zunächst existierende Implementierungen genannt und dann die neu erstellten bzw. modifizierten oder erweiterten Implementierungen beschrieben.

Zusätzlich zu den Anwendungsprogrammen werden Module und Werkzeuge vorgestellt, die in der Anwendungssoftware zum Einsatz kommen, aber grundsätzlich eigenständige Komponenten bilden:

- Ein- und zweidimensionale Taylorarithmetik in C-XSC

- MPI-Kommunikation für C-XSC-Datentypen

Diese Komponenten erweitern insbesondere das Einsatzspektrum von C-XSC-Software, und anhand einer modifizierten Anwendung aus dem Gebiet der globalen Optimierung wird für das Beispiel des MPI-Kommunikationspakets für C-XSC-Datentypen gezeigt, dass die entwickelten Komponenten auch in weiterer existierender Software einsatzfähig sind.

Als Beispiel für die Weiterverwendung und Darstellung in anderen Umgebungen wird die Visualisierung von Lösungsfunktionen mit Hilfe eines integrierten Export-Interfaces und eines Pakets zur Intervallarithmetik in Maple vorgestellt.

Zunächst sollen jedoch die grundlegenden Werkzeuge bei der Implementierung verifizierender Verfahren und paralleler Verfahren dargestellt werden, die im Rahmen der vorgestellten Implementierungen relevant sind. Folgende Werkzeuge werden beschrieben:

- Die C++-Klassenbibliothek *C-XSC*, die zahlreiche Intervall-Datentypen für verifizierte Berechnungen zur Verfügung stellt und in den vorgestellten Implementierungen zum Einsatz kommt. Da sie im Rahmen dieser Arbeit um ein Paket zur Datenkommunikation mit *MPI* erweitert wird, erfolgt die Beschreibung ausführlicher als bei der nachfolgend genannten Bibliothek.

- Die C++-Klassenbibliothek *FI_LIB++*, die ebenfalls einen Intervall-Datentyp zur Verfügung stellt und in der existierenden und der modifizierten Implementierung des Verfahrens von Obermaier Verwendung findet.

- Der *Message Passing Interface (MPI) Standard* zur parallelen Kommunikation (in den vorliegenden Programmen durch Implementierungen dieses Standards, insbesondere *MPICH* [49, 50, 99], verwendet). Wie im ersten Aufzählungspunkt angegeben, wird der vorliegende MPI-Funktionsumfang um Routinen für C-XSC-Datentypen erweitert. Diese Routinen kommen in den vorgestellten Neuimplementierungen zum Einsatz.

5.1 Werkzeuge und Bibliotheken

In diesem Abschnitt werden grundlegende Werkzeuge näher beschrieben, die allgemein zur Verwirklichung paralleler oder verifizierter mathematischer Anwendungen zum Einsatz kommen und die speziell in den Implementierungen im Rahmen dieser Arbeit Verwendung finden. In späteren Abschnitten werden dann Erweiterungen dieser Werkzeuge zur Verbindung von parallelem und verifiziertem Rechnen betrachtet. Zusätzliche Software-Werkzeuge wurden im Rahmen dieser Arbeit zur *Erstellung* oder *Dokumentation* der Software verwendet, sollen hier aber nicht näher untersucht werden.

Zunächst werden zwei Bibliotheken beschrieben, die die Verwendung von Intervallen in Verfahrensimplementierungen ermöglichen. Es handelt sich jeweils um Implementierungen in C/C++ [123], das als Basis für die hier betrachteten Implementierungen dient. Zum verifizierten Rechnen mit Intervallen werden in den vorliegenden Implementierungen die Bibliotheken *C-XSC* und `filib++` verwendet. Bestrebungen, Intervalle in die C++-Standardbibliothek zu integrieren [15], können möglicherweise zukünftige Implementierungen von Algorithmen weiter vereinfachen.

5.1.1 C-XSC

C-XSC ist eine C++-Klassenbibliothek für das verifizierte Rechnen. Sie ermöglicht die Entwicklung bzw. Implementierung numerischer Algorithmen mit automatischer Ergebnisverifikation in C++ gemäß dem C++ Standard [64]. Zu diesem Zweck stellt C-XSC zahlreiche Datentypen zur Verfügung, für die jeweils eigenständige Implementierungen von arithmetischen Operatoren und Funktionen existieren. Zusätzlich enthält C-XSC Implementierungen einer Reihe von numerischen Algorithmen mit Ergebnisverifikation, die als *C-XSC Toolbox* [53] in C-XSC integriert sind.

In diesem Abschnitt wird eine kurze Einführung in C-XSC gegeben, wobei besonders diejenigen Eigenschaften betrachtet werden, die im Hinblick auf die Implementierungen der Verfahren in dieser Arbeit relevant sind.

Eine vollständige Beschreibung von C-XSC findet sich in [71]. C-XSC ist als freie Software verfügbar [22] und wird, insbesondere im Hinblick auf neue Compiler und Plattformen, ständig weiterentwickelt [60, 61]. Darüber hinaus nimmt C-XSC die Position einer von mehreren *XSC-Sprachen* [132] ein, zu denen z.B. auch *Pascal-XSC* [69, 70] gehört, dessen Vorgänger *Pascal-SC* [83, 101] bei der Ursprungsimplementierung der Verfahren von Klein (siehe Abschnitt 5.5) verwendet wurde.

Besondere Relevanz für die Verwendung in den nachfolgend vorgestellten Implementierungen haben die Datentypen, die von C-XSC zur Verfügung gestellt werden. Als Werkzeug zur Implementierung von verifizierten Verfahren in Intervall-Arithmetik (siehe Abschnitt 2.1) sind dies vor allem die *Intervall-Datentypen*. Folgende Datentypen werden von C-XSC zur Verfügung gestellt:

- **Skalare Datentypen (Grunddatentypen)**: Zur Darstellung reeller und komplexer Zahlen existieren die Datentypen `real` und `complex`. Intervalle derselben werden durch die Datentypen `interval` und `cinterval` zur Verfügung gestellt.

- **Vektor- und Matrix-Datentypen:** Zu jedem der Grunddatentypen existieren jeweils Vektor- und Matrix-Datentyp für Vektoren und Matrizen aus Elementen des jeweiligen Grunddatentyps: `rvector`, `ivector`, `cvector`, `civector` sowie `rmatrix`, `imatrix`, `cmatrix` und `cimatrix`.

- **Multiple-Precision- bzw. *Staggered*-Datentypen:** Mit Hilfe dieser Datentypen wird eine *Langzahl-Arithmetik* zur Verfügung gestellt, bei der die dargestellten skalaren Werte bis zur n-fachen Genauigkeit der zu Grunde liegenden skalaren Typen haben (mit einem `int`-Wert n)[1]. Dabei wird ein Wert des Langzahltyps als exakte Summe

$$\sum_{i=1}^{n} x_i$$

  von n als Vektor gespeicherten Werten $x_i, i = 1...n$, des Grunddatentyps dargestellt. Das Prinzip der *Staggered*-Arithmetik wird

[1] In der auf IEEE754 [5] basierenden Arithmetik in C-XSC können dabei effektiv Werte $n \leq 39$ verwendet werden (vgl. [71]).

in [71] genauer dargestellt. Für jeden der vorstehend genannten reellen Grund-, Vektor- und Matrix-Datentypen existiert ein korrespondierender Langzahl-Datentyp: `l_real`, `l_interval`, `l_rvector`, `l_ivector`, `l_rmatrix`, `l_imatrix`. Zusätzlich existiert der Datentyp `l_complex` für komplexe Langzahlen, jedoch keine weiteren Langzahl-Datentypen, die auf komplexen Datentypen basieren.

- ***Dotprecision*-Datentypen:** Ein Objekt eines Dotprecision-Datentyps ist in der Lage, das exakte Ergebnis eines Skalarprodukts, also einer Summe $\sum_{i=1}^{n} a_i b_i$ von Produkten beliebiger Werte $a_i, b_i, i = 1...n$, eines Grunddatentyps für eine Anzahl n von Summanden (mit einem `int`-Wert n), exakt darzustellen. Eine kurze Betrachtung des für eine Variable eines solchen Typs erforderlichen Speicherplatzes erfolgt im Anschluss an diese Übersicht. Für jeden Grunddatentyp steht ein zugehöriger Dotprecision-Datentyp zur Verfügung: `dotprecision`, `idotprecision`, `cdotprecision` und `cidotprecision`.

Speicherplatzbestimmung bei Dotprecision-Datentypen

An dieser Stelle wird kurz der erforderliche Speicherplatz für Objekte der Dotprecision-Datentypen am Beispiel des Datentyps `dotprecision` betrachtet.

Ein Gleitkommasystem gemäß Definition 2.2.1 ist durch folgende Werte bestimmt:

- Basis b und Mantissenlänge l
- Minimaler und maximaler Exponent e_{min} und e_{max}

Um Produkte beliebiger Zahlen dieses Formats darstellen zu können, benötigt man damit unter Verwendung einer Anzahl $g = 31$ von Pufferziffern für Überträge

$$L = g + 2e_{max} + 2|e_{min}| + 2l$$

Ziffern zur Basis b [71].

Der zum C-XSC-Datentyp `real` äquivalente Datentyp `double` gemäß IEEE754 Floating Point Arithmetic [5] weist dabei die folgenden konkreten Werte auf:

$$b = 2, \;\; l = 53, \;\; e_{min} = -1022, \;\; e_{max} = 1023.$$

Damit erhält man

$$L = 31 + 2 \cdot 1024 + 2 \cdot 1022 + 2 \cdot 53 = 4227$$

Ziffern zur Basis 2, d.h. 529 Byte.

Im Vergleich zu einem Wert von Typ `double`, der eine Länge von 8 Bytes besitzt, belegt eine Variable des Typs `dotprecision` also den 66,125-fachen Speicherplatz. Für die Kommunikation in parallelen Programmen bedeutet dies, dass bei Verwendung einer `dotprecision`-Variable statt einer `double`- oder `real`-Variable die 66,125-fache Datenmenge verschickt werden muss, was den Kommunikationsaufwand des betreffenden Programms erheblich steigert. Falls möglich, sollte die Kommunikation mit `dotprecision`-Objekten also vermieden werden.

Gleichzeitig ist es ratsam, möglichst nur dort `dotprecision`-Objekte einzusetzen, wo exakte Ergebnisse oder Zwischenergebnisse tatsächlich benötigt werden, um nicht unnötig Speicherplatz zu belegen. `dotprecision`-Datentypen sollten also auch dann, wenn lediglich solche Operationen auszuführen sind, die für `dotprecision`-Datentypen definiert sind[2], nicht generell als Standard-Datentypen für die Berechnungen in einem Programm benutzt werden.

Speichertechnische Realisierung

Im Hinblick auf das in Abschnitt 5.3 vorgestellte MPI-Kommunikationspaket für C-XSC-Datentypen ist die speichertechnische Realisierung der C-XSC-Datentypen von Belang. Daher soll diese hier beschrieben werden. Dabei soll nicht auf die Realisierung des zu Grunde liegenden Datentyps

[2]Für zwei Operanden eines `dotprecision`-Datentyps sind nur die arithmetischen Operationen $+$ und $-$, nicht jedoch die Operationen $\cdot$ und $/$ definiert, da gemäß der Definition der `dotprecision`-Datentypen keine exakte Berechnung der Ergebnisse dieser Operationen garantiert ist (vgl. [71]).

real, aber auf die auf diesem Datentyp aufbauenden weiteren C-XSC-Datentypen eingegangen werden.

Grunddatentypen Die Datentypen interval und complex enthalten jeweils zwei real-Werte zur Darstellung der Grenzen bzw. zur Darstellung von Real- und Imaginärteil, der Datentyp cinterval enthält zwei Objekte des Typs interval.

Vektor- und Matrixdatentypen Vektordatentypen bestehen aus zwei int-Werten für die Indexgrenzen und einem dynamisch allokierten C-Feld von Objekten des Grunddatentyps. Zur zusätzlichen Speicherung der Feldlänge existiert in älteren C-XSC-Versionen ein int-Wert size. Entsprechend sind in einem Objekt eines Matrix-Datentyps für jede Dimension die Grenzen des jeweiligen Indexbereichs (sowie die Feldlängen xsize, ysize bei zweidimensionaler Interpretation als Matrix, s.o.) sowie die Werte als dynamisch allokiertes C-Feld (jedoch wie bei Vektoren als *eindimensionales* C-Feld der Länge xsize·ysize) gespeichert.

Multiple Precision-Datentypen Die zwei Multiple-Precision-Datentypen l_real und l_interval bestehen aus einem dynamisch allokierten C-Feld von Werten des Typs real, dessen Länge sich aus der als int-Wert gespeicherten *Staggered Precision* herleitet. Objekte vom Typ l_complex bestehen aus zwei Werten des Typs l_real. Vektoren und Matrizen der Langzahldatentypen entsprechen ihren nicht langzahligen Pendants unter Ersetzung des Grunddatentyps durch den zugehörigen Langzahl-Grunddatentyp.

Dotprecision-Datentypen Weiter oben wurde bereits der zur Abspeicherung eines Objekts der Dotprecision-Datentypen benötigte Speicherplatz beschrieben. C-XSC enthält hierzu eine globale Variable BUFFERSIZE, die diese Größe als Wert in Bytes enthält. Intern wird der entsprechende Speicherplatz als dynamisch allokiertes C-Feld von long int-Werten angelegt; Letzteres ist nur unter Rückgriff auf C-XSC-interne Zeigertypen ansprechbar.

Im Hinblick auf die später vorgestellten MPI-Kommunikationsroutinen wird darauf hingewiesen, dass die Vektor- und Matrixdatentypen in C-XSC nicht als allgemeine Template-Klassen unter Verwendung des Grunddatentyps als Template-Parameter implementiert sind und somit

auch Routinen, die diese Datentypen verwenden, nicht in diesem Sinne als Template-Routinen angelegt werden können.

Zu den C-XSC-Datentypen steht jeweils eine umfangreiche Anzahl von Operatoren zur Verfügung, die jeweils für alle sinnvollen Kombinationen von Parameter-Datentypen überladen wurden. Ebenso steht eine Vielzahl von Funktionen zur Verfügung, die insbesondere die Standardfunktionen und weitere Funktionen abdecken. Dabei wurden vielfach eigene Algorithmen verwendet, die im Sinne der Intervallarithmetik optimiert sind. In der Anzahl der implementierten mathematischen Funktionen ist auch ein grundsätzliches, von der Struktur der Datentypen selbst unabhängiges Unterscheidungskriterium zwischen verschiedenen Intervall-Bibliotheken zu sehen.

5.1.2 FILIB++

Wie C-XSC (siehe Abschnitt 5.1.1) ist auch `filib++` eine Bibliothek zum verifizierten wissenschaftlichen Rechnen. Sie stellt eine C++-Erweiterung der C-Bibliothek `fi_lib` [59] dar und enthält die nachfolgend beschriebenen Bestandteile.

Zum einen wird ein Intervall-Datentyp zur Verfügung gestellt. Während `fi_lib` als C-Bibliothek, die nicht auf das Klassenkonzept von C++ zurückgreifen konnte, lediglich eine Intervall-Struktur definierte, ist der Intervall-Datentyp in `filib++` als C++-Template-Klasse realisiert. Über Template-Parameter lassen sich der zugrundeliegende Zahl-Typ (z.B. `float` oder `double`) und die Art der zu verwendenden Rundung spezifizieren. Intervall-Operationen werden als Operatoren zur Verfügung gestellt, wobei die eigentliche Operation unter Verwendung von *Traits-Klassen* definiert wird[3].

Zum anderen ist eine große Zahl von Funktionen implementiert, die mit Hilfe des Intervall-Datentyps die genaue Einschließung des Wertebereichs von Funktionen ermöglichen. Einige in C-XSC implementierte Funktionen sind hier nicht enthalten.

[3]Das Konzept der *Traits-Klasse* wird z.B. in [100] beschrieben. Auch in der C++-Standardbibliothek werden Traits-Klassen verwendet; sie finden auch bei Stroustrup [123] Erwähnung.

Im Gegensatz zu C-XSC werden in der Bibliothek `filib++` keine separaten Vektor- und Matrixklassen implementiert, ebenso sind keine Langzahl- bzw. Dotprecision-Datentypen vorhanden.

Zusätzlich beinhaltet `filib++` das Rechnen mit *Containment Sets* [86], das durch Einbeziehung der Werte $-\infty$ und ∞ als mögliche Intervallgrenzen auch Ausnahmefälle (wie etwa die Division durch 0) behandeln kann[4].

Frühere Versionen von `filib++` enthielten neben der oben beschriebenen Implementierung mit Hilfe von Templates eine alternative Implementierung als Makro-Version [85], die bei älteren Compilern Ineffizienzen des mit der Template-Version generierten Codes vermeiden sollte. Diese Version enhielt ausschließlich einen Intervall-Datentyp bezüglich des Basis-Zahltyps `double` und wurde nicht für weitere Zahltypen ausgeprägt. Die existierende Implementierung des Verfahrens von Obermaier verwendet diese Implementierung. Die Makro-Version ist jedoch mittlerweile nicht mehr verfügbar. Wegen der unterschiedlichen Syntax der Datentypen sind die Versionen nicht vollständig kompatibel, so dass sich einige Änderungen ergeben, wenn die Bibliotheksversion in Anwedungsprogrammen ausgetauscht werden soll:

- Änderung des Intervall-Typs `Interval` in `interval<>` (mit Wahl von Default-Template-Parametern) mit entsprechender Anpassung von Variablen, Funktionsinterfaces und Operationen.
- Einbindung anderer Headerdateien in den Implementierungsdateien der betrachteten Anwendung.
- Übersetzung der Anwendungsprogramme mit anderen Bibliotheken, so dass auch der Prozess der Generierung ausführbarer Programme mit Hilfe von Makefiles modifiziert werden muss.

Gleichzeitig ergeben sich Änderungen, die auf einer erhöhten Schnittstellensicherheit in der Template-Version beruhen, da in der Template-Version an verschiedenen Stellen Parameter und Referenzen als konstant deklariert werden, während dies in der Makro-Version an den betreffenden Stellen nicht der Fall ist.

[4] Dabei folgen die Autoren den Definitionen aus [21].

Als Beispiel sei die folgende Deklaration aus dem `filib++`-Quellcode gegeben:

In der Template-Version ist die Standardfunktion `exp` deklariert als

```
template
<typename N, rounding_strategy K, interval_mode E>
interval<N,K,E>  exp(interval<N,K,E> const & x);
```

In der Makro-Version ist dieselbe dagegen deklariert als

```
Interval exp(Interval x);
```

In der Template-Version ist der Funktionsparameter also als konstante Referenz vereinbart, in der Makro-Version dagegen nicht.

Im folgenden Code wird eine Funktion `myfunc` vereinbart, die als Übergabeparameter eine Funktion mit Intervall-Parameter besitzt:

```
bool myfunc(Interval f(Interval));
```

Für die Funktion `exp` in der Definition aus der Makro-Version kann `exp` nun als Parameter bei Funktionsaufrufen der Funktion `myfunc` übergeben werden.

Auch bei ordnungsgemäßem Ersatz des Parameters vom Typ `Interval` durch die Referenz `interval<>&` in

```
bool myfunc(interval<> f(interval<>&));
```

wird der Aufruf der Funktion `myfunc` mit dem Parameter `exp` in der Definition aus der Template-Version nun als Integritätsverletzung aufgefasst, da es sich nicht um eine konstante Referenz handelt bzw. die Signatur der übergebenen Funktion nicht mehr mit Ihrer Deklaration in der Bibliothek übereinstimmt. Daher müssen bei Verwendung der Template-Version entsprechende Modifikationen im Anwendungscode vorgenommen werden, um hier die Datenintegrität zu gewährleisten.

5.1.3 MPI

MPI (*Message Passing Interface*) ist die Spezifikation einer Bibliothek zur parallelen Kommunikation durch message passing (siehe Abschnitt 2.6).

MPI wurde durch das *MPI Forum* [91] standardisiert in [92] und [93]. Der Standard enthält Spezifikationen für

- Kommunikationsfunktionen zum Versenden von Daten zwischen Prozessen in einem parallelen Programm
- Objekte zur Strukturierung paralleler Kommunikation (Gruppen, Kommunikatoren, Prozess-Topologien)
- Konstanten, Funktionen und Objekte zur Unterstützung oder Vereinfachung der Kommunikation mit den obigen Funktionen (z.B. zur Unterstützung der Kommunikation benutzerdefinierter Datentypen)

Zusätzlich werden sprachbezogene Interfaces für Implementierungen in Fortran und C definiert.

Eine Implementierung des MPI-Standards liegt z.B. mit *MPICH* [49, 50, 99] vor.

Im Hinblick auf die in Abschnitt 5.3 vorgestellte Erweiterung für C-XSC-Datentypen werden im folgenden die wichtigsten Kommunikationsfunktionen angegeben. Eine Darstellung der MPI-Funktionen zum Umgang mit benutzerdefinierten Datentypen findet sich in Abschnitt 5.3.1.

Die Kommunikationsfunktionen zerfallen in zwei Gruppen:

- **Point-to-point Communication**: Mit *point-to-point communication* wird das Versenden von Daten von *einem* Prozess zu *einem einzelnen* anderen Prozess bezeichnet. Es gibt also nur einen Absender und einen Empfänger der gesendeten Daten.
- **Collective Communication**: MPI spezifiziert auch Funktionen für Kommunikationsvorgänge, an denen *mehrere* Absender und/oder *mehrere* Empfänger beteiligt sind. Diese Kommunikationsvorgänge bezeichnet man als *collective communication*.

Die wichtigsten Funktionen der *Point-to-point Communication* sind

- `MPI_Send`
- `MPI_Recv`

zum Versenden und Empfangen von Daten. Die Funktion `MPI_Send` existiert darüber hinaus in verschiedenen Versionen, die unterschiedliche Kommunikationsmodi verwenden:

- `MPI_Bsend` verwendet den *buffered mode*, der die Nachricht in einem Puffer zwischenspeichert, bis der Empfängeraufruf erfolgt und dem Absender erlaubt, bereits vor dem vollständigen Empfang der Nachricht weiterzuarbeiten.
- `MPI_Rsend` verwendet den *ready mode*, der sicherstellt, dass die Versende-Operation erst dann gestartet wird, wenn der zugehörige Empfängeraufruf erfolgt.
- `MPI_Ssend` verwendet den *synchronized mode*, der sicherstellt, dass die Versende-Operation erst dann beendet wird, wenn der zugehörige Empfängeraufruf tatsächlich erfolgt ist.

Zusätzlich existieren *nicht blockierende* Versionen `MPI_Isend`, `MPI_Ibsend`, `MPI_Irsend` und `MPI_Issend`. Bei diesen fährt der aufrufende Prozess in jedem Fall mit der Arbeit fort, ohne die Versende-Operation vollständig durchzuführen. Die benötigten Schritte zur Durchführung der Operation können architekturabhängig nebenläufig durchgeführt werden. Es ist jedoch aktiv zu überprüfen, ob bzw. wann die Operation erfolgreich beendet wurde.

Die wichtigsten Funktionen der *collective communication* sind:

- `MPI_Bcast`: Die zu versendenden Daten werden an alle beteiligten Empfänger versendet.
- `MPI_Scatter`: Die zu versendenden Daten werden stückweise an alle beteiligten Empfänger *verteilt*.
- `MPI_Gather`: Die zu empfangenden Daten werden stückweise von allen beteiligten Empfängern *gesammelt*.

Beim stückweisen Versand bzw. Empfang wird dabei der Versand bzw. Empfang aufeinanderfolgender Teile eines zusammenhängenden Speicherbereiches angenommen. Dies führt bei Betrachtung von Daten und Objekten mit unbekannter oder nicht zusammenhängender Speichergeometrie dazu, dass diese Funktionen nur eingeschränkt oder nicht im Sinne des Benutzers verwendet werden können, ohne zusätzliche Operationen von Hand vorzunehmen.

Die oben genannten Funktionen können auf Parameter angewendet werden, deren Typ einer der von MPI zur Verfügung gestellten Datentypen ist. MPI bildet dabei jeweils die meisten der Grunddatentypen derjenigen Programmiersprache ab, die der MPI-Implementierung zu Grunde liegt (Fortran oder C - in den folgenden Abschnitten wird eine C-Implementierung angenommen).

MPI-Datentypen sind etwa

- `MPI_CHAR`
- `MPI_INT`
- `MPI_DOUBLE`

Die genauen Interfaces der Funktionen und weitere Datentypen sind dem MPI-Standard [92] zu entnehmen.

Eine umfangreiche Einführung in die parallele Programmierung mit MPI findet sich in [47, 48].

5.2 Taylorarithmetik in C-XSC

In Abschnitt 2.3 wird das Rechnen mit Funktionen, die als Taylorentwicklung vorliegen, beschrieben. Hier soll nun die Implementierung der Taylorarithmetik in C-XSC vorgestellt werden.

5.2.1 Existierende Implementierung

Eine Implementierung der Taylorarithmetik in C-XSC stellt Bräuer in [11] und [12] vor. Die wesentlichen Elemente der Implementierung sind

- C++-Klassen zur Darstellung ein- oder mehrdimensionaler Funktionen als Taylor-Objekte, in denen die Koeffizienten der Taylor-Entwicklung gespeichert werden.
- Arithmetische Operatoren und Funktionen als überladene Operatoren und Funktionen in C++.
- Hilfsklassen für Vektoren von Taylor-Objekten und Hilfsfunktionen z.B. zur Initialisierung von Objekten (siehe auch Beispiel 5.2.1)

Die Klassen werden im *namespace* `taylor` zusammengefasst, so dass eine Kapselung der verwendeten Klassen und Funktionen gegeben ist und Namenskonflikte vermieden werden.

Der Umfang in Hinblick auf die implementierten Funktionen umfasst alle Standardfunktionen, d.h. die trigonometrischen Funktionen, Exponentialfunktion und Logarithmus, Wurzel und Potenzen etc.

Mittels Headerdateien, die die Interfaces zur Verfügung stellen, können die Klassen in die jeweilige Anwendung eingebunden werden.

Die Implementierung wurde von Blomquist, Hofschuster und Krämer in [10] erweitert und enthält weitere, optimierte Funktionen zur besonders genauen Einschließung von Funktionsausdrücken wie z.B. dem Ausdruck $\sqrt{x^2 - 1}$.

5.2.2 Update und Erweiterung

Im Rahmen dieser Arbeit wurde die oben angegebene Implementierung nochmals erweitert.

Die Implementierung der ein- und zweidimensionalen reellen Taylorarithmetik umfasst nunmehr auch die folgenden Ergänzungen.

Arithmetische Erweiterungen:

- Es wurden die Gaußsche Fehlerfunktion erf und die komplementäre Fehlerfunktion erfc hinzugefügt.

- Die Klassen wurden um Vergleichsoperatoren für Taylor-Objekte ergänzt.

Technische Erweiterungen:

- Es wurden folgende Operatoren hinzugefügt:

  - Ausgabeoperatoren

  - Zusätzliche Zuweisungsoperatoren

  - Indexoperatoren

- Zusätzlich wurden verschiedene kleinere Hilfsfunktionen implementiert.

Die Klassen sind, der zu Grunde liegenden Implementierung folgend, so konzipiert, dass der Entwicklungspunkt der Taylorentwicklung nicht mitgespeichert wird. Dies hat zur Folge, dass die Auswertung der durch das Objekt dargestellten bzw. eingeschlossenen Funktion nur in der jeweiligen Anwendung unter Einbezug des jeweils aktuellen Entwicklungspunktes geschehen kann. Der Einbezug des Entwicklungspunktes in die Implementierung ist jedoch nicht problemlos möglich, da bei verschiedenen zweistelligen Operationen (etwa der Addition) unterschiedliche Entwicklungspunkte der Operanden dazu führen würden, dass das Ergebnis der Operation nicht wohldefiniert ist bzw. nur unter Anwendung zusätzlicher Umformungen der Koeffizienten ein wohldefiniertes Ergebnis erzielt werden kann.

Anwendung

Das folgende Beispiel zeigt die Anwendung der zweidimensionalen Taylorarithmetik.

Beispiel 5.2.1 *In diesem Beispiel werden zweidimensionale Taylorentwicklungen der Ordnung 1 der Funktionen*

$$\begin{aligned}&\mathrm{erf_}spt : (s,t) \mapsto \mathrm{erf}(s+t)\\&\mathrm{erfc_}spt : (s,t) \mapsto \mathrm{erfc}(s+t) \qquad \textit{und}\\&\mathrm{erf_}spt + \mathrm{erfc_}spt\end{aligned}$$

am Entwicklungspunkt $(2,-3)$ *berechnet und ausgegeben. Es handelt sich dabei um die neu implementierten Funktionen nach den Formeln aus Abschnitt 2.3.3. Anhand der dritten Funktion, die gleich der Konstanten* 1 *ist, lassen sich dabei leicht die praktisch berechneten Taylorkoeffizienten überprüfen. Der Programmcode des Beispielprogramms wird im Folgenden wiedergegeben.*

```
#include <iostream>
#include "dim2taylor.hpp"
using namespace std;
using namespace cxsc;
using namespace taylor;

int main()
{
    // Ordnung der Taylorentwicklung
    int ord = 1;

    // Entwicklungspunkt in s- und t-Richtung
    ivector iv(2);
    iv[1] = interval(2);
    iv[2] = interval(-3);

    // Initialisierung der Taylorobjekte
    // fuer die Variablen
    dim2taylor s = init_var(ord,1,iv[1]);
    dim2taylor t = init_var(ord,2,iv[2]);

    //Funktionsaufrufe
    dim2taylor t1,t2,t3;
    t1 = erf(s+t);
    t2 = erfc(s+t);
    t3 = erf(s+t)+erfc(s+t);
```

```
    // Ausgabe der Ergebnisse
    cout << "erf(s+t)=␣" << t1 <<  endl;
    cout << "erfc(s+t)=␣" << t2 <<  endl;
    cout << "erf(s+t)+erfc(s+t)=␣" << t3 <<  endl;
}
```

Beispiel zweidimensionale Taylorentwicklung

In dem Programm werden zunächst die Ordnung der Taylorentwicklung und der Entwicklungspunkt ($\in \mathbb{IR}^2$) festgelegt.

Danach werden Taylorobjekte für die einzelnen Variablen initialisiert. Bei der mehrdimensionalen (hier zweidimensionalen) Taylorarithmetik ist zu beachten, dass eine Zuordnung der Variablen zu den Ableitunsrichtungen vorzunehmen ist („s ist die erste Variable, t die zweite Variable“). Hierzu werden mittels der Funktion `init_var` *zwei Objekte mit entsprechenden Ableitungsrichtungen erzeugt.*

Nach dieser Initialisierung können dann alle implementierten Funktionen und arithmetischen Operatoren verwendet werden, um Taylor-Objekte für mathematische Ausdrücke zu erstellen, die diese Funktionen und Operatoren beinhalten.

Schließlich werden die Ergebnisse mittels der neu implementierten Ausgabeoperatoren ausgegeben.

Die Ergebnisausgabe lautet wie folgt:

```
erf(s+t)= [dim2taylor object, order 1:]
[ -0.842701, -0.842700] [  0.415107,  0.415108]
[  0.415107,  0.415108]

erfc(s+t)= [dim2taylor object, order 1:]
[  1.842700,  1.842701] [ -0.415108, -0.415107]
[ -0.415108, -0.415107]

erf(s+t)+erfc(s+t)= [dim2taylor object, order 1:]
[  0.999999,  1.000001] [-1.776357E-015,1.776357E-015]
[-1.776357E-015,1.776357E-015]
```

Beispielausgabe

Die Taylorkoeffizienten werden, nach Variablenrichtung geordnet ausgegeben, d.h. die Ausgabe des ersten Objekts entspricht der Taylorentwicklung

$$[-0.842701, -0.842700] + [0.415107, 0.415108](s-2) + [0.415107, 0.415108](t+3).$$

Es ist zu erkennen, dass der dritte Ausdruck tatsächlich eine Einschließung der Identität darstellt.

Die eindimensionale Taylorarithmetik wird in gleicher Weise angewendet, ohne dass jedoch die Problematik der anfänglichen Initialisierung mehrerer Variablen zum Tragen kommt.

5.3 MPI-Kommunikationspaket für C-XSC-Datentypen

Um parallele Programme implementieren zu können, in denen Objekte von C-XSC-Datentypen zwischen Prozessen versendet werden, müssen Möglichkeiten bestehen, diese Kommunikation in der Implementierung zu realisieren.

MPI (siehe Abschnitt 5.1.3) verfügt dabei zunächst nicht über die direkte Möglichkeit, benutzerdefinierte Datentypen wie C-XSC-Datentypen als Parameter in den vordefinierten Kommunikationsfunktionen zu verwenden. Daher muss entweder die Kommunikation in jeder Anwendung so formuliert werden, dass die bestehenden Funktionen zur Anwendung kommen können, oder die benötigte Funktionalität muss gesondert bereitgestellt werden.

Die MPI-Kommunikationsfunktionen können lediglich Werte der Grunddatentypen der jeweils zu Grunde liegenden Programmiersprache (hier: C) sowie C-Felder solcher Werte korrekt verwenden. Es besteht die Möglichkeit, Objekte benutzerdefinierter Datentypen in Elemente der Basistypen zu zerlegen und diese einzeln zu versenden. Dies jedoch ist nicht nur umständlich und fehleranfällig, sondern auch ineffizient, da eine Aufteilung des Datenversands auf mehrere Versendevorgänge deutlich mehr Zeit

in Anspruch nehmen kann als der Versand eines Objekts in einem einzelnen Kommunikationsvorgang. Ein Test hierzu findet sich in Kapitel 6.

Im Folgenden sollen zunächst die Mittel vorgestellt werden, die MPI zur Behandlung benutzerdefinierter Datentypen bietet. Danach sollen bestehende Tools angegeben werden, die diese Mittel zur Bereitstellung von Kommunikationsfunktionen nutzen, und deren Einsatzmöglichkeiten geprüft werden. Schließlich soll ein neues Interface zur MPI-Kommunikation mit C-XSC-Datentypen vorgestellt werden.

5.3.1 Verarbeitung benutzerdefinierter Datentypen mit MPI

MPI bietet zwei Vorgehensweisen zur Behandlung benutzerdefinierter Datentypen [92]:

- Definition neuer MPI-Datentypen
- *Packing/Unpacking*

Definition neuer MPI-Datentypen

MPI stellt eine Reihe von Funktionen zur Definition neuer MPI-Datentypen zur Verfügung. Das Vorgehen zur Definition solcher Datentypen beruht auf dem Begriff der *type map*, mit der eine Beschreibung der Struktur der Daten eines Objekts vorgenommen wird.

Eine *type map* T ist definiert als

$$T := ((type_0, disp_0), ..., (type_{n-1}, disp_{n-1}))$$

mit bestehenden MPI-Datentypen $type_i$ und Abständen $disp_i$ für $i = 0...n-1$, die entweder in Bytes oder als Vielfaches der Größe eines Typs gegeben sind für festes $n \in \mathbb{N}$. Der Vektor $(type_i)_{0 \leq i < n}$ heißt dann *Signatur*.

Mit der *type map* liegt eine Definition eines neuen MPI-Datentyps vor, indem die Struktur des neuen Typs durch Typ und Anordnung der zu einem Objekt gehörigen Elemente im Speicher beschrieben wird.

MPI stellt folgende Funktionen zur Verfügung, die einen neuen Datentyp im obigen Sinn definieren:

- `MPI_Type_contiguous` erfasst speichertechnisch zusammenhängende Aneinanderreihungen von Elementen eines bekannten Datentyps.
- `MPI_Type_vector` und `MPI_Type_hvector` erfassen Aneinanderreihungen von speichertechnisch zusammenhängenden, gleich großen Blöcken gleicher Elemente mit gleichmäßigen Abständen zwischen den Blöcken.
- `MPI_Type_indexed` erfasst Aneinanderreihungen von speichertechnisch zusammenhängenden, unterschiedlich großen Blöcken gleicher Elemente mit ungleichmäßigen Abständen zwischen den Blöcken.
- `MPI_Type_struct` schließlich erfasst Aneinanderreihungen von speichertechnisch zusammenhängenden, unterschiedlich großen Blöcken unterschiedlicher Elemente mit ungleichmäßigen Abständen zwischen den Blöcken.

Diese Vorgehensweise erlaubt die Definition neuer Datentypen unterschiedlicher Struktur. Sie ist jedoch in zweierlei Weise eingeschränkt:

- Die Länge des Datentyps und seiner Elemente muss zum Definitionszeitpunkt feststehen. Es können also keine Datentypen variabler Länge definiert werden.
- Die Abstandsangaben der type map beziehen sich stets auf einen festen Bezugspunkt. Dies bedeutet jedoch, dass keine MPI-Datentypen definiert werden können, die dynamischen Speicherplatz enthalten, da sich dynamisch allokierter Speicherplatz nicht in feste Relation zum Speicherort des Objekts selbst setzen lässt.

Dies schränkt auch im Hinblick auf C-XSC-Datentypen die Verwendbarkeit dieser Vorgehensweise ein.

Packing/Unpacking

Die Vorgehensweise des *Packing/Unpacking* ist nicht auf die Definition von Datentypen gerichtet, sondern darauf, für die Kommunikation einen

zusammenhängenden Speicherbereich mit Daten zu erstellen, da MPI nur zusammenhängende Speicherbereiche verschicken kann. Hierfür steht der spezielle MPI-Datentyp `MPI_Packed` zur Verfügung, der Daten als nicht typisierten Speicher betrachtet und nicht im Sinne eines vorgegebenen Datentyps interpretiert.

MPI stellt die beiden Funktionen

- `MPI_Pack` und
- `MPI_Unpack`

zur Verfügung. Mit `MPI_Pack` können beliebige Datenelemente in einen zuvor definierten Speicherbereich kopiert werden, mit `MPI_Unpack` können Datenelemente aus selbigem extrahiert und als Elemente eines Datentyps interpretiert werden. Mittels einer Folge von `MPI_Pack`-Aufrufen können so beliebige Folgen von Daten in zusammenhängende Speicherbereiche gepackt werden. Diese Speicherbereiche können dann verschickt und vom Empfängerprozess wieder ausgepackt werden. Bei dieser Vorgehensweise bietet sich insbesondere die Möglichkeit, dass Anteile eines gepackten Datenpakets von zuvor gepackten Daten des Pakets abhängen. Es ist somit z.B. möglich, zuerst die Länge eines Vektors und dann die Daten des Vektors in ein Datenpaket zu packen und zu verschicken, so dass der Empfängerprozess zunächst die Vektorlänge und dann die Vektordaten aus dem empfangenen Datenpaket extrahiert. Damit wird im Gegensatz zur Definition von MPI-Datentypen der Versand von Objekten variabler Länge ermöglicht.

Mit Hilfe des obigen Prozesses lassen sich überladene Versionen der MPI-Kommunikationsroutinen konstruieren, die die Daten von Objekten eines Datentyps zuerst packen, dann verschicken und schließlich wieder entpacken. Diese überladenen Versionen können als Interface zur Verfügung gestellt werden.

In Abschnitt 5.3.3 wird ein Erweiterungspaket zur MPI-Kommunikation mit C-XSC-Datentypen vorgestellt, das auf diesen beiden Vorgehensweisen basiert.

Zunächst sollen jedoch zwei bestehende Tools untersucht werden, die die Erstellung von MPI-Datentypen aus benutzerdefinierten Datentypen vereinfachen.

5.3.2 Tools

Zur einfacheren Handhabung benutzerdefinierter Datentypen mit MPI existieren zwei Tools, deren Umfang an dieser Stelle kurz geschildert werden soll:

- AutoMap/AutoLink
- C++2MPI

AutoMap/AutoLink

AutoMap und *AutoLink* sind zwei aufeinander aufbauende Tools, die Verarbeitung benutzerdefinierter Datentypen in MPI vereinfachen.

Die Grundlage bildet das Paket AutoMap [23]. Das Paket automatisiert den Prozess der Erstellung von MPI-Datentypen aus *C-Strukturen*. Das Vorgehen zur Erstellung eines neuen MPI-Datentyps mit Hilfe der von MPI zur Verfügung gestellten Definitionsfunktionen wird dem Anwendungsprogrammierer abgenommen. AutoMap ist ein Compiler, der Quellcode-Dateien mit den Definitionen der C-Strukturen liest und eine separate Datei mit den Anweisungen zur Erstellung des entsprechenden MPI-Datentyps generiert. Dabei werden die Definitionsmöglichkeiten von MPI (siehe vorausgehender Abschnitt) zur Generierung des neuen Codes genutzt.

Das Paket AutoLink [24] ist eine auf AutoMap aufbauende Erweiterung, die zusätzlich die Behandlung dynamischer Datenstrukturen wie verketteter Listen erlaubt. Da die Definition von MPI-Datentypen mit dynamischem Speicherplatz nicht möglich ist, wird auch hier auf das Packen von Daten zurückgegriffen. Um einen dynamischen Datentyp zu versenden, wird seine logische Struktur in einer geeigneten Verwaltungsstruktur dargestellt, die in Form von C-Feldern gespeichert wird. Mit Hilfe dieser zusätzlichen Struktur wird es dem Empfänger möglich gemacht, seinerseits die erforderliche Datenstruktur wieder aufzubauen.

Die Pakete bieten umfangreiche Möglichkeiten zur Verwendung benutzerdefinierter Datentypen in MPI, sind aber leider auf C-Strukturen beschränkt. Zudem sieht die Vorgehensweise von AutoMap vor, die Quell-

code-Dateien mit den Datentyp-Definitionen zu lesen, so dass Datentypen mit variabel langen Attributen nicht berücksichtigt werden.

Da die Pakete keine Verwendung bei der Erstellung des MPI-Kommunikationspakets für C-XSC-Datentypen fanden, wird für detaillierte Beschreibungen und technische Details auf [39] und [94] verwiesen.

C++2MPI

Das Paket *C++2MPI* [58] stellt im Gegensatz zu AutoMap/AutoLink die Möglichkeit zur Verfügung, Objekte von C++-Datentypen mit MPI zu verschicken. C++2MPI ist Teil des *Processing Graph Method Tool* [65, 105], aber auch eigenständig benutzbar. Es stellt ein C++-Äquivalent zu AutoMap dar, d.h. ebenso wie dieses Paket handelt es sich um einen Compiler, der vorgegebenen Code, hier in Form von C++-Klassen, liest und als Ausgabedatei den Code zur Definition eines entsprechenden MPI-Datentyps generiert.

Dieses Paket ist auch in sofern ein Pendant zu AutoMap, als dass die Möglichkeiten von AutoLink zur Verarbeitung dynamischer Datentypen nicht zur Verfügung gestellt werden. Damit ist das Paket nicht auf die C-XSC-Datentypen anwendbar. Zukünftige Entwicklungen können die Definition von MPI-Datentypen aus C++-Klassen möglicherweise auch bei Benutzung dynamischen Speicherplatzes ermöglichen.

Mit dem nachfolgend vorgestellten Paket wird unabhängig von Tools wie den oben beschriebenen die MPI-Kommunikation für C-XSC-Datentypen ermöglicht. Da es sich um ein für C-XSC angepasstes Paket handelt, können spezielle Konzepte und Strukturen direkt umgesetzt werden, so z.B. die Definition der *Dotprecision*-Datentypen, die unter Verwendung C-XSC-interner Konstanten und Felder definiert werden und deren komplexe Definitionsstruktur vermutlich in allgemeinen Paketen nicht korrekt abgebildet werden könnte.

5.3.3 Das neue Kommunikationspaket

Es soll nun das neue MPI-Kommunikationspaket für C-XSC-Datentypen vorgestellt werden.

Zunächst soll die Fragestellung betrachtet werden, welche MPI-Kommunikationsfunktionen in einem allgemeinen Interface neu implementiert werden können.

Die Routinen der *Point-to-point communication* können sinnvoll umgesetzt werden, ebenso ist dies für diejenigen Routinen der *Collective communication* möglich, bei denen Objekte vollständig zwischen mehreren Prozessen verschickt werden, wie es beim *Broadcast* der Fall ist.

Die weiteren Funktionen der Collective communication sind jedoch nur mit Einschränkungen zu handhaben. Bei Routinen wie *scatter* und *gather* werden keine vollständigen Objekte betrachtet, sondern Teilstrukturen an verschiedene Prozesse versandt bzw. von verschiedenen Prozessen empfangen. Dabei geht MPI von einer zusammenhängenden Speicherung der Daten aus und es werden aufeinanderfolgende Teile des zu Grunde liegenden Speicherbereiches versandt oder empfangen. Diese Aufteilung der Daten entspricht jedoch oft nicht der Intention des Anwendungsprogrammierers, der vielfältige Möglichkeiten zur Bestimmung der Datenverteilung hat. Bereits beim Verteilen einer Matrix per *scatter* wird üblicherweise eine Verteilung der Matrix*elemente* (spalten- oder zeilenweise) gewünscht sein, während die Verwaltungsinformation der Matrix allen Prozessen zur Verfügung gestellt werden soll. Schon hier ist eine individuelle Anpassung der Verteilungsstrategie notwendig.

Wie aus dem obigen Beispiel ersichtlich, kann keine allgemeine Implementierung der so arbeitenden Kommunikationsfunktionen angegeben werden, ohne den Anwendungsentwickler in seinem Entscheidungsspielraum zu beschneiden. Daher wurde im vorliegenden Paket auf die Implementierung dieser Funktionen verzichtet.

Die folgenden Kommunikationsfunktionen sind somit in dem vorliegenden Paket implementiert.

- `MPI_Pack`, `MPI_Unpack`
- Point-to-point communication:
  - `MPI_Send`, `MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`
  - `MPI_Isend`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend`
  - `MPI_Recv`

- Collective Communication:
  - MPI_Bcast

Alle Arten von C-XSC-Datentypen sind abgedeckt:

- Grunddatentypen: real, interval, complex, cinterval
- Vektor- und Matrix-Datentypen
- Staggered- (bzw. Multiple Precision)-Datentypen
- Dotprecision- (bzw. „Akkumulator“-) Datentypen

Zusätzlich wurden Versionen der oben genannten Funktionen erstellt, die den Versand von Teilmatrizen, insbesondere Zeilen und Spalten von Matrizen und deren Einbettung in Matrizen beim Empfang erlauben.

Dabei wird je nach Datentyp ein unterschiedliches Vorgehen implementiert. Für die Grunddatentypen werden MPI-Datentypen zur Verfügung gestellt:

C-XSC-Klasse	Neuer MPI-Datentyp
real	MPI_CXSC_REAL
interval	MPI_CXSC_INTERVAL
complex	MPI_CXSC_COMPLEX
cinterval	MPI_CXSC_CINTERVAL

Die MPI-Datentypen sind im technischen Sinn Werte des Aufzählungstyps MPI_Datatype, denen intern die bei der Definition erstellte *type map* zugeordnet wird.

Für die auf diesen Datentypen aufbauenden Vektor-, Matrix- und Multiple Precision-Datentypen sowie für die Dotprecision-Datentypen wird das Packing/Unpacking verwendet.

Die Grunddatentypen können nach der Definition direkt als Parameter in den von MPI zur Verfügung gestellten Kommunikationsfunktionen benutzt werden. Für sie gelten somit nicht die oben genannten Einschränkungen auf bestimmte Funktionen. Für die anderen Datentypen kommen die

obigen Einschränkungen jedoch zur Anwendung, da für sie neue Implementierungen zur Verfügung gestellt werden.

Die Implementierung der Kommunikation mit C-XSC-Datentypen wurde in zwei Schritten vorgenommen.

1. Zuerst wurden für alle betrachteten Datentypen die Funktionen

   `MPI_Pack` und `MPI_Unpack`

   überladen. Da die C-XSC-Datentypen nicht auf Template-Basis implementiert sind, ist auch bei den Implementierungen der Funktionen `MPI_Pack` und `MPI_Unpack` keine direkte generische Implementierung möglich.

2. Aufbauend auf diesen Implementierungen wurden dann die jeweiligen Kommunikationsfunktionen überladen. Hierbei konnten Template-Versionen implementiert werden, da alle typspezifischen Aktivitäten in die Vorgänge des Packens und Entpackens fallen.

Im Folgenden wird ein Implementierungsbeispiel für den Datentyp `ivector` angegeben.

Dafür werden zunächst die Implpementierungen der Funktionen `MPI_Pack` und `MPI_Unpack` angegeben; hierbei handelt es sich um Überladungen speziell für den Typ `ivector`.

Danach werden die Implementierungen der Template-Kommunikationsfunktionen `MPI_Send` und `MPI_Recv` angegeben, die für den Typ `ivector` als Parameter ausgeprägt werden können.

```
int MPI_Pack (ivector& iv, void* buff, int bufflen,
              int* pos,  MPI_Comm MC)
{
  int err;
  int lb=Lb(iv);
  int ub=Ub(iv);

  if (!MPI_CXSC_TYPES_DEFINED)
     if (err=MPI_Define_CXSC_Types()!=MPI_SUCCESS)
        return err;
```

```
  if (err=MPI_Pack(&lb,1,MPI_INT,buff,bufflen,pos,MC)
          !=MPI_SUCCESS)
    return err;
  if (err=MPI_Pack(&ub,1,MPI_INT,buff,bufflen,pos,MC)
          !=MPI_SUCCESS)
    return err;

  err=MPI_Pack(&iv[lb],ub-lb+1,MPI_CXSC_INTERVAL,buff,
                 bufflen,pos,MC);
  return err;
}
```

MPI_Pack für ivector

```
int MPI_Unpack (void* buff, int bufflen, int* pos,
                ivector& iv, MPI_Comm MC)
{
  int err;
  int vlen, lb, ub;

  if (!MPI_CXSC_TYPES_DEFINED)
     if (err=MPI_Define_CXSC_Types()!=MPI_SUCCESS)
        return err;

  if (err=MPI_Unpack(buff, bufflen, pos, &lb, 1,
                     MPI_INT, MC)!=MPI_SUCCESS)
    return err;
  if (err=MPI_Unpack(buff, bufflen, pos, &ub, 1,
                     MPI_INT, MC)!=MPI_SUCCESS)
    return err;
  vlen=ub-lb+1;
  Resize(iv,vlen);
  SetLb(iv,lb);

  err=MPI_Unpack(buff, bufflen, pos, &iv[lb],
                 vlen, MPI_CXSC_INTERVAL, MC);

  return err;
}
```

MPI_Unpack für ivector

```
template<class T>
int MPI_Send(T& Tobj, int i1, int i2, MPI_Comm MC)
{
  int pos=0;
  int err;

  // Call overloaded MPI_Pack for C-XSC data type:

  if (err=MPI_Pack(Tobj, commbuffer,
                   MPI_CXSC_BUFFERLEN, &pos, MC)
          !=MPI_SUCCESS)
    return err;

  err=MPI_Send(commbuffer, pos, MPI_PACKED,
               i1, i2, MC);
  return err;
}
```

MPI_Send - Template

```
template<class T>
int MPI_Recv(T& Tobj, int i1, int i2, MPI_Comm MC,
             MPI_Status* MS)
{
  int pos=0;
  int err;

  if (err=MPI_Recv(commbuffer, MPI_CXSC_BUFFERLEN,
                   MPI_PACKED, i1, i2, MC, MS)
          != MPI_SUCCESS)
    return err;

  // Call overloaded MPI_Unpack:
  err=MPI_Unpack(commbuffer, MPI_CXSC_BUFFERLEN,
                 &pos, Tobj, MC);
  return err;
}
```

MPI_Recv - Template

Zum Packen und Entpacken wird ein Datenpuffer benötigt, in den die Daten kopiert werden. Hierzu stellt das Paket ein Feld namens `commbuffer` zur Verfügung, dessen Länge durch den Wert der Konstanten `MPI_CXSC_BUFFERLEN` bestimmt wird. Der Wert kann vom Anwender des Pakets verändert werden. Zusätzliche Flexibilität und geringerer Speicherplatzbedarf wäre hier möglich durch individuelles Anlegen von Puffern der jeweils benötigten Länge in den jeweiligen Funktionen. Hierzu wurden in [116] experimentelle Untersuchungen vorgenommen. Tests mit derartigen Vorgehensweisen führten jedoch vor allem bei der Kommunikation großer Objekte mit hohem Speicherbedarf zu Instabilitäten in den Programmen; offenbar wurde hier das Speichermangagement der verwendeten Systeme in zu hohem Maße beansprucht. Daher wurde in der vorgestellten Version auf die Implementierung ähnlicher Verfahrensweisen verzichtet.

Die obige Implementierung zeigt, dass beim Packen und Entpacken der Daten die neu definierten MPI-Datentypen für die C-XSC-Grunddatentypen verwendet werden. In den Funktionen muss daher geprüft werden, ob die jeweilige Definition bereits durchgeführt wurde. Sobald die jeweiligen Daten gepackt sind, wird die eigentliche Kommunikationsfunktion mit dem Datentyp `MPI_PACKED` aufgerufen. Umgekehrt erhält der Empfänger ein Paket von Daten des Typs `MPI_PACKED` und entpackt danach die entsprechenden Daten für das zugehörige Objekt.

Als weitere Eigenschaft der Implementierungen ist auf die Größenanpassung der Vektoren mittels `Resize` hinzuweisen. Nur hierdurch hat der Anwender die Gewähr, dass die Kommunikation auch dann erfolgreich durchgeführt wird, wenn der Empfängerprozess ein Objekt zum Empfang verwendet, das nicht die gleiche Größe wie das tatsächlich empfangene Objekt besitzt - eine Eigenschaft, die häufig dann benötigt wird, wenn Datenobjekte nur von einzelnen Prozessen aktiv benutzt und damit auch korrekt initialisiert werden, von anderen jedoch nicht, so dass hier oft noch kein ausreichender Speicherplatz zur Verfügung steht.

In der oben gezeigten Implementierung liegen die Vektordaten als zusammenhängendes Feld vor. Beim Packen und Entpacken ist damit nur ein einzelner Arbeitsschritt notwendig, um diese Daten zu verarbeiten. Bei anderen Datentypen dagegen ist dies nicht mehr möglich; dies gilt insbesondere für Vektoren und Matrizen, deren Elemente Objekte eines

Multiple-Precision-Datentyps sind. Zur Verdeutlichung wird die Funktion `MPI_Pack` noch einmal für den Typ `l_ivector` angegeben.

```
int MPI_Pack (l_ivector& l_iv, void* buff,
              int bufflen, int* pos,  MPI_Comm MC)
{
  int err;
  int lb=Lb(l_iv);
  int ub=Ub(l_iv);
  if (!MPI_CXSC_TYPES_DEFINED)
     if (err=MPI_Define_CXSC_Types()!=MPI_SUCCESS)
        return err;

  if (err=MPI_Pack(&lb,1,MPI_INT,buff,bufflen,pos,MC)
          !=MPI_SUCCESS)
    return err;
  if (err=MPI_Pack(&ub,1,MPI_INT,buff,bufflen,pos,MC)
          !=MPI_SUCCESS)
    return err;

  for (int i=lb;i<=ub;i++)
    {
      int prec=StagPrec(l_iv[i]);

      if (err=MPI_Pack(&prec,1,MPI_INT,buff,
                       bufflen,pos,MC)
              !=MPI_SUCCESS)
        return err;
      if (err=MPI_Pack(&l_iv[i][1],prec+1,
                       MPI_CXSC_REAL,buff,
                       bufflen,pos,MC)
              !=MPI_SUCCESS)
        return err;
    }
  return err;
}
```

`MPI_Pack` für `l_ivector`

Anwendung

Die neu definierten Routinen können annähernd in der gleichen Weise benutzt werden wie die bestehenden MPI-Routinen.

Ein Objekt (z.B. ein `ivector iv`) kann an einen Prozess `dest` verschickt werden durch den Aufruf

```
MPI_Send (iv, dest, tag, comm);
```

mit einem *tag* und einem *communicator*[5] als weitere Parameter.

Die Unterschiede zum bestehenden MPI-Interface bestehen darin, dass zwei Parameter der Funktionen entfallen:

1. Es wird auf die Angabe des MPI-Datentyps als Parameter verzichtet. Dies kann geschehen, da der Datentyp automatisch durch den Typ des übergebenen Objekts bestimmt wird. Im Vergleich zu den bestehenden MPI-Funktionen wird hier eine Referenz auf das entsprechende Objekt als Parameter übergeben und kein (nicht typisierter) Pointer.

2. Der Parameter für die zu verschickende Anzahl von Elementen entfällt. Während es bei den bestehenden MPI-Datentypen möglich ist, zusammenhängende Felder von Werten dieses Typs anzulegen, ist dies bei Objekten einer Klasse insbesondere dann nicht mehr möglich, wenn diese dynamischen Speicherplatz verwendet. Dann lassen sich zwar Felder von Referenzen oder Pointern auf Objekte dieser Klassen anlegen, deren Attribute jedoch sind oft in anderen Speicherbereichen abgelegt. Somit erscheint es als nicht sinnvoll, Felder solcher Objekte in den implementierten Kommunikationsfunktionen zuzulassen. Es stehen jedoch Kommunikationsfunktionen für STL-Vektoren zur Verfügung (siehe Abschnitt 5.3.4).

Da die C-XSC-Grunddatentypen als MPI-Datentypen implementiert sind, muss für diese zunächst das bestehende MPI-Interface bei der Kommunikation verwendet werden. Es wurden jedoch Spezialisierungen der implementierten Template-Funktionen erstellt, die es ermöglichen, auch für diese Datentypen die neuen Interface-Versionen zu verwenden. Dabei entfällt

[5] Zur Erklärung der Begriffe *tag* und *communicator* vgl. [92].

der Prozess des Packens/Entpackens, da Objekte dieser Datentypen bereits zusammenhängend gespeichert sind.

Eine Variable `r` des C-XSC-Typs `real` kann also verschickt werden als

```
MPI_Send (r, dest, tag, comm);
```

oder

```
MPI_Send (&r, 1, MPI_CXSC_REAL, dest, tag, comm);
```

Um zu testen, ob durch die Implementierung der Kommunikationsfunktionen als Templates die Effizienz des Pakets leidet, wurde eine weitere Version des Pakets entworfen, die Implementierungen der Grundfunktionen

- `MPI_Send`, `MPI_Isend` und
- `MPI_Recv`

ohne Benutzung von Templates zur Verfügung stellt. Diese Version wird in [46] beschrieben und getestet. Die Tests führten zu dem Ergebnis, dass die Effizienz durch die Verwendung von Templates nicht nachweisbar leidet. Daher wurde hier stets die Template-Version verwendet.

5.3.4 Erweiterung für Taylorarithmetik und STL-Vektoren

Über die C-XSC-Datentypen hinaus wurden auch MPI-Kommunikationsfunktionen für weitere Datentypen realisiert. Es handelt sich hierbei um

- die Klassen der ein- und zweidimensionalen Taylorarithmetik in C-XSC, die in Abschnitt 5.2 beschrieben werden sowie um
- den Template-Datentyp `vector` aus der *C++ Standard Template Library* [79, 123].

Die Datentypen

`itaylor`, `dim2taylor` und `dim2taylor_vector`

der Taylorarithmetik enthalten dabei Objekte von C-XSC-Datentypen oder Felder derselben. Daher ist das Paket als Erweiterung des Kommunikationspakets für C-XSC-Datentypen konzipiert.

Die grundsätzliche Vorgehensweise bei der Implementierung folgt den Darstellungen für die C-XSC-Datentypen im vorangehenden Abschnitt. Bei den Implementierungen kommt in diesem Fall jeweils das *Packing/Unpacking* zum Einsatz, da es sich nicht um einfache Datentypen handelt, die aus zusammenhängenden Speicherbereichen bestehen.

Die Implementierung für STL-Vektoren setzt im Gegensatz zu den Implementierungen für C-XSC-Datentypen auch die Funktionen

`MPI_Pack` und `MPI_Unpack`

als Template-Funktionen um, da es sich bei der Klasse `vector` selbst um eine Template-Klasse handelt.

Für STL-Vektoren gilt wie bei einigen C-XSC-Klassen, dass eine sichere Implementierung nur dann möglich ist, wenn alle Elemente dieser Vektoren einzeln behandelt werden. Dies gilt umsomehr, da als Elemente des Vektors Elemente unbekannter Klassen vorkommen. Je nach Eigenschaften der Element-Datentypen kann sonst das Speichermanagement der STL-Vektoren gestört werden, etwa dann, wenn zunächst Stadardkonstruktoren für alle Vektorelemente (implizit) aufgerufen werden und später einzelne Elemente mit anderen Konstruktoren neu erstellt werden (und somit möglicherweise auch Speicherplatz unterschiedlicher Größe belegen). Falls der zu verwendende Element-Datentyp die Wahl spezieller Konstruktoren erfordert, kann es nötig sein, separate Spezialisierungen der Kommunikationsfunktionen zur Verfügung zu stellen, um diese erfolgreich nutzen zu können.

Allgemein wird vorausgesetzt, dass geeignete Implementierungen der Funktionen `MPI_Pack` und `MPI_Unpack` für den Datentyp der Vektorelemente existieren. Dies gilt insbesondere dann, wenn Pointer als Vektorelemente verwendet werden. In der vorliegenden Implementierung stehen keine hierauf ausgelegten Versionen von `MPI_Pack` und `MPI_Unpack` zur Verfügung. Es ist allerdings möglich, solche hinzuzufügen.

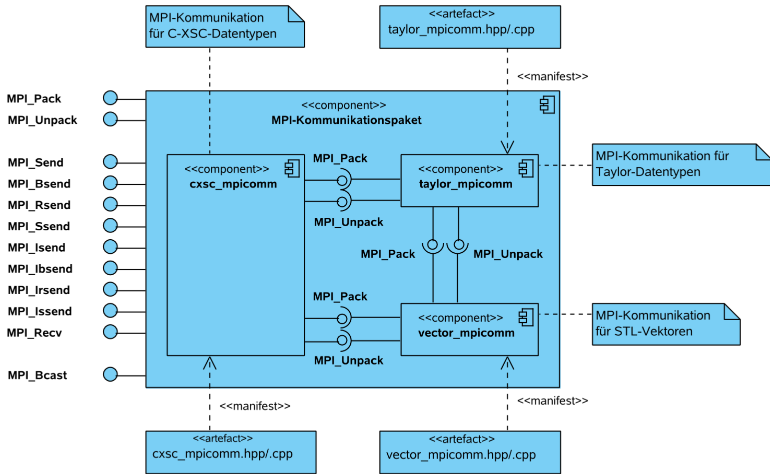

Abbildung 5.1: MPI-Kommunikationspaket

Um die Funktionen für Vektoren zu nutzen, ist es meist erforderlich, eine explizite Instantiierung der Funktionen für die zu verwendenden Element-Datentypen vorzunehmen. Für die Datentypen, die in den Anwendungen dieser Arbeit vorkommen, sind diese bereits enthalten.

In Abbildung 5.1 wird die Struktur der vorgestellten Module dargestellt. Das Modul für die C-XSC-Datentypen ist eigenständig verwendbar, für die Benutzung der Zusatzmodule für Taylor-Objekte und STL-Vektoren wird die Existenz des ersteren vorausgesetzt[6]. Die Module werden daher insgesamt als eine Komponente zur MPI-Kommunikation mit C-XSC und Taylor-Datentypen und STL-Vektoren betrachtet.

[6]Dies liegt daran, dass die Implementierung der Taylorarithmetik auf den C-XSC-Datentypen basiert und dass für die STL-Vektoren Implementierungen der Funktionen `MPI_Pack` und `MPI_Unpack` für C-XSC-Datentypen als Element-Datentypen vorliegen müssen.

5.4 Paralleler Intervall-Gleichungssystem-Löser in C-XSC

In diesem Abschnitt werden Löser für lineare Intervall-Gleichungssysteme beschrieben. Dabei werden zunächst existierende Implementierungen von Gleichungssystem-Lösern angegeben und aufgezeigt, welche Teile der gewünschten Funktionalität in ihnen verwirklicht werden. Insbesondere handelt es sich auch um serielle Löser. Daran anschließend wird die neue Implementierung des in Abschnitt 4.3.1 beschriebenen parallelen verfizierten Lösers vorgestellt und im einzelnen erläutert.

5.4.1 Existierende Implementierungen

Allgemeine Löser für lineare Gleichungssysteme existieren in verschiedensten Implementierungen. Im Rahmen dieses Abschnitts sollen speziell solche Implementierungen angegeben werden, die – wie die nachfolgend vorgestellte neue Implementierung – auf dem Verfahren von Rump basieren und die in Abschnitt 5.1 beschriebenen Werkzeuge verwenden (dabei wird im Hinblick auf C-XSC auch auf weitere XSC-Sprachen Bezug genommen). Dabei handelt es sich sowohl um serielle als auch parallele Implementierungen. Zu jeder Implementierung wird kurz ihr Funktionsumfang im Vergleich zu der gewünschten Funktionalität des hier verwendeten Lösers erläutert.

Folgende Implementierungen sind in diesem Rahmen verfügbar:

Serielle Implementierungen in XSC-Sprachen:

- Implementierung im Rahmen der *Numerical Toolbox for Verified Computing II* in Pascal-XSC [76].
- Implementierung im Rahmen der *C++ Toolbox for Verified Computing* [53] in C-XSC.
- Erweiterung der Implementierung der *C++ Toolbox for Verified Computing* von Hölbig/Krämer [62].

Parallele Implementierungen unter Verwendung von `filib++` und MPI:

- Implementierung von Obermaier im Rahmen von [106].

Implementierungen in Pascal-XSC und C-XSC

Das Verfahren von Rump wurde in unterschiedlichem Rahmen bereits in XSC-Sprachen implementiert. Es handelt sich dabei ausschließlich um serielle Implementierungen. Die neueste dieser seriellen Implementierungen diente als Basis für die hier vorgestellte parallele Implementierung.

Die erste zu nennende Implementierung erfolgte in PASCAL-XSC und steht als Bestandteil der *Numerical Toolbox for Verified Computing II* [76] zur Verfügung.

Die Implementierung umfasste den ersten Teilschritt des Verfahrens von Rump, in dem die erste Lösungseinschließung bestimmt wird. Der zweite Teil, in dem eine verbesserte Einschließung unter Zuhilfenahme einer potentiell numerisch stabileren Matrix berechnet wird, wurde hier nicht implementiert.

Die Strukturierung des Grundmoduls entspricht bereits im Wesentlichen den nachfolgenden Versionen in C-XSC. Bei der verifizierten Berechnung von Skalarprodukt-Ausdrücken mit PASCAL-XSC kommen zusätzlich die dort verfügbaren *#-Ausdrücke* (siehe z.B. [69]) zur Anwendung, mit denen die genaue Berechnung von Skalarprodukt-Ausdrücken im Sinne der *Dotprecision*-Typen (siehe Abschnitt 5.1.1) in besonders einfacher Syntax dargestellt werden kann. Die Einschließung

$$z := b - Ax$$

für eine Intervallmatrix A und Intervallvektoren b und x geeigneter Dimension kann etwa mit Hilfe eines *#-Ausdrucks* in PASCAL-XSC (bei Wahl geeigneter Datentypen) dargestellt werden als

```
z := ##(b - A*x);
```

In C-XSC dagegen ist für die gleiche Operation der folgende Code unter Verwendung eines `idotprecision`-Objekts (und passender Datentypen für die weiteren Objekte) nötig:

```
for (i=Lb(x);i<=Ub(x);i++)
{
  iaccu = b[i];
  accumulate(iaccu,-A[Row(i)],x);
  rnd(iaccu,z[i]);
}
```

Ansätze, diese Ausdrücke in äquivalenter Syntax in C-XSC einzuführen, werden etwa in [9, 136] diskutiert.

Eine C-XSC-Version der oben genannten Implementierung wurde in die *C++ Toolbox for Verified Computing* [53] integriert. Die Implementierung weist den gleichen Umfang wie die zuvor beschriebene PASCAL-XSC-Implementierung auf, wobei der oben diskutierte Unterschied zwischen PASCAL-XSC und C-XSC zum Tragen kommt.

Eine erweiterte Version der C-XSC-Implementierung stellt die Implementierung von Hölbig und Krämer in [62] dar. Hier wurde zusätzlich der zweite Schritt des Rump-Verfahrens, die zweite verifizierte Iteration bei unzureichendem Ergebnis des ersten Schritts, implementiert. Zusätzlich bietet die Implementierung die Möglichkeit, lineare Gleichungssysteme zu lösen, deren Ausgangsdaten Intervalldaten sind. Bei den vorausgehenden Versionen werden zwar verifizierte Lösungen berechnet, die Ausgangsdaten werden jedoch als reell vorausgesetzt.

Der Aufbau der obigen C/C++-Versionen bildet die Basis für die parallele Implementierung und wird aus diesem Grund kurz erläutert. Grundsätzlich kann der Löser als ein Modul konzipiert werden, das technisch durch eine Quellcodedatei mit zugehörigem Header zur Bereitstellung der Interface-Informationen realisiert wird. Gleichzeitig lässt sich jedoch mit der Näherungsinversion einer Matrix ein Teilproblem finden, das auch separat von Bedeutung ist. Es ist somit sinnvoll, dieses Teilverfahren als separates Modul zu konzipieren, das intern vom Gleichungssystem-Löser verwendet wird. Da die oben beschriebenen Implementierungen jeweils serielle Implementierungen sind, stehen alle auszuführenden Grundoperationen bereits durch die verwendeten Klassen aus C-XSC zur Verfügung. Im Gegensatz zu der nachfolgend vorgestellten parallelen Version ist daher keine besondere Betrachtung der Grundoperationen notwendig.

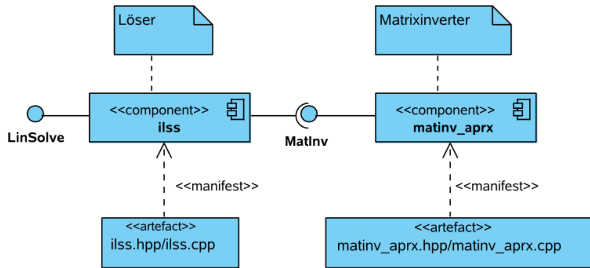

Abbildung 5.2: Intervall-Löser: Komponenten

Der Aufbau des Moduls kann also am Beispiel der Version für reelle Intervall-Eingabedaten in den in Abbildung 5.2 dargestellten Komponenten erfolgen.

Implementierung von Obermaier

Eine weitere Implementierung des Verfahrens von Rump liegt mit der Implementierung von Obermaier im Rahmen von [106] vor. Diese Implementierung erfolgte nicht in PASCAL-XSC bzw. C-XSC, sondern unter Verwendung der Bibliothek `filib++`. Es handelt sich um eine parallele Implementierung; zur parallelen Kommunikation wird MPI verwendet. Das Verfahren wird in der Implementierung von Obermaier nicht zur Lösung einzelner linearer Gleichungssysteme verwendet, sondern zur verifizierten Bestimmung der Inversen einer Intervallmatrix, d.h. zur parallelen Lösung von $n \in \mathbb{N}$ linearen Intervall-Gleichungssystemen.

Die Implementierung umfasst den ersten Teil des Verfahrens von Rump; dabei wird allerdings die in den oben beschriebenen Implementierungen durchgeführte Iteration zur Verbesserung des Startwertes nicht vorgenommen.

Die Parallelisierung umfasst die parallele Berechnung der Näherungsinversen R der Mittelpunktsmatrix der Ausgangs-Intervallmatrix $[A]$ und die parallele Berechnung des Matrixprodukts $R \cdot [A]$. Bei der Bestimmung der Inversen wird eine zeilenzyklische Verteilung der Matrix so-

wie Zeilenpivotsuche verwendet. Die in jedem Schritt der LU-Zerlegung durchzuführende Pivotisierung wird durch explizites Vertauschen der Matrixdaten erreicht. Im Gegensatz zur impliziten Pivotisierung, die in Verfahren 4.3.4 verwendet wird, fällt hierbei zusätzlicher Aufwand für die Vertauschungen an. Gleichzeitig wird auf das *Send-Ahead-Prinzip* (siehe Bemerkung 4.3.3) verzichtet, wodurch vergleichweise längere Totzeiten einzelner Prozesse entstehen und der Overhead des parallelen Verfahrens erhöht wird.

Da die Bibliothek `filib++` verwendet wird, besteht in der Implementierung nicht die Möglichkeit, das in den *Dotprecision*-Datentypen implementierte exakte Skalarprodukt zu verwenden. Ebenso müssen hier Matrizen und Vektoren von Hand erstellt und bearbeitet werden, da keine Matrix- und Vektor-Klassen zur Verfügung stehen.

Ebenso wird die Kommunikation mit MPI von Hand durchgeführt, so dass die Kommunikation ohne Verwendung eines allgemein verwendbaren Interfaces, wie in Abschnitt 5.3 vorgestellt, stattfindet. Um bei der Kommunikation von Hand Vereinfachungen zu erzielen, werden mehrfach Matrix-Transpositionen zusätzlich durchgeführt, die im eigentlichen Verfahren nicht relevant sind und die Übersicht erschweren.

Der Gleichungssystem-Löser ist als Bestandteil eines Teilabschnitts im Verfahren von Obermaier integriert und kann nicht als separates Modul verwendet werden wie die oben beschriebenen XSC-Implementierungen und die nachfolgend vorgestellte neue Implementierung.

5.4.2 Neue parallele Implementierung

In diesem Abschnitt wird die Neuimplementierung des parallelen Lösers für quadratische lineare Intervall-Gleichungssysteme vorgestellt, der in Abschnitt 4.3.1 beschrieben wird.

Das Verfahren gliedert sich in verschiedene Teile, die getrennt von einander betrachtet werden können und unterschiedliche Funktionalitäten zur Verfügung stellen (siehe auch Verfahren 4.3.2, 4.3.3 und 4.3.4). Da die betreffende Funktionalität jeweils auch unabhängig von den anderen Teilen von mathematischer Bedeutung ist, ist es sinnvoll, dies in der Struktur der Implementierung zu berücksichtigen. Gleichzeitig erscheint es als nicht

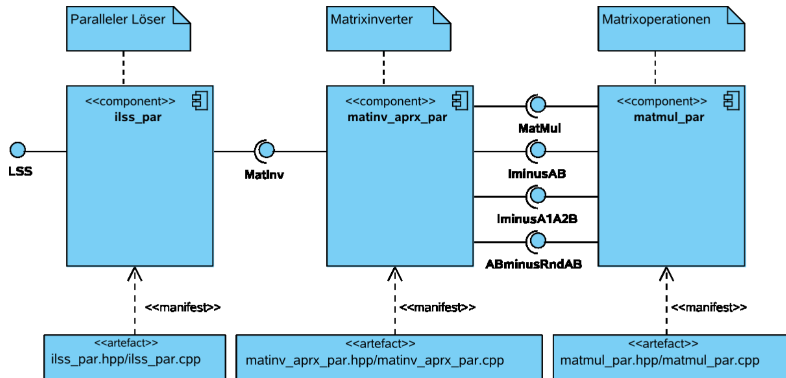

Abbildung 5.3: Paralleler Intervall-Löser: Komponenten

sinnvoll, die Funktionalität des Gleichungssystem-Lösers und seiner Teile als Klassen im Sinne der Objektorientierung zu gestalten, da durch die vorliegenden Verfahren eine funktionale Gestaltung der Implementierung nahegelegt wird.

Die Gestaltung des parallelen Gleichungssystem-Lösers erfolgte demgemäß in den folgenden Komponenten, die als separate Übersetzungseinheiten mit zugehörigen Headerdateien strukturiert wurden:

- Paralleler Gleichungssystem-Löser (Paralleler Gesamtalgorithmus als Funktion, Interface)
- Komponente für näherungsweise reelle Matrixinversion (Paralleler Algorithmus als Funktion, Interface für Inversion)
- Komponente für Matrixmultiplikation und verwandte Operationen (Parallele Algorithmen jeweils als Funktion, Interface)

Damit wird mit der parallelen Implementierung die Struktur der seriellen Löser in XSC-Sprachen, die im vorausgehenden Abschnitt beschrieben wurden, fortgeführt bzw. erweitert. Die Struktur ist auch in Abbildung 5.3 dargestellt.

Im Folgenden werden die bereitgestellten Interfaces angegeben und einige Aspekte der Implementierungen genauer betrachtet.

Interface (Gleichungssystem-Löser)

Zur Benutzung des parallelen Lösers werden mehrere Interface-Versionen bereitgestellt, die die im Folgenden genannten Funktionalitäten abdecken[7].

```
void LSS(imatrix& A, ivector& b, ivector& Y,
         int& errc, int& commerrc, ofstream& ausg)
```

Diese Version beinhaltet folgende Parameter:

`A`, `b`	Matrix und rechte Seite des Gleichungssystems
`Y`	Einschließung der Lösung
`errc`, `commerrc`	Fehlerparameter
`ausg`	Ausgabedatei als Filestream

Damit handelt es sich um eine nur gerinfgügig erweiterte Version auf Basis des Interface der seriellen Vorgänger-Implementierung von Hölbig und Krämer.

Folgende Funktionalität wird dabei neben der Lösung des Gleichungssystems von der Funktion eigenständig gehandhabt:

- Bestimmung der Anzahl von Prozessen innerhalb der parallelen Umgebung und Vergabe von Prozess-Identifikationsnummern (*Prozess-IDs*).
- Verteilung der Matrixdaten auf die Prozesse

Damit ist dieses Interface für die eigenständige Verwendung zum Lösen linearer Gleichungssysteme geeignet, da keine vorherige Benutzung oder Initialisierung der parallelen Umgebung oder eine Verteilung der Daten auf Prozesse vorausgesetzt wird.

```
void LSS(imatrix& A, ivector& b, ivector& Y,
         int procs, int mypid,
         int& errc, int& commerrc,
         ofstream& ausg)
```

[7] Zur besseren Verständlichkeit werden die frei gewählten Parameternamen der vorliegenden Implementierung mit angegeben.

Eine zweite Interface-Version besitzt folgende zusätzliche Parameter:

`procs`, `mypid`	Anzahl der Prozesse und aktuelle Prozess-ID

Damit wird die Benutzung des Lösers in parallelen Programmen ermöglicht, in denen zwar keine bereits feststehende Verteilung der Matrixdaten vorliegt (d.h. in denen ein Prozess die gesamte Ausgangsmatrix speichert), aber dennoch die Initialisierung der parallelen Umgebung bereits durchgeführt wurde.

```
void LSS( imatrix& A, ivector& b, ivector& Y,
          imatrix& MyA, vector<int>& mycolumns,
          intvector& mycol, bool distribute,
          int procs, int mypid,
          int& errc, int& commerrc,
          ofstream& ausg)
```

Schließlich existiert eine dritte Interface-Version, die es ermöglicht, den Löser in Anwendungen einzusetzen, in denen nicht nur die parallele Umgebung bereits initalisiert ist, sondern in denen zusätzlich auch bereits eine Verteilung der Matrix auf die Prozesse vorliegt oder in denen eine vorweg bestimmte Verteilung zum Einsatz kommen soll. Bei der vorliegenden Implementierung des Lösers wird dabei von einer spaltenzyklischen Verteilung der Daten ausgegangen. Die Parameter könnten jedoch genauso in alternativen Implementierungen für eine nicht zyklische Verteilung der Spalten oder eine Verteilung der Zeilen zum Einsatz kommen. Als zusätzliche Parameter gegenüber dem zuvor beschriebenen Interface sind enthalten:

`MyA`	Dem aktuellen Prozess zugeordnete Spalten der verteilt gespeicherten Matrix `A`
`mycolumns`, `mycol`	Spaltenindizes der Spalten des aktuellen Prozesses und Zuordnung der Spaltenindizes von `A` zu `MyA`
`distribute`	Mittels dieses `bool`-Parameters wird angegeben, ob eine Verteilung erfolgen soll.

Diese letzte Interface-Version ist insbesondere auf die Verwendung in der später vorgestellten C-XSC-Implementierung des Intgralgleichungslösers nach Klein zugeschnitten, aber allgemein bei der gleichartigen Benutzung

von C-XSC-Datentypen verwendbar. Da der Masterprozess die vollständige Matrix benötigt, müssen hier dennoch sowohl der Parameter `A` als auch der Parameter `MyA` übergeben werden.

Interface (Matrixinversion)

Analog zu den oben angestellten Überlegungen für die Interface-Versionen des parallelen Lösers existieren folgende Interface-Versionen der Matrix-Inversionsfunktion `MatInv`, die für die eigenständige Verwendung inclusive Initialisierung der parallelen Umgebung und Datenverteilung sowie die Verwendung in bereits initialisierten parallelen Anwendungen mit oder ohne bereits erfolgte Verteilung der Daten vorgesehen sind. Für die Bedeutung der Parameter wird auf die Interface-Versionen des parallelen Lösers verwiesen.

```
void MatInv(rmatrix& A, rmatrix& R,
            int& errc, int& commerrc,
            ofstream& ausg)
```

Interface mit Initialisierung und Verteilung

```
void MatInv(rmatrix& A, rmatrix& R,
            int procs, int mypid,
            int& errc, int& commerrc,
            ofstream& ausg)
```

Interface ohne Initialisierung und mit Verteilung

```
void MatInv(rmatrix& A, rmatrix& R,
            rmatrix& MyA, vector<int>& mycolumns,
            intvector& mycol, bool distribute,
            int procs, int mypid,
            int& errc, int& commerrc,
            ofstream& ausg)
```

Interface ohne Initialisierung und Verteilung

Matrix-Operationen

In der Komponente für Matrixoperationen wurden die Matrixmultiplikation sowie eine Reihe von verwandten Operationen implementiert, die im

parallelen Gleichungssystem-Löser von Interesse sind. Sind Matrizen A, A1, A2 der Dimension r×s und Matrizen B, B1, B2 der Dimension s×t gegeben (als Objekte des jeweils angegeben Datentyps entsprechend den unten verwendeten Bezeichnungen) und wird mit I die (ggf. um Nullzeilen oder -spalten ergänzte) Einheitsmatrix passender Dimension verstanden, sind desweiteren die beteiligten Prozesse durch startproc und endproc definiert und der Masterprozess durch root ausgezeichnet, so sind die Interfaces für die nachfolgend genannten Operationen vorhanden:

```
void MatMul(imatrix& A, imatrix& B, imatrix& R,
            int r, int s, int t,
            int root, int startproc, int endproc,
            int& errc, int& commerrc,
            ofstream& ausg)
```

[R]=[A]·[B]

```
void MatMul(rmatrix& A, rmatrix& B, rmatrix& R,
            int r, int s, int t,
            int root, int startproc, int endproc,
            int& errc, int& commerrc,
            ofstream& ausg)
```

R=A·B

```
void IminusAB(rmatrix& A, imatrix& B, imatrix& R,
              int r, int s, int t,
              int root, int startproc, int endproc,
              int& errc, int& commerrc, ofstream& ausg)
```

[R]=I−A·[B]

```
void IminusA1A2B(rmatrix& A1, rmatrix& A2,
              imatrix& B, imatrix& R,
              int r, int s, int t,
              int root, int startproc, int endproc,
              int& errc, int& commerrc,
              ofstream& ausg)
```

[R]=I−A1·[B]−A2·[B]

Bei der letzten Operation handelt es sich um die Darstellung eines Skalarproduktausdrucks als Summe zweier `real`-Matrizen. Die spezielle Eigenschaft dieser Summe, das Ergebnis der Operation besonders genau wiederzugeben, wird nachfolgend im Abschnitt zu besonderen Vorgehensweisen bei der Implementierung mit C-XSC dargestellt.

```
void ABminusRndAB(rmatrix& A, rmatrix& B,
            rmatrix& C1, rmatrix& C2,
            int r, int s, int t,
            int root, int startproc, int endproc,
            int& errc, int& commerrc,
            ofstream& ausg)
```

C1=#⋆(A·B); C2=#⋆(A·B−C1)[8]

Hinweise zur Parallelisierung

Die Parallelisierung des Lösers erfolgte nach dem Master-Worker-Modell (siehe Abschnitte 2.6 und 4.3.1). Folgende Überlegung kommt bei der so entstehenden Parallelisierung zum Tragen:

Die Anwendung des Master-Worker-Modells beinhaltet auch eine korrespondierende Verteilung der Daten: Der Master verfügt über die vollständigen Daten (insbesondere die der Matrix $[A]$ des Gleichungssystems), die Worker verfügen z.T. nur über Teile der involvierten Daten. Dies hat Vorteile, aber auch Nachteile. Der wesentliche Nachteil besteht darin, dass der Speicherbedarf der Programme hoch bleibt, da die Daten nicht ausschließlich verteilt vorliegen. Speicher-Restriktionen kommen damit schneller zum Tragen (siehe auch Kapitel 6). Bei ausschließlich verteilt vorliegenden Daten wäre der Speicherbedarf niedriger, d.h. die praktisch berechenbare Problemgröße könnte erhöht werden. Jedoch würden dann auch dem Master nur Teile der Daten zur Verfügung stehen. Dies hätte folgende Auswirkungen:

- Es könnten keine Operationen mit den vollständigen Daten seriell ausgeführt werden. Es entstünde somit zusätzlicher Kommunikationsaufwand bei der verteilten Berechnung dieser Operationen. Berechnungen des Aufwands $O(n^2)$, deren Aufwand zunächst nicht

[8] #-Ausdruck gemäß der Notation in PASCAL-XSC.

ins Gewicht fällt, würden hierdurch Kommunikations-Overhead erzeugen, der sich gerade bei ansonsten wenig aufwändigen Operationen negativ bemerkbar machen kann.

- Das Verfahren enthält einige Iterationen. Iterationsverfahren haben die für die Parallelisierung ungünstige Eigenschaft, dass in jedem Schritt der Iteration die Ergebnisse aus dem jeweils vorausgehenden Schritt vorliegen müssen. Um dies für verteilte Daten zu erreichen, entsteht weiterer erheblicher Kommunikationsbedarf, indem jedem Prozess die entsprechenden Daten mitgeteilt werden müssen. Dabei müssen speziell viele einzelne Datenpakete verschickt werden. Wie in Kapitel 6 belegt, kann aber gerade die Kommunikation vieler einzelner, unabhängiger Datenpakete sehr ineffizient sein und ist daher zu vermeiden.

Aus den obigen Erwägungen wurde insgesamt darauf verzichtet, ausschließlich mit verteilten Daten zu arbeiten, auch wenn es Testfälle gab, in denen die genannten Nachteile restriktiv wirkten. Zur Verwendung der Teilkomponenten in Anwendungen, in denen die benötigten Daten ausschließlich verteilt vorliegen, sind somit geeignete Modifizierungen vorzunehmen.

Besondere Vorgehensweisen bei der Implementierung mit C-XSC

Bei der Implementierung des Lösers kommen C-XSC und das MPI-Kommunikationspaket für C-XSC-Datentypen aus Abschnitt 5.3 zum Einsatz.

Die Verwendung von C-XSC ermöglicht die folgende Vorgehensweise zur besonders genauen Darstellung von reellen Näherungen:

Findet eine Berechnung eines Skalarprodukt-Ausdrucks mit Hilfe eines Dotprecision-Objekts statt, so wird der entstehende Ausdruck abschließend meist in eine Variable des Grunddatentyps gerundet. Alle in der Berechnung vorkommenden Operationen werden dann zwar exakt ausgeführt, jedoch kann der bei der abschließenden Rundung auftretende Fehler dennoch das Ergebnis in relevanter Weise beeinflussen. Um eine Darstellung der doppelten Genauigkeit des ursprünglichen Rundungsergebnisses zu erhalten, kann wie folgt vorgegangen werden:

- Das exakte Ergebnis der Berechnung liege in A vor[9].
- Runde A nach R_1.
- Berechne das exakte Ergebnis von $A - R_1$ und runde es nach R_2.

Da der oben berechnete Ausdruck $A - R_1$ nahe bei 0 liegt, wird der absolute Rundungsfehler in der IEEE-754-Arithmetik sehr gering ausfallen. Es ist also zu erwarten, dass $R = R_1 + R_2$ eine bessere Näherung an A darstellt als R_1 allein.

Ein weiterer Aspekt, dessen Betrachtung bei parallelen Berechnungen mit C-XSC notwendig ist, liegt im Zusammenhang zwischen der Verwendung von Dotprecision-Objekten und paralleler Kommunikation. Sollen exakte Berechnungen von Skalarproduktausdrücken auf mehrere Prozesse verteilt durchgeführt werden, so muss das Dotprecision-Objekt, das das Ergebnis oder Zwischenergebnis enthält, während der Berechnung zwischen den beteiligten Prozessen versendet werden. Wie in Abschnitt 5.1.1 bereits gezeigt, belegt etwa ein Objekt des Typs `dotprecision` den 66,125-fachen Speicherplatz einer `double`- bzw. `real`-Variablen. Damit erhöht sich bei Kommunikation von Dotprecision-Objekten auch das Kommunikationsvolumen um den Faktor 66,125.

Um auf die Kommunikation von Dotprecision-Objekten verzichten und dennoch exakte Operationen verwenden zu können, ist eine Verteilung der Daten anzustreben, die es erlaubt, dass exakte Operationen jeweils seriell von einem Prozess durchgeführt werden können. Am Beispiel einer Matrixmultiplikation $A \cdot B$ etwa bedeutet dies, dass A zeilenweise, B jedoch spaltenweise verteilt vorliegen muss, um je ein Element der Ergebnismatrix vollständig von einem einzelnen Prozess berechnen lassen zu können. Diese Verfahrensweise wurde in der vorliegenden Implementierung gewählt. Es muss dabei eine erneute Verteilung der nicht zeilenweise vorliegenden Daten in Kauf genommen werden, um den oben beschriebenen 66,125-fachen Kommunikationsaufwand bei der Kommunikation mit `dotprecision`-Objekten zu vermeiden.

[9] Für diese Betrachtung ist es unerheblich, ob insgesamt die Berechnung eines Skalars, eines Vektors oder einer Matrix betrachtet wird.

5.5 Integralgleichungslöser nach Klein

In diesem Abschnitt wird die Implementierung des Integralgleichungslösers nach dem Verfahren von Klein vorgestellt. Die zugehörigen Verfahren und Algorithmen werden in den Abschnitten 3.1 und 4.2 beschrieben.

Der Abschnitt zerfällt in die folgenden Teile:

- Beschreibung der existierenden Implementierungen von Klein
- Vorstellung der neuen seriellen Implementierungen für einzelne Integralgleichungen und Systeme
- Vorstellung der neuen parallelen Implementierungen

Die neuen seriellen und parallelen Implementierungen in C-XSC verwenden die Softwarekomponenten, die in den vorausgehenden Abschnitten dieses Kapitels vorgestellt wurden.

5.5.1 Existierende Implementierung

Klein stellt in [72] zusammen mit den Verfahren auch eine Implementierung vor. Die zur Verfügung stehenden Programme umfassen serielle Implementierungen der in 3.1 beschriebenen Verfahren.

Diese Implementierungen bilden den Ausgangspunkt für die Entwicklung der neuen seriellen Implementierungen.

Die Implementierung erfolgte in PASCAL-SC [83, 101]. Der Code wurde auf verschiedene Teile aufgeteilt, die jedoch hauptsächlich die Übersicht erleichtern, da es sich um eine funktionale Sprache handelt, die nicht die Möglichkeiten der Datenkapselung und des objektorientierten Design bietet wie eine moderne objektorientierte Sprache.

Auf eine Darstellung der Programmteile als Diagramm wird verzichtet, da ein rein funktionaler Aufbau vorliegt und die Programmteile in zahlreichen unterschiedlichen Arten und Weisen aufeinander zugreifen. Insbesondere wird die strukturelle Modularisierung durch viele globale Variablen, Konstanten und global verfügbare Datenstrukturen eingeschränkt.

Es existieren zwei voneinander unabhängige Programmversionen für

- einzelne Integralgleichungen und für
- Integralgleichungssysteme.

Die im Folgenden beschriebenen Programmteile sind jeweils in eigenen Übersetzungseinheiten vorhanden.

Hauptprogramm und Berechnungsverfahren für Integralgleichungen

Dieser Programmteil stellt jeweils (für Integralgleichungen und Integralgleichungssysteme) den Kern der Anwendung dar. Er enthält das eigentliche Programm, das sich der anderen Programmteile bedient.

Es werden durchgeführt:

- Das interaktive Einlesen der Eingabewerte. Bei der Verarbeitung dieser Werte wird auf den in zwei separaten Programmteilen zur Verfügung stehenden Formelparser zurückgegriffen; die entsprechenden Datenstrukturen stehen global zur Verfügung.
- Die Taylorentwicklung der Funktionen mit Hilfe der Funktionen der Taylorarithmetik aus dem gleichnamigen separaten Programmteil, ebenfalls mit Hilfe globaler Datenstrukturen.
- Die Berechnung der Lösungen. Dabei folgt die Einteilung der Unterprogramme der Einteilung des Verfahrens, die in den Verfahren 3.1.1-3.1.5 dargestellt ist.

Zur Lösung des linearen Intervall-Gleichungssystems wird der Löser verwendet, der später in die *Numerical Toolbox for Verified Computing* [52] einging.

Dieser Programmteil enthält auch das Hauptprogramm aus implementierungstechnischer Sicht. Das Hauptprogramm ist fest in diesen Teil integriert. Es ist damit nicht ohne Eingriff in den Programmcode möglich, die Funktionalität des Programms in anderen Anwendungen wiederzuverwenden.

Formelparser

Die Implementierung enthält, auf zwei Übersetzungseinheiten verteilt, einen Formelparser und die zugehörige Verwaltung von Formelausdrücken. Der eine Teil ist für das eigentliche Lesen von Strings und deren Interpretation zuständig, der andere Teil stellt Datenstrukturen und Funktionen zum Aufbau der Datenstruktur des Formelbaums zur Verfügung. Der Zugriff durch eine hierfür vorgesehene Funktion ist so gestaltet, dass jeweils direkt vom Bildschirm oder aus einer Eingabedatei gelesen wird, eine Übergabe von Zeichenketten als Parameter bzw. eine Nutzung als unabhängiges Modul ist nicht ohne Modifikationen möglich.

Die folgenden Operatoren und Funktionen können verarbeitet werden:

- Arithmetische Grundoperatoren
- Quadrat- und Wurzelfunktion, Potenzen
- Exponentialfunktion und Logarithmus
- Einige trigonometrische Funktionen: $\sin, \cos, \sinh, \cosh, \arctan$

Taylorarithmetik

Auch eine Taylorarithmetik ist als eigenständige Übersetzungseinheit vorhanden. Es können ein- und zweidimensionale Taylorentwicklung durchgeführt werden, wobei dieselben globalen Datenstrukturen verwendet werden. Es sind dieselben Operatoren und Funktionen abgedeckt wie beim Formelparser. Auch dieser Programmteil ist nicht direkt eigenständig verwendbar.

Globale Typdefinitionen und Hilfsfunktionen

Zusätzlich zu den obigen Teilen wird separat eine Anzahl von Hilfsfunktionen und Strukturen bereitgestellt. Diese umfassen:

- Ein- und Ausgabefunktionen
- Definition globaler Datenstrukturen
- Operatoren zur Verwendung mit den im Programm definierten Datenstrukturen

- Initialisierungsfunktionen
- mathematische Hilfsfunktionen

und Ähnliches.

Diejenigen Programmteile, die unabhängig von der Verwendung des Einzel- oder Systemverfahrens sind, werden als gemeinsame Module von beiden Anwendungen benutzt. Programmteile, die von der Anwendung abhängen, also insbesondere alle Teile, die Datenstrukturen für das Einzel- oder Systemverfahren zur Verfügung stellen, sind jeweils in zwei Versionen vorhanden.

5.5.2 Neue Implementierung

In diesem Abschnitt wird die neue serielle Implementierung des Verfahrens von Klein für Integralgleichungen vorgestellt. Die neue Implementierung für Integralgleichungssysteme folgt in Abschnitt 5.5.3, die neue parallele Implementierung in Abschnitt 5.5.4.

Das Verfahren für (einzelne) Integralgleichungen wird in den Verfahren 3.1.1, 3.1.2, 3.1.3, 3.1.4 und 3.1.5 angegeben. Diese Verfahren werden in der hier beschriebenen Implementierung umgesetzt.

Zunächst soll die Struktur der Software und der benutzten Komponenten erklärt werden, danach wird ihre Anwendung beschrieben.

Daraus, dass in der Implementierung die Verfahren 3.1.1-3.1.5, d.h. funktional orientierte Algorithmen umgesetzt werden, ist bereits zu ersehen, dass es sich bei der erstellten Software um eine algorithmisch und damit auch funktional orientierte Anwendung handelt. Dies hat zur Folge, dass das Design der Software nur in eingeschränktem Maße die Verwendung objektorientierter Konzepte zulässt. Dennoch wurden diejenigen Einheiten des Verfahrens, die sich als Klassen abbilden lassen, in diesem Sinne umgesetzt.

Zunächst werden die Klassen der Kernkomponente betrachtet.

Es wurden folgende Klassen mit den nachstehenden Attributen und Methoden angelegt:

- Klasse `IntegralOperator`

  Die Klasse `IntegralOperator` bildet den im Verfahren benötigten Integraloperator K ab, der in den iterativen Verfahren für nicht entartete Kerne in jedem Iterationsschritt zur Anwendung kommt. Als Attribut ist enthalten

  - Die Taylordarstellung des jeweiligen Kerns, gegeben als `ivector`. Im Spezialfall dieser Anwendung kommen hier die Restgliedkoeffizienten der zweidimensionalen Taylorentwicklung der ursprünglichen Kernfunktion zum Einsatz. Statt der vollständigen *zweidimensionalen* Taylordarstellung genügt es daher, diese Restgliedkoeffizienten in einem eindimensionalen Vektor zu speichern.

  Die wesentliche Methode (neben den stets vorhandenen Konstruktoren) ist hier

  - der Auswertungsoperator `operator()`

  Dieser Operator stellt die Anwendung des Integraloperators im Sinne der Anwendung einer Funktion auf einen Parameter dar. Dabei ist der Parameter ebenfalls eine Funktion, die als eindimensionale Taylorentwicklung gegeben ist.

  Objekte der Klasse sind also auch aus programmtechnischer Sicht Funktionsobjekte.

- Klasse `IntegralGleichung`

  Die Klasse `IntegralGleichung` stellt die Hauptklasse der Komponente dar. Sie bildet den Hauptgegenstand des Verfahrens, die Integralgleichung, ab. Die enthaltenen Attribute und Methoden spiegeln insbesondere die in den Verfahren vorkommenden Funktionen wider, sofern diese direkt auf die Integralgleichung bezogen sind (z.B. die Funktion `Loesung_bilden`, nicht dagegen die Iterationsfunktionen, s.u.)

  Als wesentliche Attribute sind enthalten:

  - Kern und rechte Seite als Funktionen, durch Pointer referenziert.

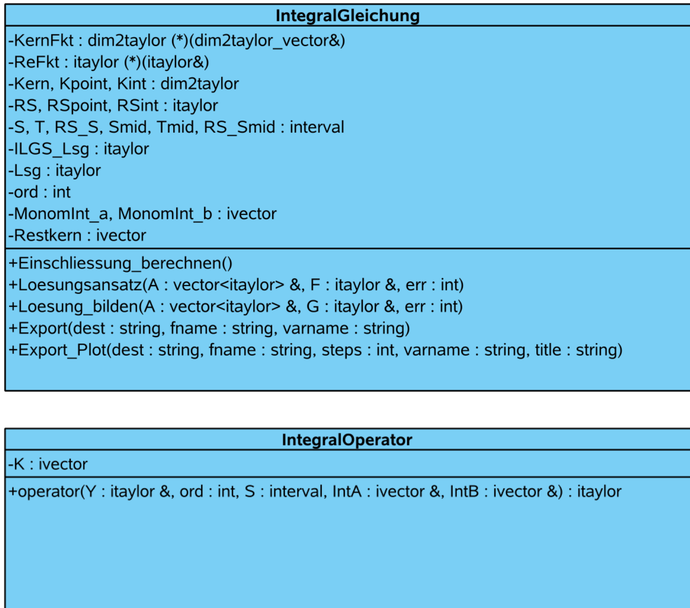

Abbildung 5.4: Integralgleichungslöser: Klassen der Kernkomponente

– Kern und rechte Seite als ein- bzw. zweidimensionale Taylorentwicklungen der Ordnung `ord`

– Definitionsintervalle für Kern und rechte Seite sowie die daraus berechneten Intervallmittelpunkte

– Attribute für die Lösung des linearen Gleichungssystems und die Lösung der Integralgleichung

– Die Ordnung `ord` der verwendeten Taylorentwicklung

– Attribute zur Speicherung der Auswertungen der Monomintegrale nach Verfahren 3.1.2

Die Methoden umfassen

– Konstruktoren, die geeignete Initialisierungen vornehmen

- Zugriffsfunktionen
- die die Verfahren 3.1.1, 3.1.4 und 3.1.5 widerspiegelnden Methoden
  * `Einschliessung_berechnen`
  * `Loesungsansatz`
  * `Loesung_bilden`
- sowie Methoden zum Export von Maple-Sourcecode zur Verwendung der Funktionen und ihrer Visualisierung in Maple mit Hilfe des Pakets `intpakX`.

Die Verfahren 3.1.2 und 3.1.3 werden nicht als Methoden der Klasse `IntegralGleichung`, sondern als freie Funktionen implementiert, da sie keinen mathematisch-logischen Bezug zur Integralgleichung selbst haben. Der Iterationsfunktion wird allerdings der direkte Zugriff auf die Attribute der Klasse gewährt.

Die Klassen sind in Abbildung 5.4 dargestellt[10].

Neben den oben beschriebenen Klassen und Funktionen werden eine Reihe von externen Komponenten verwendet, die wichtige Funktionalitäten zur Verfügung stellen.

- **C-XSC** Zur Darstellung von Intervallen in der Implementierung werden die Datentypen und die zugehörigen Arithmetiken von C-XSC benutzt. Dazu wird C-XSC als Bibliothek ins Programm eingebunden. Die beiden folgenden Komponenten basieren ebenfalls auf C-XSC.
- **Linearer Gleichungssystem-Löser** In den seriellen Implementierungen wird der von Hölbig und Krämer in [62] beschriebene Löser für lineare Intervall-Gleichungssysteme in C-XSC verwendet.

[10] Konstruktoren, Zugriffs- und Hilfsfunktionen werden der Übersicht halber nicht dargestellt.

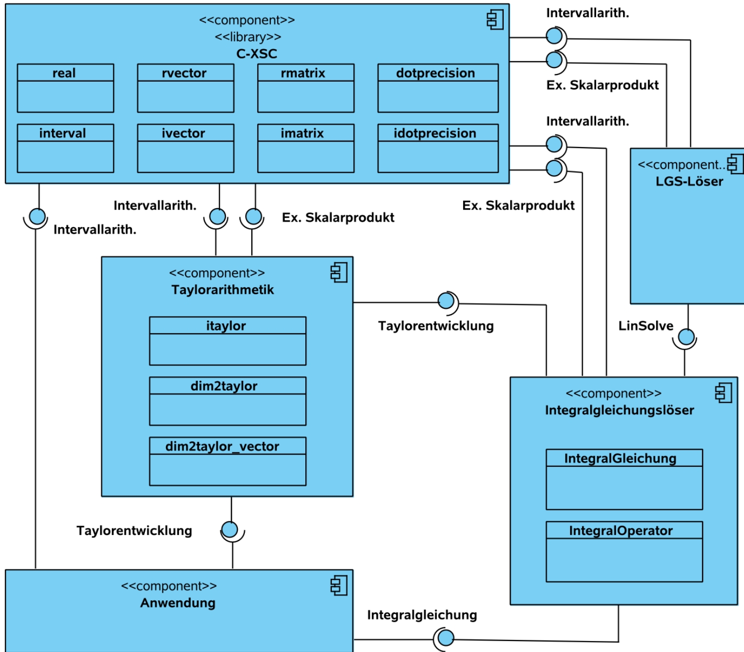

Abbildung 5.5: Integralgleichungslöser: Komponenten

- **Taylorarithmetik** Die in Abschnitt 5.2 beschriebene erweiterte Taylorarithmetik in C-XSC wird zur Darstellung der Funktionen im Integralgleichungslöser verwendet. Zur Benutzung in eigenen Testprogrammen muss der Benutzer lediglich die erforderlichen Funktionen für die Integralgleichung als Code bereitstellen.

Eine Übersicht der Komponenten ist in Abbildung 5.5 zu finden[11].

[11] Die STL wird dabei als integraler Bestandteil der Programmiersprache angesehen und nicht abgebildet.

Anwendung

Die Definition und Lösung einer Integralgleichung mit dem vorliegenden Programm kann somit wie im folgenden Beispiel geschehen.

Beispiel 5.5.1 *In diesem Beispiel soll die Lösung der Integralgleichung*

$$y(s) - \int_0^1 k(s,t)y(t)dt = g(s)$$

mit

$$k : [0,1] \times [0,1] \to \mathbb{IR},\ (s,t) \mapsto \frac{1}{2}(s+1) \cdot \exp(-st)$$

und

$$g : [0,1] \to \mathbb{IR},\ s \mapsto \exp(-s) - \frac{1}{2} + \frac{1}{2} \cdot \exp(-s-1)$$

mit Hilfe des vorgestellten Integralgleichungslösers eingeschlossen werden. Es soll eine Taylorordnung von 6 *gewählt werden.*

Folgende Schritte müssen bei der Erstellung des Testprogramms ausgeführt werden:

- *Einbindung aller notwendigen Interfaces (Headerdateien)*
- *Definition der Kernfunktion und der rechten Seite für die Integralgleichung sowie der Definitionsbereiche*
- *Implementierung eines Hauptprogramms mit Definition der Integralgleichung und Aufruf der Lösungsmethode. (Die Ausgabe ist hierbei bereits integriert.)*

Nachfolgend ist der Programmcode des Testprogramms, gefolgt von der Ausgabe der berechneten Lösung, angegeben.

```
#include "interval.hpp"
#include "dim2taylor.hpp"
#include "itaylor.hpp"
#include "Einzel-FIGL-Loeser.hpp"
using namespace std;
using namespace cxsc;
```

```
using namespace taylor;

dim2taylor K(dim2taylor_vector& x)
{
  return (x[1]+1)*exp(-x[1]*x[2])/real(2.0);
}
itaylor RS(itaylor& s)
{
  return exp(-s)-real(0.5)+real(0.5)*exp(-s-1);
}

const interval KDefS=interval(0.0,1.0);
const interval KDefT=interval(0.0,1.0);
const interval RSDef=interval(0.0,1.0);

int main()
{
  // Taylorordnung
  int ord=6;

  // Integralgleichung definieren
  IntegralGleichung IGL(K,KDefS,KDefT,RS,RSDef,ord);

  // Loesung (incl. Ausgabe)
  IGL.Einschliessung_berechnen();

  return 0;
}
```

Beispielanwendung Integralgleichungslöser

```
Endergebnis:
[itaylor object, order 6:]
[0][ 0.60577860379007186, 0.60728391890997813]
[1][-0.60678974895211036,-0.60625786277932347]
[2][ 0.30274153818543630, 0.30386377565571188]
[3][-0.10559806117187399,-0.09687121404098759]
[4][ 0.01098664924319565, 0.03878467555226311]
[5][-0.01896748609620867, 0.00810107545170247]
[6][-0.00336059418687250, 0.00602060230743478]
```

Beispielausgabe Integralgleichungslöser

Das Endergebnis wird als Objekt der Klasse `itaylor`*, d.h. als Vektor von Taylorkoeffizienten, ausgegeben. Das Objekt kann, falls gewünscht, in weitere Berechnungen mit weiteren Taylor-Objekten eingehen.*

Wird eine Ausgabe als Funktion, d.h. etwa als

$$\begin{aligned}
y(s) \quad = \quad & [0.60577860379007186, 0.60728391890997813] \\
& +[-0.60678974895211036, -0.60625786277932347] \cdot (s - \frac{1}{2}) \\
& +[0.30274153818543630, 0.30386377565571188] \cdot (s - \frac{1}{2})^2 \\
& +[-0.10559806117187399, -0.09687121404098759] \cdot (s - \frac{1}{2})^3 \\
& +[0.01098664924319565, 0.03878467555226311] \cdot (s - \frac{1}{2})^4 \\
& +[-0.01896748609620867, 0.00810107545170247] \cdot (s - \frac{1}{2})^5 \\
& +[-0.00336059418687250, 0.00602060230743478] \cdot (s - \frac{1}{2})^6
\end{aligned}$$

gewünscht, so lässt sich dies mit dem integrierten Maple-Interface des Integralgleichungslösers erreichen, der in Abschnitt 5.6 vorgestellt wird. Dies kann im Beispiel durch Einfügen der folgenden Zeile nach der Berechnung der Lösungseinschließung geschehen, wobei ein geeigneter Name für die Ausgabedatei als Parameter gewählt werden muss:

```
IGL.Export("Maple","maplecode.mpl");
```

Export nach Maple

Gleichzeitig ist auch der Export einer Einschließung des Wertebereichs der Funktion, ausgewertet auf einer durch einen Parameter vorgegebenen Anzahl von Teilintervallen, möglich, so dass der Wertebereich in Maple mit dem Paket `intpakX` *visualisiert werden kann:*

```
IGL.Export_Plot("Maple","maplecode.mpl", 10);
```

Export nach Maple (zur grafischen Darstellung)

Die zugehörige Grafik ist in Abbildung 5.9 auf Seite 189 dargestellt.

5.5.3 Neue Implementierung für Integralgleichungssysteme

Nach der Implementierung für die einzelne Integralgleichung wird jetzt die neue serielle Implementierung des Verfahrens von Klein für Integralgleichungssysteme vorgestellt.

Das Verfahren für Integralgleichungssysteme wird in den Verfahren 3.1.6, 3.1.7, 3.1.8, 3.1.9 und 3.1.10 angegeben. Diese Verfahren werden in der hier beschriebenen Implementierung umgesetzt.

Die Software ist analog zu der Software für die einzelne Integralgleichung strukturiert.

Zur Umsetzung der Vektor- und Matrix-Strukturen gemäß Abschnitt 2.5.3 werden Vektoren der STL verwendet. C-XSC-Datentypen konnten hierfür nicht zur Anwendung kommen, da sie keine Möglichkeit der Definition von Vektoren und Matrizen für beliebige Elementtypen sowie keine höherdimensionalen Strukturen anbieten.

Die Kernkomponente enthält die folgenden Klassen mit den nachstehenden Attributen und Methoden:

- Klasse `IntegralSystemOperator`

  Die Klasse `IntegralSystemOperator` bildet den im Verfahren benötigten Integraloperator K ab, der im Systemverfahren als Vektor gemäß (2.59) dargestellt wird und der in den iterativen Verfahren für nicht entartete Kerne in jedem Iterationsschritt zur Anwendung kommt. Als Attribut ist enthalten

  – eine Matrix der Taylordarstellungen der jeweiligen Kerne, jeweils gegeben als `ivector`. Auch hier kommen die Restgliedkoeffizienten der zweidimensionalen Taylorentwicklung der ursprünglichen Kernfunktionen zum Einsatz und statt der vollständigen *zweidimensionalen* Taylordarstellungen genügt es, diese Restgliedkoeffizienten in eindimensionalen Vektoren zu speichern. Zusätzlich werden die Definitionsbereiche der Kerne gespeichert.

Die wesentliche Methode (neben den stets vorhandenen Konstruktoren) ist auch im Fall des `IntegralSystemOperators`

- der Auswertungsoperator `operator()`

Dieser Operator stellt auch hier die Anwendung des Integraloperators im Sinne der Anwendung einer Funktion bzw. eines Vektors von Funktionen auf einen Parameter dar, wobei hier Vektoren von Funktionen als Parameter auftreten. Somit sind auch Objekte dieser Klasse Funktionsobjekte.

- Klasse `IntegralGleichungsSystem`

  Die Klasse `IntegralGleichungsSystem` stellt die Hauptklasse der Komponente dar. Sie bildet den Hauptgegenstand des Verfahrens, das Integralgleichungssystem, ab.

  Als wesentliche Attribute sind enthalten:

  - Die Kernmatrix und der Vektor der rechten Seiten als Funktionen, durch Pointer referenziert.
  - Die Kernmatrix und der Vektor der rechten Seiten als ein- bzw. zweidimensionale Taylorentwicklungen der Ordnung `ord`
  - Definitionsintervalle für Kerne und rechte Seiten sowie die daraus berechneten Intervallmittelpunkte
  - Attribute für die Lösung des linearen Gleichungssystems und den Lösungsvektor des Integralgleichungssystems
  - Die Ordnung `ord` der verwendeten Taylorentwicklung sowie die Ordnung `sysord` des Integralgleichungssystems
  - Attribute zur Speicherung der Auswertungen der Monomintegrale nach Verfahren 3.1.7

  Die Methoden umfassen

  - Konstruktoren, die geeignete Initialisierungen vornehmen
  - Zugriffsfunktionen
  - die die Verfahren 3.1.1, 3.1.4 und 3.1.5 widerspiegelnden Methoden

```
IntegralGleichungsSystem
-KernFkt : vector<vector<dim2taylor (*)(dim2taylor_vector&)>>
-ReFkt : vector<itaylor (*)(itaylor&)>
-Kern, Kpoint, Kint : vector<vector<dim2taylor>>
-RS, RSpoint, RSint : vector<itaylor>
-Sglob, Tglob, RS_Sglob : interval
-S, T, RS_S, Smid, Tmid, RS_Smid : ivector
-ILGS_Lsg : ivector
-Lsg : vector<itaylor>
-ord : int
-sysord : int
-BasisInt_a, BasisInt_b : imatrix
-Restkern : vector<vector<ivector>>
+Einschliessung_berechnen()
+SystemLoesungsansatz(C : vector<vector<vector<itaylor>>> &, F : vector<itaylor> &, err : int)
+SystemLoesung_bilden(C : vector<vector<vector<itaylor>>> &, G : vector<itaylor> &, err : int)
+Export(dest : string, fname : string, varname : string)
+Export_Plot(dest : string, fname : string, steps : int, varname : string, title : string)
```

```
IntegralSystemOperator
-K : ivector
-SSys, SmidSys : ivector
+operator(Y : vector<itaylor> &, ord : int, sysord : int, IntA : imatrix &, IntB : imatrix &) : vector<itaylor>
```

Abbildung 5.6: Systemlöser: Klassen der Kernkomponente

  * `Einschliessung_berechnen`
  * `SystemLoesungsansatz`
  * `SystemLoesung_bilden`

- sowie Methoden zum Export von Maple-Sourcecode zur Verwendung der Funktionen und Ihrer Visualisierung in Maple mit Hilfe des Pakets `intpakX`.

Der vorliegende Löser ist daraufhin konzipiert, einzelne Integralgleichungen durch Zerlegung in Integralgleichungssysteme zu lösen. Zur Lösung von Integralgleichungssystemen mit einzeln definierten Kernen und rechten Seiten, die sich nicht auf eine einzelne Funktion zurückführen lassen, ist in der Klasse `IntegralGleichungsSystem` ein zusätzlicher Konstruktor zur Verfügung zu stellen.

Die Verfahren 3.1.7 und 3.1.8 werden analog zur Einzelversion als freie Funktionen implementiert.

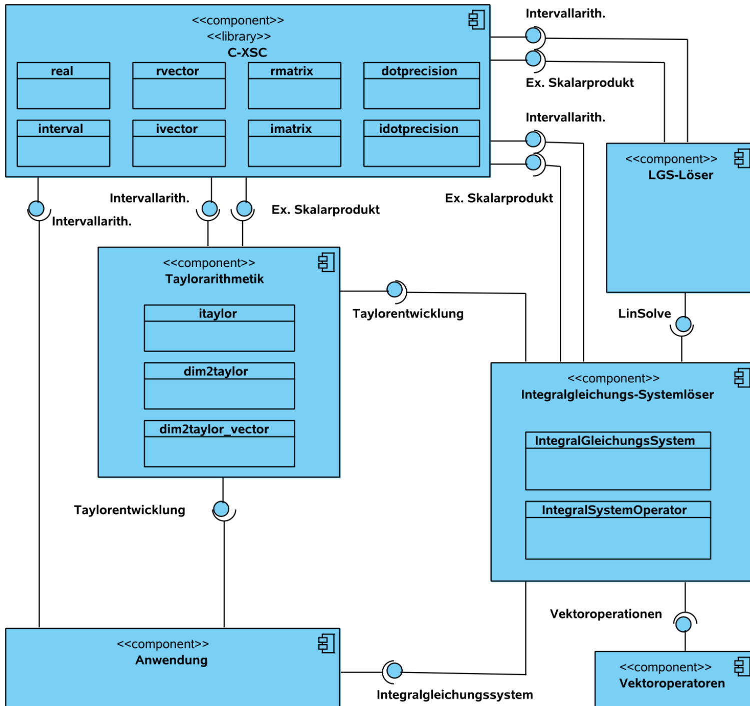

Abbildung 5.7: Systemlöser: Komponenten

Die oben beschriebenen Klassen des Lösers für Integralgleichungssysteme sind in Abbildung 5.6 dargestellt[12].

Zusätzlich zu den in der Einzelversion verwendeten Komponenten (Klassenbibliothek C-XSC zur Intervallrechnung, Taylorarithmetik gemäß Abschnitt 5.2 und linearer Gleichunssystemlöser aus [62]) werden in der Systemversion in umfangreicherem Maße als zuvor Vektoren der STL

[12] Konstruktoren, Zugriffs- und Hilfsfunktionen werden der Übersicht halber nicht dargestellt.

[79, 123] eingesetzt. Hierfür wurde ein Zusatzpaket zur Verfügung gestellt, das eine Reihe von Operatoren für die Klasse `vector` implementiert, die für diese Klasse nicht durch die STL bereitgestellt werden. Hierunter fallen:

- die Vektoraddition als arithmetischer Operator
- die relationalen Operatoren <, <=, >, >=
- Ausgabeoperatoren

Die oben angeführten arithmetischen und relationalen Operatoren sind in der Klasse `vector` aus konzeptionellen Gründen ursprünglich nicht vorgesehen, um die Containerklassen möglichst vielseitig verwendbar zu halten und insbesondere die Anwendung für Elementtypen zu gestatten, für die keine sinnvolle Definitionsmöglichkeit dieser Operatoren besteht.

Eine Übersicht der Komponenten des Lösers für Integralgleichungssysteme ist in Abbildung 5.7 zu finden[13].

Anwendung

Die Definition und Lösung einer Integralgleichung mit dem Systemverfahren durch das vorliegende Programm kann somit wie im folgenden Beispiel geschehen.

Beispiel 5.5.2 *In diesem Beispiel soll die in Beispiel 5.5.1 mit dem Einzelverfahren berechnete Lösung der Integralgleichung*

$$y(s) - \int_0^1 k(s,t)y(t)dt = g(s)$$

mit

$$k : [0,1] \times [0,1] \to \mathbb{R},\ (s,t) \mapsto \frac{1}{2}(s+1) \cdot \exp(-st)$$

und

$$g : [0,1] \to \mathbb{R},\ s \mapsto \exp(-s) - \frac{1}{2} + \frac{1}{2} \cdot \exp(-s-1)$$

[13] Die STL selbst wird dabei als integraler Bestandteil der Programmiersprache angesehen und nicht abgebildet.

mit Hilfe des Integralgleichungs-Systemlösers eingeschlossen werden.

Die Ordnung des Integralgleichungssystems soll 4 *betragen und es soll wieder eine Taylorordnung von* 6 *verwendet werden.*

Bei der Definition des Integralgleichungssystems werden die Definitionsbereiche unterteilt und man erhält

$$
\begin{aligned}
k_{11} &: [0, \tfrac{1}{4}] \times [0, \tfrac{1}{4}] \to \mathbb{IR},\ (s,t) \mapsto \tfrac{1}{2}(s+1) \cdot \exp(-st) \\
&\vdots \\
k_{44} &: [\tfrac{3}{4}, 1] \times [\tfrac{3}{4}, 1] \to \mathbb{IR},\ (s,t) \mapsto \tfrac{1}{2}(s+1) \cdot \exp(-st) \\
g_1 &: [0, \tfrac{1}{4}] \to \mathbb{IR},\ s \mapsto \exp(-s) - \tfrac{1}{2} + \tfrac{1}{2} \cdot \exp(-s-1) \\
&\vdots \\
g_4 &: [\tfrac{3}{4}, 1] \to \mathbb{IR},\ s \mapsto \exp(-s) - \tfrac{1}{2} + \tfrac{1}{2} \cdot \exp(-s-1)
\end{aligned}
$$

Der Programmcode des zugehörigen Programms ist dem der Anwendung des Einzelverfahrens sehr ähnlich:

```
#include "interval.hpp"
#include "itaylor.hpp"
#include "dim2taylor.hpp"
#include <fstream>

#include "System-FIGL-Loeser.hpp"

dim2taylor K(dim2taylor_vector& x)
{
  return (x[1]+1)*exp(-x[1]*x[2])/real(2.0);
}

itaylor RS(itaylor& s)
{
  return exp(-s)-real(0.5)+real(0.5)*exp(-s-1);
}
```

```
const interval KDefS=interval(0.0,1.0);
const interval KDefT=interval(0.0,1.0);
const interval RSDef=interval(0.0,1.0);

int main()
{
  // Taylor- und Systemordnung
  int ord=6;
  int sysord=4;

  // Ausgabedatei
  ofstream outf("004-6-4.out");

  //Integralgleichung definieren
  IntegralGleichungsSystem IGLSys(K, KDefS, KDefT,
                                  RS, RSDef,
                                  sysord,ord,outf);
  // Loesung (incl. Ausgabe)
  IGLSys.Einschliessung_berechnen(outf);

  return 0;
}
```

Beispielanwendung Integralgleichungs-Systemlöser

In diesem Beispiel wurde im Unterschied zu Beispiel 5.5.1 die Ausgabe in eine Datei vorgesehen[14].

Als Lösung erhält man in diesem Fall einen Lösungsvektor, dessen Funktionen wieder auf den Intervallen

$$[0, \tfrac{1}{4}], [\tfrac{1}{4}, \tfrac{1}{2}], [\tfrac{1}{2}, \tfrac{3}{4}], [\tfrac{3}{4}, 1]$$

definiert sind. Jedes Element wird dabei wieder durch seine Taylorkoeffizienten ausgedrückt, wobei die Taylorentwicklung sich jeweils auf den Mittelpunkt des Definitionsintervalls bezieht[15]. *Das Beispielergebnis lau-*

[14] Auch in der Einzelversion ist eine Dateiausgabe möglich statt der Verwendung des Standardausgabestroms.

[15] In der vorliegenden Taylorarithmetik wird der jeweilige Entwicklungspunkt nicht mitgespeichert.

tet wie folgt:

```
Endergebnis:

[0]: [itaylor object, order 6:]
[0][  0.882496,  0.882498]
[1][ -0.882497, -0.882496]
[2][  0.441248,  0.441249]
[3][ -0.147084, -0.147082]
[4][  0.036763,  0.036776]
[5][ -0.007432, -0.007380]
[6][  0.001050,  0.001463]

[1]: [itaylor object, order 6:]
[0][  0.687289,  0.687290]
[1][ -0.687290, -0.687289]
[2][  0.343644,  0.343645]
[3][ -0.114550, -0.114548]
[4][  0.028627,  0.028644]
[5][ -0.005787, -0.005736]
[6][  0.000813,  0.001142]

[2]: [itaylor object, order 6:]
[0][  0.535261,  0.535262]
[1][ -0.535262, -0.535261]
[2][  0.267630,  0.267631]
[3][ -0.089213, -0.089210]
[4][  0.022290,  0.022312]
[5][ -0.004507, -0.004456]
[6][  0.000629,  0.000892]

[3]: [itaylor object, order 6:]
[0][  0.416861,  0.416863]
[1][ -0.416863, -0.416861]
[2][  0.208430,  0.208432]
[3][ -0.069481, -0.069476]
[4][  0.017354,  0.017382]
[5][ -0.003513, -0.003461]
[6][  0.000486,  0.000697]
```

Beispielausgabe Integralgleichungs-Systemlöser

Auch die Ergebnisse des Systemlösers können als Funktionen oder Wertebereiche für die Visualisierung in Maple exportiert werden. Dabei werden die Einschließungen der Komponenten des Lösungsvektors in einer Grafik kombiniert (siehe Abschnitt 5.6).

5.5.4 Neue parallele Implementierung

Nach den seriellen Software-Versionen soll nunmehr die parallele Software vorgestellt werden. Für die parallele Implementierung wird ausschließlich das Systemverfahren verwandt, da es stärkere Parallelisierungsmöglichkeiten bietet.

Bei der Parallelisierung eines Programms wird zunächst der existierende Code für die parallele Ausführung in mehreren Prozessen technisch angepasst. Hierzu müssen etwa

- geeignete Bibliotheken oder Spracherweiterungen für die parallele Kommunikation eingebunden werden
- ein Kommunikationsraum für die beteiligten Prozesse bereitgestellt werden, der die Adressierung von Prozessen bei der Kommunikation erlaubt
- ein geeigneter Ein- und Ausgabemodus, meist auf der Basis von Ein- und Ausgabedateien, geschaffen werden, da die Ein- und Ausgabe via Bildschirm bei parallelen Prozessen nicht mehr oder nur noch eingeschränkt möglich ist

Wenn diese Voraussetzungen gegeben sind, kann die eigentliche Parallelisierung der in der Software implementierten Verfahren beginnen. Hierzu werden Verfahrensteile zur Parallelisierung identifiziert, und der Code für die entsprechenden Teile wird so modifiziert, dass jeder Prozess nur die für ihn vorgesehenen Teile ausführt. Die von den einzelnen Prozessen berechneten Ergebnisse werden dann je nach Notwendigkeit unmittelbar oder später denjenigen Prozessen, die die Ergebnisse für die weitere Verarbeitung benötigen, durch die zur Verfügung stehenden Kommunikationsmechanismen mitgeteilt. Dafür muss der Code der Berechnungsverfahren an den gewünschten Stellen modifiziert werden. Oft ist es dabei der Fall, dass die Struktur des Gesamtprogramms und seiner Komponenten sich dabei

nicht ändert. Lediglich der Code für einzelne Funktionen bzw. Methoden wird verändert.

Dies trifft auch bei dem hier vorgestellten Integralgleichungs-Systemlöser zu. Die Klassen des parallelen Lösers bleiben im Vergleich zum seriellen Systemlöser bis auf einige Details gleich und werden hier nicht noch einmal detailliert aufgeführt.

Die Parallelisierung der implementierten Verfahren ist in Abschnitt 4.2 beschrieben.

Für die parallele Implementierung muss geprüft werden, welche Komponenten zu parallelisierende Programmteile enthalten. Dies ist hier der Fall für den linearen Gleichungssystem-Löser und seine Teilkomponenten, der gemäß Abschnitt 5.4 parallelisiert wurde.

Desweiteren werden zusätzliche Komponenten zur parallelen Kommunikation benötigt:

- MPI als technische Grundlage zur parallelen Kommunikation
- das Kommunikationspaket für C-XSC-Datentypen, Taylor-Datentypen und Vektoren aus Abschnitt 5.3

In Abbildung 5.8 ist der parallele Systemlöser mit allen Komponenten dargestellt.

Anwendung

Bei der Erstellung einer Anwendung, die den parallelen Systemlöser verwendet, müssen im Vergleich zum seriellen Löser folgende zusätzliche Schritte durchgeführt werden:

- Initialisierung der parallelen Umgebung
- Initialisierung der MPI-Kommunikation für C-XSC-Datentypen
- Einrichtung von Ein- und Ausgabedateien
- Nach Ausführung der parallelen Berechnungen parallele Umgebung beenden

Im Folgenden ist der Code eines Beispielprogrammes in gekürzter Fassung angegeben.

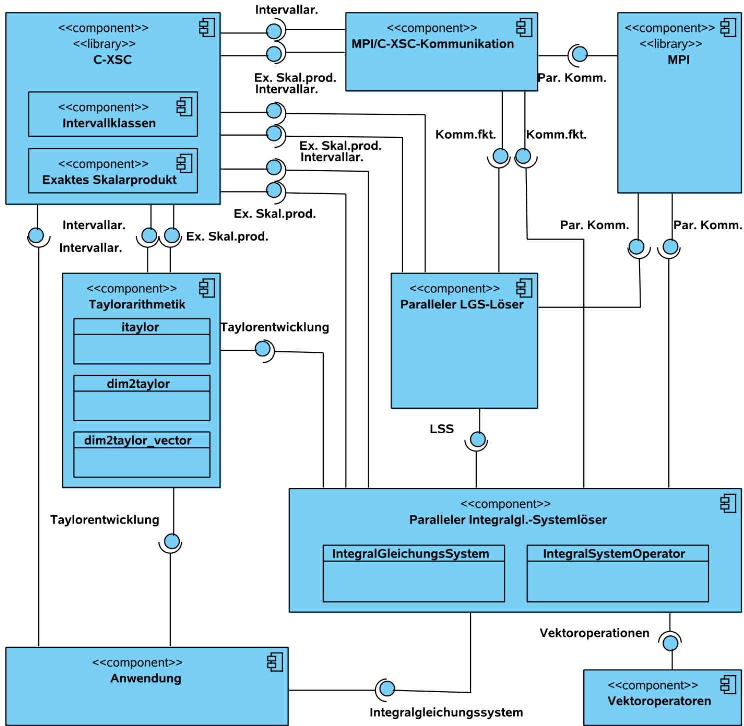

Abbildung 5.8: Paralleler Systemlöser: Komponenten

```
#include <mpi.h>
//[weitere...]

#include "System-FIGL-Loeser-par.hpp"

// Definition der Funktionen

// [ausgelassen]
```

```
int main(int argc, char *argv[])
{
  // MPI-Kommunikation initialisieren
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  // MPI-C-XSC-Kommunikation initialisieren
  MPI_Define_CXSC_Types();

  // Ausgabedateien spezifizieren und oeffnen
  // [Code ausgelassen]

  // Anwendung parallel ausfuehren
  int root=0;
  if (mypid==root)
  {
    IGLSys=IntegralGleichungsSystem(K,KS,KT,ReS,GS,
                     sysord,ord,ausg,ausgabedatei);
    IGLSys.DistributeValues(root);
  }
  else
  {
    IGLSys.PartInitIntegralGleichungsSystem(K,KS,KT,
              ReS,GS,sysord,ord,ausg,ausgabedatei);
    IGLSys.DistributeValues(root);
  }
  IGLSys.Einschliessung_berechnen(ausg,ausgabedatei,
                                  procs,mypid,err);
  // Export nach Maple
  if (mypid==root)
  {
    IGLSys.Export_Plot("Maple",plotdatei,8);
  }

  // Beenden
  MPI_Finalize();
  return 0;
}
```

Beispielanwendung Integralgleichungs-Systemlöser

In der Klasse `IntegralGleichungsSystem` wurde eine zusätzliche Methode zur partiellen Initialisierung einer Integralgleichung eingeführt, die einige Schritte auslässt und z.B. die Taylorentwicklung der Funktionen nicht durchführt. Diese Schritte können dann von Prozess 0 durchgeführt werden; die von den restlichen Prozessen benötigten Daten werden anschließend an diese verschickt. In der praktischen Anwendung konnte hierdurch eine höhere Programmstabilität bei nur wenig zusätzlichem Kommunikationsaufwand in Fällen erreicht werden, in denen die benötigten Ressourcen hardwareseitig den kritischen Bereich erreichten, d.h. in denen der zur Verfügung stehende Speicher stark ausgeschöpft wurde.

Im obigen Beispielcode ist auch der Export der Ergebnisdaten zur Visualisierung in Maple mit dem in Abschnitt 5.6 vorgestellten Interface enthalten.

Zur Übersetzung und Ausführung in der jeweils verwendeten parallelen Umgebung sind weitere Schritte notwendig. Im Anhang findet sich ein Beispiel für die praktische Benutzung der hier verwendeten parallelen Umgebung *ALiCEnext* [4].

5.6 Ergebnisexport und Visualisierung in Maple mit intpakX

Das Ergebnis einer Berechnung, in diesem Fall die Einschließung der Lösung einer Integralgleichung, kann auf verschiedene Weise weiterverarbeitet werden. Bei der Verwendung der Ergebnisse kann es von Interesse sein,

- das Ergebnis als Taylor-Objekt in weiteren Berechnungen mit weiteren Funktionen in Taylor-Darstellung einzubringen
- das Ergebnis als Funktion auf dem Bildschirm auszugeben
- das Ergebnis als Funktionsausdruck in anderen Sprachumgebungen zu verwenden
- das Ergebnis als Graph bzw. Einschließung eines Wertebereichs grafisch zu visualisieren.

Während die erste Verwendung mit dem Ergebnis, das als Objekt des Taylor-Datentyps `itaylor` vorliegt, sofort möglich ist, ist eine sinnvolle Darstellung auf dem Bildschirm und vor allem eine Verwendung in externen Systemen und eine Visualisierung nur durch die Einbringung zusätzlicher Funktionalität erreichbar.

Hierzu wurde in dem vorgestellten Integralgleichungslöser ein Interface zum Export der Ergebnisse nach Maple geschaffen. Hierbei wurde das Computer-Algebra-System Maple [89] als Beispielumgebung für den Export gewählt, die Erweiterung auf den Export in weitere Umgebungen ist möglich.

Maple wurde aus folgenden Gründen als Umgebung für den Export der berechneten Lösungen und ihre Visualisierung gewählt:

- Maple als Computer-Algebra-Software bietet eine Umgebung, in der Funktionen direkt in mathematischer Schreibweise definiert, ausgegeben und verarbeitet werden können. Dabei sind sowohl numerische als auch symbolische Berechnungen möglich. Zudem ermöglicht Maple die grafische Darstellung von Mengen und Funktionen und damit auch die Entwicklung von Visualisierungsmethoden für Intervallverfahren (s.u.).

- Durch das Maple-Paket `intpakX` [35, 44, 78] wird in Maple eine Intervallarithmetik zur Verfügung gestellt, die einen Intervall-Datentyp sowie Intervall-Operationen und -Funktionen bietet. Hierdurch wird erst die Voraussetzung geschaffen, dass Ergebnisse eines Intervall-Verfahrens erfolgreich in dieser Umgebung verwendet werden können. Durch das Paket `intpakX`, das auch als *Maple Research PowerTool* [43] verfügbar ist, werden ebenso Funktionen zur Visualisierung von Intervallen, Einschließungen der Wertebereiche von Intervall-Funktionen und Ergebnissen verschiedener Intervall-Verfahren (z.B. Intervall-Newton-Verfahren) zur Verfügung gestellt.

- Darüberhinaus verfügt Maple über eine *Multiple Precision*-Arithmetik mit beliebig einstellbarer Länge, so dass auch Ergebnisse von weiteren Implementierungen in C-XSC, die Langzahldatentypen oder das exakte Skalarprodukt nutzen, verarbeitet werden können. Untersuchungen hierzu und zu weiteren Umgebungen mit *Multiple Precision*-Arithmetik finden sich in [45].

Die in Abschnitt 5.5.2, 5.5.3 und 5.5.4 vorgestellten Implementierungen enthalten zwei Exportfunktionen als Schnittstelle zu Maple. Sie stellen zwei verschiedene Möglichkeiten zur Verfügung, die Ergebnisse des Integralgleichungslösers in Maple weiterzuverwenden.

Weiterverwendung der Ergebnisse als Funktionsausdrücke

Die erste Exportfunktion stellt Maple-Code zur Verfügung, der eine Funktionsdefinition für das Ergebnis des Integralgleichungslösers, das als Taylor-Objekt vorliegt, enthält.

Das Interface der Funktion, die Teil der Klasse `IntegralGleichung` (in der Einzelversion) bzw. der Klasse `IntegralGleichungsSystem` (in der Systemversion) ist, lautet wie folgt:

```
void Export(string dest, string fname,
            string varname="IGL_Lsg");
```

Export-Interface (funktionale Ausgabe)

Dabei kann die Zielumgebung durch einen `string`-Parameter spezifiziert werden. Hier ist bisher die Verwendung des Strings `"Maple"` möglich, es können jedoch weitere Möglichkeiten hinzugefügt werden.

Beispiel 5.6.1 *Das mit dem Integralgleichungslöser gewonnene Ergebnis*

```
[itaylor object, order 6:]
[0][ 0.605778603790071 86, 0.607283918909978 13]
[1][-0.606789748952110 36,-0.606257862779323 47]
[2][ 0.302741538185436 30, 0.303863775655711 88]
[3][-0.105598061171873 99,-0.096871214040987 59]
[4][ 0.010986649243195 65, 0.038784675552263 11]
[5][-0.018967486096208 67, 0.008101075451702 47]
[6][-0.003360594186872 50, 0.006020602307434 78]
```

Export-Interface: Ausgangsobjekt

wird wie folgt als Maple-Code ausgegeben, der unter Verwendung des Pakets `intpakX` *als Definition einer Intervall-Funktion verwendet werden kann.*

```
IGL_Lsg := x ->
[0.60577860B3790071867, 0.607283918909978127]
&+([-0.606789748952103 53,-0.606257862779323475]
&*((x &- [0.500000000000000000, 0.500000000000000000])
  &intpower 1 ) )
&+([0.302741538185436309, 0.303863775655711877]
&*((x &- [0.500000000000000000, 0.500000000000000000])
  &intpower 2 ) )
&+([-0.105598061171873981,-0.096871214040987599]
&*((x &- [0.500000000000000000, 0.500000000000000000])
  &intpower 3 ) )
&+([0.010986649243195653, 0.038784675552263101]
&*((x &- [0.500000000000000000, 0.500000000000000000])
  &intpower 4 ) )
&+([-0.018967486096208662, 0.008101075451702467]
&*((x &- [0.500000000000000000, 0.500000000000000000])
  &intpower 5 ) )
&+([-0.003360594186872493, 0.006020602307434780]
&*((x &- [0.500000000000000000, 0.500000000000000000])
  &intpower 6 ) ) ;
```

Export-Interface: Funktion in Maple-Code

Visualisierung der Ergebnisse

Die zweite Exportfunktion stellt zwei Vektoren als Maple-Code zur Verfügung, die zusammen Paare von Definitionsbereichen und Wertebereichseinschließungen der Ergebnisfunktionen definieren. Die Anzahl der Teilintervalle, in die der Definitionsbereich zerlegt wird, kann dabei frei gewählt werden. Bei der Systemversion werden zwei Gesamtvektoren für den Gesamtdefinitions- bzw. Wertebereich erstellt, wobei jede Lösungsfunktion jeweils bezüglich ihres Teil-Definitionsbereichs unterteilt und ausgewertet wird.

Die Wertebereichseinschließung erfolgt mit der C-XSC-Arithmetik, wenn eine Maple-Auswertung gewünscht wird, kann das vorstehend beschriebene Interface gewählt werden.

Die entstehenden Vektoren werden unter Verwendung des Pakets `intpakX` mit der eigens hierfür vorgesehenen Funktion

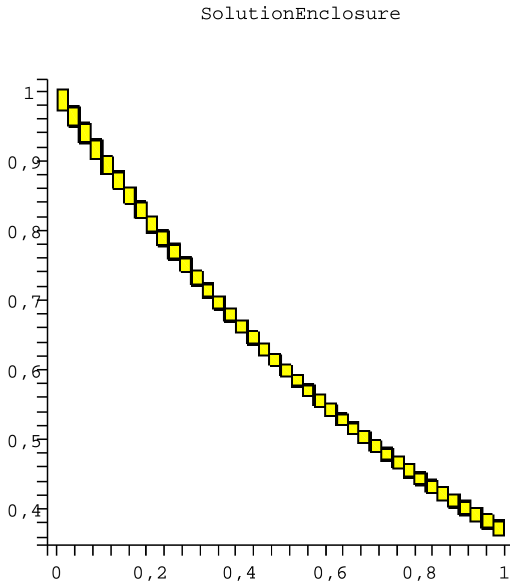

Abbildung 5.9: Lösungsexport: Maple-Grafik

`interval_list_plot` in eine Grafik umgewandelt und mit der Maple-Funktion `display` angezeigt. Auch hierfür wird von der Export-Funktion automatisch Maple-Code erzeugt. Für zwei Vektoren `IGL_Lsg_Args` und `IGL_Lsg_Vals` (deren Definition in Maple-Code hier nicht wiedergegeben wird) lautet der Code etwa wie folgt:

```
IGL_Lsg_Plot := interval_list_plot( IGL_Lsg_Args,
          IGL_Lsg_Vals, color=yellow, style=patch):
display(IGL_Lsg_Plot, axes=framed,
          title="Solution␣Enclosure");
```

Export-Interface: Maple-Code zur Erstellung der Grafik

Beispiel 5.6.2 *Für die Ergebnisfunktion aus Beispiel 5.6.1 wird bei der*

Wahl von 40 Teilintervallen der Maple-Code für Grafik 5.9 generiert.

5.7 Integralgleichungslöser nach Obermaier

Dieser Abschnitt stellt die Implementierung des Verfahrens von Obermaier [106] dar. Die Implementierung dient im Rahmen dieser Arbeit zur Berechnung von Vergleichsergebnissen zu den Ergebnissen der Implementierungen der Verfahren von Klein, die bereits beschrieben wurden.

Zuerst wird die existierende Implementierung aus [106] in Kurzform erläutert.

Zur Verwendung der Implementierung in der vorliegenden parallelen Umgebung *ALiCEnext* [4] mussten einige Modifikationen an der Software vorgenommen werden, die in Abschnitt 5.7.2 diskutiert werden.

5.7.1 Existierende Implementierung

Die von Obermaier in [106] erstellte Implementierung enthält sowohl das serielle Verfahren (Abschnitt 3.3) als auch das parallele Verfahren (Abschnitt 4.4). Sie stellt damit die einzige bekannte Software zur parallelen verifizierten Lösung Fredholmscher Integralgleichungssysteme zweiter Art dar, die zum Vergleich verfügbar war.

Der Code gliedert sich gemäß der Einteilung des Verfahrens (siehe Verfahrensbeschreibung in Abschnitt 3.3) in einzelne Übersetzungseinheiten:

- `omega`: Berechnung einer Näherungslösung ω der Integralgleichung
- `defekt`: Berechnung einer Konstante δ zur Abschätzung des Defekts der Näherungslösung
- `const_k`: Abschätzung der Norm der Inversen des Operators $I-\mathcal{K}$ nach oben durch eine Konstante K
- `func_g`: Bestimmung einer monoton wachsenden Funktion G

- `alpha`: Bestimmung einer Konstanten $\alpha \geq 0$ für die Gesamtabschätzung

Die ersten vier Einheiten sind für das serielle als auch für das parallele Verfahren separat implementiert, die Bestimmung der Konstanten α unterscheidet sich nicht zwischen serieller und paralleler Version. Einige Hilfsstrukturen und Funktionen sind als separate Übersetzungseinheiten vorhanden.

Für das parallele Verfahren existiert zusätzlich eine Übersetzungseinheit

- `loadbalance`: Verwaltung der parallel berechneten Teilprobleme.

Die Implementierung der obigen Einheiten erfolgte in C++, wobei aufgrund des funktionalen Charakters der Anwendung auch hier nur wenige Klassen für interne Strukturen definiert werden. Der Gesamtablauf wird in Funktionen strukturiert.

Abweichende Aussagen ergeben sich für die Implementierung der Steigungsarithmetik zur Darstellung von Funktionen (siehe Abschnitt 2.4), die als Klasse in einem eigenen Modul implementiert ist, sowie für die im Folgenden beschriebene Benutzeroberfläche, die eine Reihe von Klassen enthält.

Eine Besonderheit ergibt sich bei der Struktur der Gesamtsoftware und ihrer Komponenten. Die einzelnen Module wurden jeweils so konzipiert, dass sie jeweils vollständige C++-Programme darstellen, d.h. jeder Teilschritt der Verfahren ist als eigenständiges Programm mit integriertem Hauptprogramm implementiert. Die Gesamtanwendung wird durch ein Steuerungsmodul mit grafischer Benutzeroberfläche zusammengeführt. Der Programmablauf sieht in der vorliegenden Version die interaktive Durchführung des Programms vor, die folgende Schritte beinhaltet:

- Start der Benutzeroberfläche
- Bestimmung der Eingabedaten durch Eingabe in eine Eingabemaske
- Start der einzelnen, unabhängigen Teilprogramme durch Aktivieren per Button und Auswahl der Programmparameter (insbesondere

Wahl der seriellen oder parallelen Version und der Anzahl der Prozessoren für die parallele Version) in Auswahlfeldern.

Die Eingabedaten werden bei diesem Vorgehen in Dateien codiert, danach wird automatisch ein Compilierungsvorgang für die Teilprogramme durchgeführt. Die Programme übergeben die benötigten Daten an die nachfolgenden Programmteile durch Speicherung in Dateien.

Das obige Vorgehen schließt ein, dass auch die parallelisierten Programmteile als einzelne Schritte, die separat gestartet werden müssen, ausgeführt werden.

Das Steuerungsmodul ist im Gegensatz zu den restlichen Programmteilen in Java implementiert.

In der Kernanwendung, deren Teile durch die grafische Oberfläche gesteuert werden, werden zusätzlich folgende Komponenten verwendet:

- die Bibliothek `filib++` (siehe Abschnitt 5.1.2), die die Intervallarithmetik zur Verfügung stellt
- ein Modul, das die Steigungsarithmetik implementiert. Die Steigungsarithmetik wird in einer Klasse `mslope` zur Verfügung gestellt, die die arithmetischen Grundoperationen sowie die meisten Standardfunktionen implementiert. Einige Funktionen, die in der C-XSC-Taylorarithmetik (siehe Abschnitt 5.2) implementiert sind, sind jedoch nicht enthalten, insbesondere fehlen die Gaußsche Fehlerfunktion und die komplementäre Fehlerfunktion.

Einige Teilverfahren wie ein linearer Gleichungssystemlöser sind fest in den Modulen, die diese benötigen, integriert.

Die Komponenten der Software sind in Abbildung 5.10 zusammenfassend dargestellt.

5.7.2 Portierung

Um die Software von Obermaier in der parallelen Umgebung ALiCEnext [4] zu verwenden, mussten einige Anpassungen vorgenommen und zusätz-

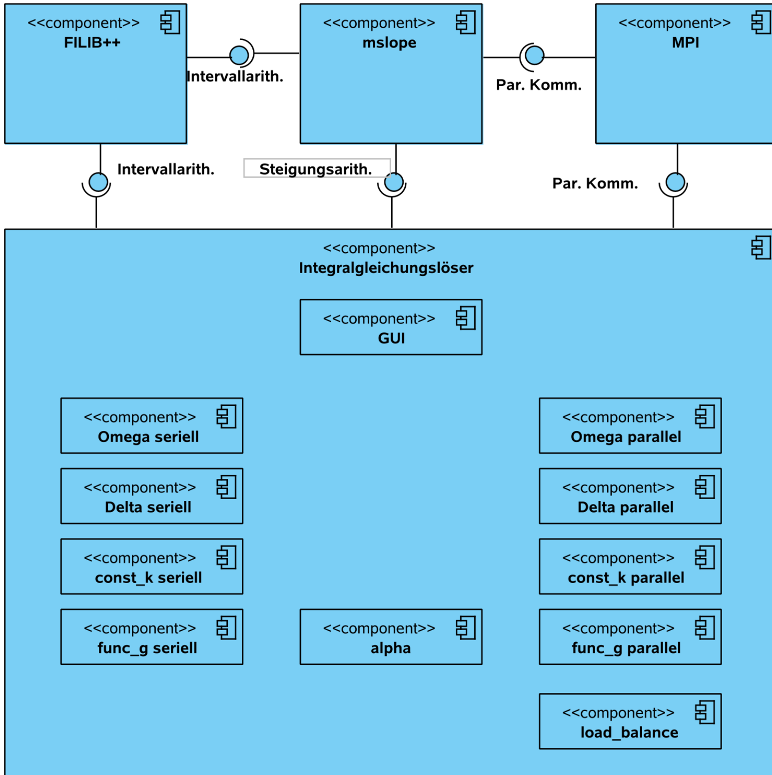

Abbildung 5.10: Implementierung von Obermaier: Komponenten

liche Aktivitäten durchgeführt werden. Die Gründe hierfür liegen in der speziellen Struktur der Software und werden im Folgenden erläutert.

Parallele Ausführung

Die Struktur der Software als Einzelbausteine, die durch eine Benutzeroberfläche steuerbar sind, schließt ein, dass die zur Ausführung der beteiligten Einzelprogramme notwendigen Anweisungen im Programm codiert

sind. In parallelen Umgebungen hängt aber insbesondere die Aufrufsyntax für den Start des parallelen Programms stark von der zur Verfügung stehenden Umgebung ab. In der verwendeten Umgebung *ALiCEnext* etwa unterscheiden sich die Struktur der Dateisysteme auf der Frontend-Hardware (auf der die Programmaufrufe erfolgen) und den ausführenden Knoten, so dass die aktive Spezifikation von Verzeichnissen notwendig wird. Dies ist für die Software von Obermaier nur durch Erstellung zusätzlicher Skript-Dateien möglich, die die zusätzlich benötigten Parameter enthalten und statt des direkten Programmaufrufs ausgeführt werden. Hier musste im Steuerungsmodul die entsprechende Funktionalität ersetzt werden.

Verwendung im Batchbetrieb

Ebenso gilt für viele parallele Umgebungen, dass sie keine direkte Ausführung von parallelen Programmen erlauben. Die zur Verfügung stehenden Prozessoren und die zur Verfügung stehende Rechenzeit werden von *Batchsystemen* verwaltet, die Warteschlangen für die zu verarbeitenden Programme verwalten und Programme je nach Verfügbarkeit von Ressourcen starten. In der parallelen Umgebung *ALiCEnext* ist hierfür das System *Torque* [126] installiert, eine Version des *Portable Batch System* [104].

Eine interaktive Verarbeitung mit einer Benutzeroberfläche ist bei Verwendung eines Batchsystems nicht möglich. Um die Software von Obermaier zusammen mit einem Batchsystem zu verwenden, das das gesamte Verfahren in einem Stück im Batchbetrieb ausführt, muss die Software Software weitgehend umstrukturiert werden, um die Verwendung der Benutzeroberfläche zu umgehen. Im Rahmen dieser Arbeit wurde die Software auf einer begrenzten Anzahl von maximal 16 Prozessoren, auf denen eine Direktausführung der Software und der Benutzeroberfläche möglich war, getestet.

Neue Bibliotheksversionen und neue Compiler

Eine weitere Aufgabe bei der Modifikation der Software bestand in der Anpassung der verwendeten Intervallbibliothek `filib++`. Bei der verwendeten parallelen Umgebung handelt es sich um eine 64-Bit-Architektur, so dass zur Verfügung stehende 32-Bit-Bibliotheksversionen nicht mit dem 64-Bit-Code der restlichen Software verwendbar waren. Es konnte jedoch durch Anwendung geeigneter Compileroptionen erreicht werden,

dass die gesamte Software als 32-Bit-Code übersetzt werden konnte (ohne die Eigenschaften der 64-Bit-Architektur ausnutzen zu können).

Gleichzeitig bestand eine Aufgabe darin, den Code der Software so anzupassen, dass statt der Makro-Version der Bibliothek `filib++` die Template-Version benutzt werden konnte, da nur diese mit dem in der parallelen Umgebung zur Verfügung stehenden Compiler (eine Version des GNU g++ [37]) übersetzbar war. Die hierbei auftretenden Unterschiede zwischen den verschiedenen Versionen der Bibliothek `filib++` sind in Abschnitt 5.1.2 erläutert.

Nach Durchführung der oben beschriebenen Modifikationen konnte die Software in der parallelen Umgebung *ALiCEnext* auf einer Anzahl von 16 Prozessoren, auf denen der Direktstart der Benutzeroberfläche möglich war, ausgeführt werden.

5.8 Weitere Anwendungen des MPI-Kommunikationspakets für C-XSC-Datentypen

Da das MPI-Kommunikationspaket für C-XSC-Datentypen, das in Abschnitt 5.3 vorgestellt wird, als allgemeines Interface zur Anwendung von C-XSC in parallelen Umgebungen konzipiert ist, sollen auch andere Anwendungen als der hier vorgestellte parallele Integralgleichungslöser diese Erweiterung nutzen können. Als Beispiel hierfür wird nachfolgend die globale Optimierungssoftware von Wiethoff betrachtet, die in [131] beschrieben wird.

Das zu Grunde liegende Verfahren der globalen Optimierung wird hier nicht näher dargestellt. Verfahren der globalen Optimierung sind grundsätzlich gekennzeichnet durch die Suche nach dem globalen Minimum oder Maximum einer Funktion. Zahlreiche Verfahren verfolgen dieses Ziel, unter ihnen auch verifizierte Verfahren. Eine häufig angewandte Strategie in solchen Verfahren ist die Unterteilung des Gesamtproblems in Teilprobleme, wodurch eine Struktur von zu bearbeitenden Teilproblemen entsteht, die parallel bearbeitet werden können.

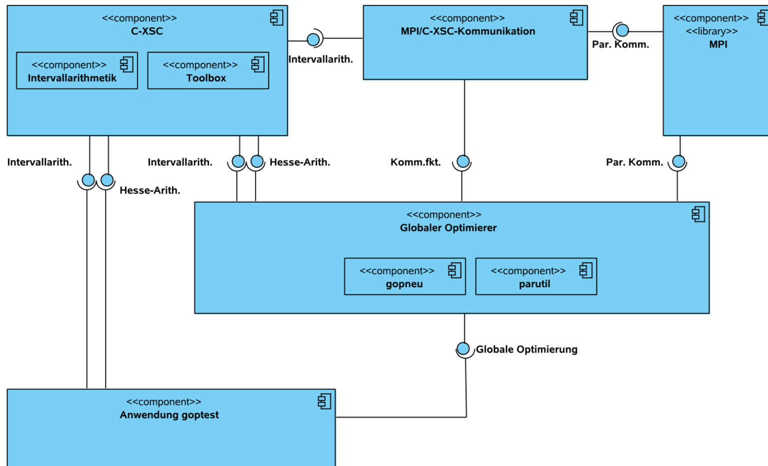

Abbildung 5.11: Globaler Optimierer von Wiethoff: Komponenten

Neben Wiethoff [131] beschäftigen sich z.B. auch Ibraev [57] und Tapamo [124] mit Fragestellungen der globalen Optimierung.

Die grundlegenden Teile der Software sind:

- das Modul `gopneu`, das das Verfahren zur globalen Optimierung implementiert
- das Modul `parutil`, das die Verwaltung der parallel zu verarbeitenden Teilaufgaben enthält
- eine Anwendung `goptest`, die die Auswahl einer Anzahl von Test-Funktionen erlaubt und das Optimierungs-Modul aufruft.

Die Software benutzt folgende externe Komponenten:

- C-XSC (inclusive einiger Module aus der C-XSC-Toolbox [53])
- MPI

Die Komponenten sind in Abbildung 5.11 dargestellt.

Im Rahmen der hier angestellten Betrachtungen ist nur die Verwendung des MPI-Kommunikationspakets für C-XSC-Datentypen in der Software von Wiethoff von Interesse.

Die Software stand nur in einer unvollständigen und nicht übersetzbaren Version zur Verfügung, in der insbesondere die Funktionalität zur MPI-Kommunikation mit C-XSC-Datentypen, die in [131] bereits beschrieben wird, nicht vorhanden war. Aufgabe war es daher, die Software erneut mit dieser Funktionalität auszustatten.

Es wurden daher die nicht funktionsfähigen Kommunikationsanweisungen im Programm durch Aufrufe der Kommunikationsfunktionen aus dem neuen MPI-Kommunikationspaket für C-XSC-Datentypen ersetzt.

Neben der Kommunikationsfunktionalität mussten weitere Details an der vorliegenden Implementierung modifiziert werden, um eine übersetzbare und funktionsfähige Software zu erhalten:

- Korrekte Verwendung konstanter Referenzen
- Austausch der Datentypen einiger Matrizen, bei denen der Zugriff auf Teilmatrizen nicht korrekt funktionierte. Dies hatte zur Folge, dass zusätzliche Kommunikationsroutinen für die Kommunikation von STL-Vektoren zur Verfügung gestellt werden mussten und somit auch die Kommunikationsfunktionen aus dem in Abschnitt 5.3.4 beschriebenen Teilpaket zur Anwendung kommen. Auch hier wurden die entsprechenden Aufrufe der Kommunikationsfunktionen eingefügt bzw. ersetzt.

Es steht nunmehr eine Version der Software zur Verfügung, die in der parallelen Umgebung *ALiCEnext* übersetzbar und ausführbar ist.

6 Test und Ergebnisse

In diesem Kapitel werden die Testergebnisse vorgestellt und diskutiert, die mit der in Kapitel 5 präsentierten Software gewonnen wurden.

Es wurden Tests im Hinblick auf Rechenzeit, Einschließungsgenauigkeit und parallele Performance mit den neuen Softwarekomponenten durchgeführt. Die einzelnen Komponenten sind dabei von unterschiedlicher Bedeutung für die Ergebnisse und wurden in unterschiedlichem Umfang betrachtet:

- Die erweiterte C-XSC-Taylorarithmetik bildet die Grundlage für die Tests der einzelnen Anwendungen und wurde nicht separat getestet.
- Das MPI-Kommunikationspaket für C-XSC-Datentypen wurde in einigen Aspekten bereits in [46] getestet. Die hier durchgeführten Tests zielen insbesondere auf die Effizienz der Routinen im Vergleich zur manuellen Kommunikation ab.
- Der parallele verifizierte Gleichungssystem-Löser wird im Rahmen der Tests des Integralgleichungslösers betrachtet, u.a. wird dabei der Anteil des Gleichungssystem-Lösers an der benötigten Rechenzeit untersucht.
- Den Schwerpunkt der Tests bilden die Tests des parallelen verifizierten Integralgleichungslösers. Neben der Genauigkeit der Lösungseinschließungen wurde insbesondere das Verhalten der Rechenzeit in Abhängigkeit verschiedener Parameter getestet.
- Die Software nach dem Verfahren von Obermaier wurde zur Bestimmung von Vergleichsergebnissen getestet, um die Ergebnisse des parallelen verifizierten Integralgleichungslösers einzuordnen und Vor- und Nachteile der Verfahren und der verwendeten Komponenten zu bestimmen.

Die jeweiligen Testkriterien und -parameter werden in den nachfolgenden Abschnitten genauer erläutert.

Zunächst wird jedoch die Hard- und Softwareumgebung beschrieben, in der die Tests durchgeführt wurden.

6.1 Hard- und Softwareumgebung

Die Testumgebung für die Tests mit der neuen Software besteht zum einen aus der Hardware und zum anderen aus den in dieser Hardwareumgebung benutzten Softwarewerkzeugen zum Übersetzen und Ausführen der jeweiligen Anwendungen.

Als Hardwareumgebung für die parallele Software stand *ALiCEnext* [4], der Hochleistungsrechner der Bergischen Universität Wuppertal, zur Verfügung.

ALiCEnext ist seit 2004 an der Bergischen Universität Wuppertal installiert und hat die folgenden Merkmale:

- 1024 1.8 GHz AMD Opteron 64-Bit-Prozessoren mit je 1024 MB Hauptspeicher auf 512 Knoten
- Performance 2083 GFlops (gemessen mit dem *LINPACK Benchmark* [26, 27])

Zum Zeitpunkt der Inbetriebnahme im Juni 2004 nahm *ALiCEnext* Platz 74 in der *Top500*-Liste ein [125].

Als Betriebssystem dienten unterschiedliche[1] Versionen und Distributionen des Betriebssystems Linux. Ende 2006 war das System *CentOS 3.5* [16] installiert.

Folgende weitere Software stand zur Übersetzung und Ausführung der Anwendungen zur Verfügung:

- Compiler g++(gcc) 3.3.1 [37]

[1] Nicht auf allen Knoten war stets dieselbe Betriebssystemversion installiert; darüberhinaus wurden die Systeme in unregelmäßigen Abständen aktualsiert.

- Batchsystem Torque v2.0.0p2 [126]
- MPI-Implementierung MPICH 1.2.7p1 [99]
- C-XSC 2.1.1 [22]

Abweichende Umgebungen bei einzelnen seriellen Tests werden im Rahmen der jeweiligen Tests angegeben.

6.2 Parallele Kommunikation

Zunächst werden Ergebnisse zweier Tests der parallelen Kommunikation unabhängig von den betrachteten konkreten Anwendungen angegeben.

6.2.1 Parallele Kommunikation mit C-XSC-Datentypen

Das MPI-Kommunikationspaket für C-XSC-Datentypen aus Abschnitt 5.3 ermöglicht die Benutzung von MPI-Kommunikationsroutinen mit Parametern, die C-XSC-Datentypen besitzen. Die Effizienz der Routinen wurde wie folgt getestet.

Es wurden jeweils unterschiedlich große Datenobjekte (Vektoren bzw. Matrizen unterschiedlicher Dimension und unterschiedlicher Elementtypen) zyklisch zwischen einer festen Anzahl von Prozessoren verschickt. Dabei wurden drei Szenarien der Kommunikation verglichen:

EV Einzelversand jedes Vektor- bzw. Matrixelementes mit den MPI-Standardroutinen

KP Versand des Objektes mit den Routinen aus dem MPI-Kommunikationspaket für C-XSC-Datentypen

DT Versand mit Hilfe eines jeweils von Hand individuell definierten MPI-Datentyps für Datenobjekte dieses Grundtyps (kein Langzahldatentyp) und dieser (festen) Dimension.

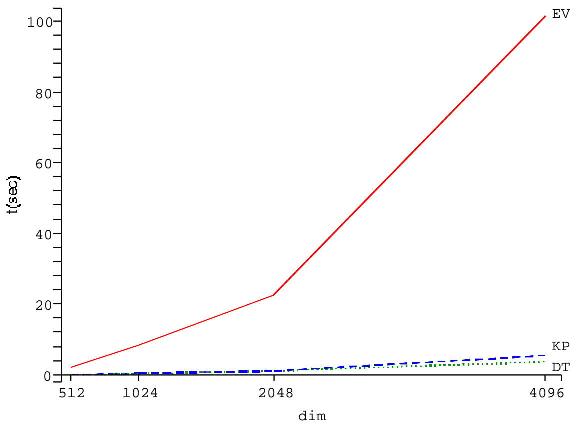

Abbildung 6.1: Kommunikationszeiten für C-XSC-Objekte

Für eine Anzahl von 32 Prozessen bzw. Prozessoren bei der Verwendung des Grunddatentyps `interval` wurden die folgenden Kommunikationszeiten für quadratische Matrizen der angegebenen Dimensionen bestimmt.

Dim.	512	1024	2048	4096
EV	0.97-2.09	4.11-8.35	16.99-22.57	67.84-101.58
KP	0.09-0.11	0.23-0.36	0.80-1.07	4.82-5.42
DT	0.07-0.10	0.25-0.31	0.96-1.10	3.17-3.67

(Zeit in sec. bei 32 Prozessoren)

Die maximal benötigten Komunikationszeiten sind in Abbildung 6.1 dargestellt.

Während die Zeiten mit dem Kommunikationspaket und den von Hand definierten Typen sich nur geringfügig unterscheiden, übertreffen die Zeiten für den Einzelversand der Elemente diese bei weitem. Solange für die versendeten Daten keine kritische Menge an Speicherplatz verwendet wird, so dass bei Kopiervorgängen der zur Verfügung stehende Hauptspeicher überschritten wird, ist damit das neue Kommunikationspaket in

etwa gleich effizient wie die Vorgehensweise mit individueller Definition benutzerdefinierter Datentypen.

6.2.2 Netzwerkstruktur

In der Testumgebung *ALiCEnext* wurden zwei unterschiedliche Netzwerkstrukturen verwendet. Zum einen waren die Knoten durch eine Baumstruktur verbunden, zusätzlich war ein Crossbar geschaltet - dieses wurde jedoch zu Wartungs- und Testzwecken zu einigen Zeiten abgeschaltet. Daher wurde getestet, ob die unterschiedliche Netzwerkstruktur die Kommunikationszeiten in den ausgeführten Anwendungen entscheidend verändert.

Hierzu wurde der parallele Integralgleichungslöser mit verschiedenen Parametergrößen sowohl mit verfügbarem Crossbar als auch mit abgeschaltetem Crossbar getestet. Es wurden jeweils 32 zufällig durch das Batchsystem zugewiesene Prozessoren verwendet[2].

Die folgenden Testwerte zeigen Rechenzeiten des Verfahrens für verschiedene Problemparameter für jeweils identische Funktionen, die in der Dimension der Matrix des im Laufe des Verfahrens resultierenden linearen Gleichungssystems zusammengefasst sind.

Dim.	320	640	832	1152	2048
mit Crossbar	10.0	65.1	218	461	3462
ohne Crossbar	10.0	65.9	220	465	3470

(Zeit in sec. bei 32 Prozessoren)

Die getesteten Parameterwerte stellten den Bereich dar, in dem auch die weiteren Tests mit dem Integralgleichungslöser ausgeführt wurden.

Die Werte sind in Abbildung 6.2 dargestellt. Die Zeiten in den durchgeführten Tests unterschieden sich nur unwesentlich, so dass der Einfluss

[2] Die zufällige Zuteilung von Prozessoren zu Rechenaufträgen stellt dabei das Standardvorgehen bei der Zuteilung von Ressourcen durch das Batchsystem in der vorliegenden Konfiguration dar.

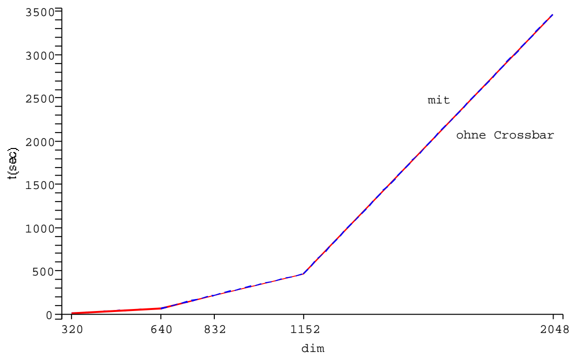

Abbildung 6.2: Netzwerkstruktur und Rechenzeit

der unterschiedlichen Netzwerkstruktur auf die Ergebnisse der nachfolgenden Tests des Integralgleichungslösers vernachlässigt werden kann.

6.3 Paralleler verifizierter Integralgleichungslöser

Mit dem parallelen Integralgleichungslöser wurden zahlreiche Tests durchgeführt. Dabei wurde zum einen das Verhalten von Lösungsgenauigkeit und insbesondere Rechenzeit in Abhängigkeit unterschiedlicher Parameter untersucht, zum anderen wurden Vergleichswerte zu anderer Software bzw. anderen Verfahren bestimmt.

Die einzelnen getesteten Kriterien werden im nachfolgenden Abschnitt aufgelistet, im Anschluss daran werden die Ergebnisse angegeben und ausgewertet.

6.3.1 Testkriterien im Überblick

Beim Test des parallelen Integralgleichungslösers mit verschiedenen Parametern sind zu unterscheiden:

- Verfahrensparameter, die bestimmte Größen im ausgeführten Verfahren bestimmen:
  - Taylorordnung T
  - Systemordnung N des Integralgleichungssystems
- Parallele Parameter: Bei gleichbleibender Netzwerkstruktur (siehe vorausgehender Abschnitt) und Kommunikationsart bzw. -bibliothek ist hier die Anzahl der verwendeten Prozessoren als Parameter zu betrachten.
- Problemparameter, d.h. die Betrachtung unterschiedlicher Probleme und damit unterschiedlicher Funktionen als Eingangsgrößen des Integralgleichungslösers. Da, soweit darstellbar, unterschiedlichste Funktionen verwendet werden können, ist es zunächst nur wichtig, die gleichen Funktionen als Eingangsparameter zur Berechnung von Werten, die unmittelbar verglichen werden sollen, zu verwenden. Die Funktion selbst ist nur bei Interesse an der Lösung des konkreten Problems oder zur Prüfung der Anwendbarkeit des Verfahrens relevant. In den meisten Tests wird daher auf die Angabe der Eingangsfunktionen verzichtet. Es werden jedoch an einigen Stellen explizit Beispiele angegeben, um zusätzlich einen Eindruck von den verwendeten Testproblemen zu erhalten.

Folgende Sachverhalte wurden im Hinblick auf die Parameter der ersten beiden Arten betrachtet:

- Entwicklung von Genauigkeit und Rechenzeit bei variabler Taylorordnung und fester Systemordnung
- Entwicklung von Genauigkeit und Rechenzeit bei variabler Systemordnung und fester Taylorordnung
- Rechenzeit für verschiedene Prozessorzahlen (parallele Performance)

- Anteile einzelner Programmteile an der Rechenzeit (bei variablen Parameterwerten)

Als Beispiel für Funktionen, die numerische Probleme mit sich bringen können, wird die Kernfunktion

$$K : [-0.5, 0.5] \times [-0.5, 0.5], (s,t) \mapsto \frac{\rho\alpha^2}{2 \cdot ((t-s)^2 + \alpha^2)^{\frac{3}{2}}}$$

aus [119] mit Parametern $0 < \alpha, \rho \leq 1$ betrachtet. Dieses Beispiel stellt gleichzeitig eine Anwendung aus der Physik dar, in der die Wärmeabstrahlung paralleler Platten betrachtet wird.

Zusätzlich werden Vergleiche mit folgender Software angestellt:

- alte serielle Software von Klein aus [72]
- Integralgleichungslöser nach Obermaier wie in Abschnitt 5.7 beschrieben

Folgende Vergleiche mit der Software aus [72] werden angestellt:

- Vergleich von Rechenzeit und Genauigkeit der alten und neuen seriellen Software
- Vergleich der absoluten Rechenzeit aus [72] mit den erzielten neuen Werten

Folgende Vergleiche mit der Software aus Abschnitt 5.7 werden angestellt:

- Vergleich der Rechenzeiten zum Erreichen einer bestimmten Lösungsgenauigkeit

Einzelne Zusatzaspekte werden im Rahmen der jeweiligen Tests diskutiert.

Je nach verwendetem Beispiel unterscheiden sich die nachfolgend dargestellten absoluten Ergebnisse deutlich, die relative Entwicklung der Ergebnisse bei veränderlichen Parameterwerte dagegen ist für die meisten getesteten Beispiele ähnlich, so dass sich die Ergebnisse anhand von Beispielwerten darstellen lassen. Dort, wo Ergebnisse direkt verglichen werden,

wurden jeweils identische Probleme zur Ermittlung der Vergleichszahlen zu Grunde gelegt.

6.3.2 Entwicklung der Genauigkeit bei variabler Taylorordnung bzw. variabler Systemordnung

Zunächst wird die Entwicklung von Lösungsgenauigkeit des parallelen Integralgleichungslösers in Abhängigkeit von Taylor- und Systemordnung betrachtet.

Da es sich bei den Lösungen von Integralgleichungen um Funktionen handelt, ist die Genauigkeit der Einschließung je nach betrachtetem Auswertungsintervall verschieden. Um die unten aufgeführten Werte für die Genauigkeit zu erhalten, werden die jeweiligen Lösungsfunktionen (Intervallpolynome) an verschiedenen Stellen ausgewertet und für die berechneten Intervalle jeweils die Anzahl der korrekten Ziffern (binär) bestimmt. Das Minimum dieser Werte wird dann als Genauigkeitswert für die Lösungsfunktion verwendet.

Die folgende Tabelle zeigt am Beispiel die Entwicklung der Genauigkeit der Lösungen für unterschiedliche Taylorordnungen und Systemordnungen.

Systemordnung → ↓ Taylorordnung	8	16	32	64	128	256
3	0	0	0	1	2	4
4	0	0	1	3	6	8
6	0	0	4	10	14	17
8	0	1	9	17	24	26
12	0	4	15	29	30	30
16	0	9	25	30	31	-
24	0	18	30	31	-	-
32	0	26	32	-	-	-

(Anzahl korrekte Ziffern binär)

Für die in der Tabelle mit „-“ gekennzeichneten Fälle wurden keine Werte ermittelt, um die erwartete Rechenzeit der getesteten Parameterkombina-

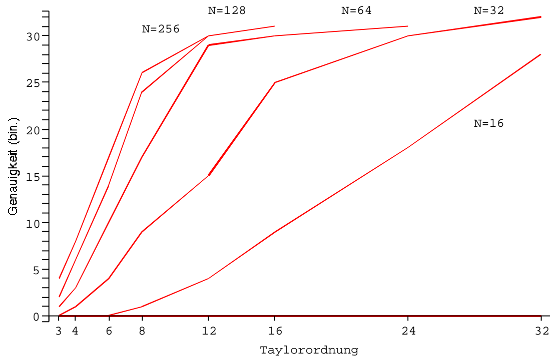

Abbildung 6.3: Lösungsgenauigkeit bei wachsender Taylorordnung und fester Systemordnung

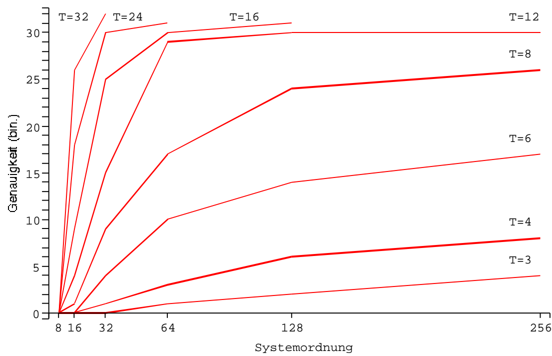

Abbildung 6.4: Lösungsgenauigkeit bei wachsender Systemordnung und fester Taylorordnung

tionen in handhabbaren Grenzen zu halten. Gleichzeitig wurde für Parameterkombinationen mit Werten für Taylorordnung T und Systemordnung N ab ca. $(T+1) \cdot N \geq 4096$ der zur Verfügung stehende Hauptspeicher überschritten, so dass diese Berechnungen nicht erfolgreich abgeschlossen werden konnten. Gleiches gilt für die nachfolgenden Tabellen.

Abbildung 6.3 zeigt die Genauigkeit in Abhängigkeit von der Taylorordnung für die fest gewählten Systemordnungen N der Tabelle, Abbildung 6.4 die Genauigkeit in Abhängigkeit von der Systemordnung für die fest gewählten Taylorordnungen T der Tabelle.

Sowohl die Erhöhung der Systemordnung als auch die Erhöhung der Taylorordnung steigerten die Genauigkeit der Lösungseinschließungen deutlich. Wurde nur einer der beiden Werte erhöht, so fiel der Genauigkeitsgewinn auch bei starker Erhöhung meist geringer aus als bei Kombination von erhöhter Taylorordnung und Systemordnung. Die Systemordnung musste im Vergleich zur Taylorordnung stärker erhöht werden, um Verbesserungen zu erzielen. Oft war zu beobachten, dass ab einem gewissen Wert der jeweiligen Ordnung, insbesondere der Systemordnung, die bis dorthin erreichte Genauigkeit bei weiterer Erhöhung der Ordnung nur noch mäßig oder nicht mehr stieg, auch wenn im Rahmen der verwendeten Arithmetik noch die Möglichkeit zur Verbesserung der Ergebniseinschließung bestand.

Der nächste Abschnitt zeigt, wie sich die Rechenzeit für die betrachteten Parameterkombinationen verhielt.

6.3.3 Entwicklung der Rechenzeit bei variabler Taylorordnung bzw. variabler Systemordnung

Für das obige Testszenario werden nun die Rechenzeiten betrachtet.

Die folgende Tabelle zeigt die Entwicklung der Rechenzeit für unterschiedliche Taylorordnungen und Systemordnungen bei einer festen Prozessorzahl von 4.

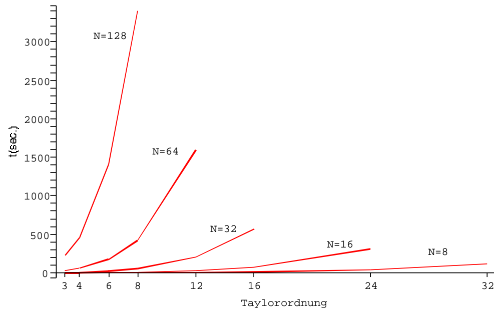

Abbildung 6.5: Rechenzeit bei wachsender Taylorordnung und fester Systemordnung

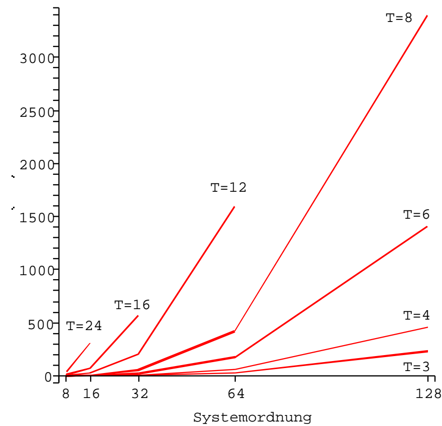

Abbildung 6.6: Rechenzeit bei wachsender Systemordnung und fester Taylorordnung

Systemordnung → ↓ Taylorordnung	8	16	32	64	128
3	0.1	0.5	3.7	28.4	229
4	0.1	0.9	7.3	57.2	455
6	0.4	2.8	22.0	174	1408
8	0.8	6.8	53.4	419	3400
12	3.2	26.0	206	1596	-
16	8.7	71.1	567	-	-
24	37.8	307.5	-	-	-
32	111	-	-	-	-

(Zeit in sec.)

Abbildung 6.5 zeigt die Rechenzeit in Abhängigkeit von der Taylorordnung für die fest gewählten Systemordnungen der Tabelle, Abbildung 6.6 die Rechenzeit in Abhängigkeit von der Systemordnung für die fest gewählten Taylorordnungen der Tabelle.

Bei Erhöhung der Taylorordnung stieg die Rechenzeit in den Tests schneller an als bei Erhöhung der Systemordnung. Allerdings war im Vergleich zur Taylorordnung eine stärkere Erhöhung der Systemordnung notwendig, um Genauigkeitsgewinne zu erzielen.

Betrachtet man das Verhältnis von Rechenzeit (bei fester Prozessorzahl) und erreichter Genauigkeit, so konnten im Beispiel die besten Werte für mittlere Werte von System- und Taylorordnung erzielt werden.

6.3.4 Parallele Performance

Es folgt nunmehr die Betrachtung der parallelen Performance, d.h. der Rechenzeit bei variabler Prozessorzahl.

Zunächst soll die Entwicklung der Gesamtrechenzeit des parallelen Integralgleichungslösers betrachtet werden.

Die folgende Tabelle zeigt die Entwicklung der Rechenzeit für verschiedene Prozessorzahlen bei unterschiedlichen festen Werten von Taylorordnung und Systemordnung. Die Entwicklung ist auch in Abbildung 6.7 dargestellt.

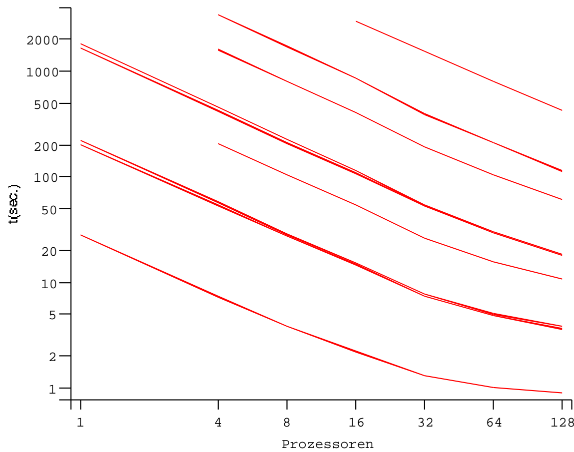

Abbildung 6.7: Rechenzeit der Beispiele bei wachsender Prozessorzahl (parallele Performance)

Prozessorzahl → Taylorordnung/ ↓ Systemordnung	1	4	8	16	32	64	128
4/32	28.0	7.3	3.8	2.2	1.3	1.0	0.9
4/64	219	57.2	28.4	15.3	7.7	5.0	3.8
4/128	1801	455	226	115	53.4	30.2	18.4
8/32	201	53.4	27.2	14.7	7.4	4.8	3.6
8/64	1642	419	208	108	53.1	29.3	18.0
8/128	-	3400	1714	862	392	209	113
12/32	-	206	104	53.6	26.1	15.7	10.8
12/64	-	1596	801	407	193	105	61.0
12/128	-	-	-	2974	1532	801	426

(Zeit in sec.)

Es zeigt sich, dass die Software gut parallelisiert: Für mittlere und hohe Parameterwerte ergibt sich eine lineare Beschleunigung, auch bei größeren

Prozessorzahlen. Lediglich bei kleineren Parameterwerten bzw. in Fällen, in denen die Gesamtrechenzeit ohnehin gering ist, ist die Beschleunigung kleiner. Bei derartigen Werten kommt Kommunikations-Overhead in der parallelen Umgebung stärker zum Tragen. Aufgrund der meist geringen absoluten Rechenzeit fallen diese Fälle jedoch nicht ins Gewicht.

Im Rahmen der hier getesteten Parameterwerte und Prozessorzahlen zeigt sich, dass mit Hilfe des parallelen Integralgleichungslösers hohe Zeitgewinne möglich sind: Mit der im Test verwendeten Maximalzahl von 128 Prozessoren konnten Lösungseinschließungen, deren Berechnungen seriell mehrere Stunden in Anspruch genommen hätte, in wenigen Minuten bestimmt werden. Der parallele Integralgleichungslöser erlaubt es damit in vielen Fällen gegenüber den alten seriellen Verfahrensimplementierungen sogar erst, überhaupt erfolgreich Lösungseinschließungen berechnen zu können (z.B. im Beispiel aus Abschnitt 6.3.6).

Nach der Gesamtrechenzeit soll nun noch anhand einer Auswertung von Beispielläufen die Parallelisierung der einzelnen Programmteile betrachtet werden. Hier wird zunächst die Verteilung des Rechenaufwands auf die Prozesse analysiert, die Entwicklung der Zeitanteile der Programmteile gegenüber der Gesamtzeit wird danach betrachtet.

Um die Anteile einzelner Programmteile an der Gesamtrechenzeit getrennt für alle Prozesse zu bestimmen, wurde die *MPI Parallel Environment (MPE)* [18] verwendet. Mit dieser Umgebung ist es möglich, Logdateien paralleler Programme im Format *SLOG* [19] anzulegen, in denen verschiedenste Programmaktivitäten benutzerdefiniert protokolliert werden können. Mit Hilfe der Anzeigesoftware *Jumpshot* [20] können die protokollierten Daten dann visualisiert werden. Die so erzeugten Logdateien sowie die Visualisierung beanspruchten schon bei Reduktion der Protokollierungsaktivitäten auf ein geringes Maß umfangreichen Speicherplatz, so dass eine erfolgreiche Verwendung und Visualisierung nur für kleinere und mittlere Parameterwerte und Prozessorzahlen möglich war. Es werden jedoch bereits hier die wesentlichen Aussagen zur Parallelisierung deutlich.

Der parallele Gleichungssystem-Löser ist durch die beschriebenen Techniken besonders stark parallelisiert und soll hier nicht in Einzelheiten betrachtet werden. Einige Aspekte zum Anteil des Gleichunssystem-Lösers finden sich unten.

Iterationsphase
Kommunikation und Leerlaufzeit
Aufbau des linearen Gleichungssystems
Matrixinversion
Weitere Matrixoperationen

Abbildung 6.8: Parallele Performance (Beispiel mit Taylorordnung 12, Systemordnung 32, 16 Prozessoren)

An dieser Stelle wird die Parallelisierung der beiden weiteren Hauptkomponenten des parallelen Integralgleichungslösers im Sinne der Rechenzeit betrachtet:

- Durchführung der Iterationen
- Aufbau des linearen Gleichungssystems

In Abbildung 6.8 sind die Aktivitäten der Prozesse für einen Beispiellauf mit Taylorordnung 12 und Systemordnung 32 bei einer Prozessorzahl von 16 (ein Prozess pro Prozessor[3]) dargestellt.

[3]Diese Beziehung gilt für alle dargestellten Beispiele.

Es ist zu erkennen, dass im Beispiel auch ohne adaptive Verteilungsschritte eine gleichmäßige Verteilung der Rechenzeiten ohne größere Leerlaufzeiten sowohl in der iterativen Phase als auch bei der Berechnung der Gleichungssystem-Einträge erreicht werden konnte.

6.3.5 Anteile einzelner Programmteile an der Rechenzeit

Nun werden noch die Anteile der einzelnen Programmteile an der Rechenzeit untersucht. Dabei werden die Anteile der drei Hauptphasen des Verfahrens (Verfahren 3.1.6 bzw. 4.2.2) betrachtet:

- Durchführung der Iterationen (mit Verfahren 3.1.8)
- Aufbau des linearen Gleichungssystems (Verfahren 3.1.9 bzw. 4.2.3)
- Lösung des linearen Gleichungssystems (Verfahren 3.2.1 bzw. 4.3.2)

Diese Phasen machen den Hauptanteil der Rechenzeit aus.

Zur Betrachtung der Zeitanteile für verschiedene Parameterwerte sind in den Abbildungen 6.9, 6.10 und 6.11 die Zeitanteile für ein Beispiel mit den drei Parameterwert-Kombinationen

- Taylorordnung $T = 3$, Systemordnung $N = 32$
- Taylorordnung $T = 12$, Systemordnung $N = 32$
- Taylorordnung $T = 3$, Systemordnung $N = 128$

bei jeweils 16 Prozessoren angegeben.

In Abbildung 6.9 wird zunächst der Programmablauf eines Beispiels für die vergleichsweise kleinen Parameterwerte

- $T = 3$, $N = 32$

für 16 Prozessoren betrachtet. In diesem Beispiel wird der größte Zeitanteil von der Iterationsphase des Integralgleichungslösers in Anspruch genommen, gefolgt von der Berechnung der Einträge für das lineare Gleichungssystem. Der parallele Löser nimmt an dritter Stelle ebenso einen relevanten Anteil der Zeit ein.

- Iterationsphase
- Kommunikation und Leerlaufzeit
- Aufbau des linearen Gleichungssystems
- Matrixinversion
- Weitere Matrixoperationen

Abbildung 6.9: Anteile der Programmphasen (Beispiel mit Taylorordnung 3, Systemordnung 32)

Wird die Taylorordnung erhöht, so ergibt sich für $T = 12$ und $N = 32$ das Bild in Abbildung 6.10. Der für die Berechnung der Einträge des linearen Gleichungssystems benötigte Anteil der Rechenzeit steigt deutlich an und nimmt den weitaus größten Teil der Rechenzeit ein. Betrachtet man die Abhängigkeit von der Taylorordnung T in Verfahren 3.1.9, so ist festzustellen, dass dieser Programmteil diesbezüglich den asymptotischen Aufwand $O(T^4)$ hat, während die anderen Programmteile nur höchstens den Aufwand $O(T^3)$ haben. Die Entwicklung entspricht somit dem asymptotischen Verhalten der Verfahren.

Abbildung 6.10: Anteile der Programmphasen (Beispiel mit Taylorordnung 12, Systemordnung 32)

Wird statt der Taylorordnung die Systemordnung erhöht, so ergibt sich für $T = 3$ und $N = 128$ das Bild aus Abbildung 6.11. Hier bleiben die Anteile der Programmteile im Gegensatz zum vorher betrachteten Fall stabil. Auch diese Entwicklung entspricht der asymptotischen Betrachtung für die Systemordnung N als Parameter, da alle drei Verfahrens-Hauptteile den Aufwand $O(N^3)$ besitzen.

6.3.6 Ein Beispiel

Das folgende Beispiel zeigt eine Integralgleichung zu einer Anwendung aus der Physik mit parameterabhängigem Kern, deren verifizierte Lösung bei Einsatz der Taylorarithmetik aufgrund numerischer Schwierigkeiten nur für bestimmte Parameterwerte gelingt.

Die untenstehende Integralgleichung tritt bei der Bestimmung der Wärmeabstrahlung in einem parallelen Plattensystem auf und wurde bereits von

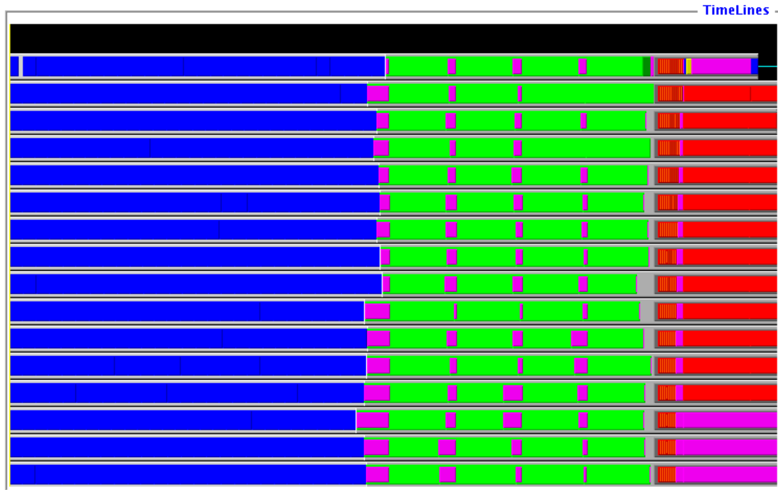

Abbildung 6.11: Anteile der Programmphasen (Beispiel mit Taylorordnung 3, Systemordnung 128)

Sparrow in [119] beschrieben. Aguirre-Ramirez und Shashanis geben in [1] Methoden zur näherungsweisen Lösung der Gleichung an. An dieser Stelle wird die verifizierte Lösbarkeit mit dem vorgestellten Integralgleichungslöser untersucht.

Die betrachtete Integralgleichung ist gegeben als

$$y(s) - \int_{-\frac{1}{2}}^{\frac{1}{2}} \frac{\rho\alpha^2}{2[(t-s)^2+\alpha^2]^{\frac{3}{2}}}\, y(t)\, dt = 1$$

In der Anwendung stellt dabei der Parameter $\rho \in [0,1]$ die Reflektivität des verwendeten Materials der Platten und $\alpha \in [0,1]$ das Verhältnis von Plattenabstand zu Plattenbreite dar.

Die Ableitungen der Kernfunktion sind durch Summen von Quotienten gegeben, bei denen im Nenner für höhere Ableitungen immer höhere Po-

tenzen der Art

$$[(t-s)^2+\alpha^2]^{\frac{2n+1}{2}}, n \in \mathbb{N}$$

auftreten, so dass für $(t-s)^2+\alpha^2 \ll 1$ immer höhere Werte der Summanden vorkommen, so dass einerseits der Wertebereich der Ableitungen immer größer wird, zum anderen immer größere numerische Probleme durch Auslöschung entstehen. Je größer also die Taylorordnung gewählt wird, desto größer werden auch die möglichen numerischen Probleme. Für Kernfunktionen dieser Art ist also durch Erhöhung der Taylorordnung keine Steigerung der Einschließungsgenauigkeit zu erwarten. Hier kann nur durch Erhöhung der Systemordnung ein Genauigkeitsgewinn erzielt werden.

Für die obige Funktion ist der Effekt umso stärker zu beobachten, je kleiner der Parameter $\alpha \in [0,1]$ gewählt wird. Im Folgenden wird die Integralgleichung für $\rho = 0.9$ und drei verschiedene Werte von α,

- $\alpha = 1.0$
- $\alpha = 0.8$ und
- $\alpha = 0.6$

betrachtet. Es wird jeweils die Genauigkeit der Einschließung als Anzahl der korrekten Ziffern (binär) angegeben.

Die erzielte Genauigkeit für die verschiedenen Parameterwerte ist in Tabelle 6.1 angegeben. Für die mit „-“ gekennzeichneten Fälle konnte hier keine Einschließung erzielt werden.

6.3.7 Vergleich mit der Software aus [72]

Bei Betrachtung der seriellen Software kann die alte Version von Klein aus [72] mit der neuen Software verglichen werden. Hier sind folgende Aspekte von Interesse:

- Veränderungen der Ergebnisgenauigkeit in der neuen Implementierung, insbesondere durch die Verwendung anderer bzw. veränderter Komponenten
- Vergleich der Rechenzeiten der Softwareversionen

Taylorord.	Systemord.	$\rho = 0.9$ $\alpha = 1.0$	$\rho = 0.9$ $\alpha = 0.8$	$\rho = 0.9$ $\alpha = 0.6$
3	8	2	0	-
3	16	5	2	0
3	32	7	4	1
3	64	8	6	3
3	128	10	7	5
3	256	12	9	6
4	8	2	-	-
4	16	5	0	-
4	32	7	2	0
4	64	8	4	1
4	128	10	7	3
4	256	12	9	5
6	8	2	-	-
6	16	5	0	-
6	32	7	1	-
6	64	9	4	0
6	128	11	5	1
6	256	12	8	2
8	8	0	-	-
8	16	4	-	-
8	32	6	0	-
8	64	8	1	-
8	128	10	4	-
8	256	11	5	0

(Anzahl korrekte Ziffern binär)

Tabelle 6.1: Einschließungsgenauigkeit für verschiedene Werte von α

Darüberhinaus kann ein Vergleich der absolut erzielten Ergebnisse angestellt werden, indem die ursprünglich und neu benötigte Rechenzeit unter Einbezug der parallelen Software und der weiterentwickelten, im Test verwendeten Hardware betrachtet wird.

Die nachfolgenden Tabellen zeigen zunächst am Beispiel die Einschließungsgenauigkeit der alten und neuen seriellen Software.

Systemordnung → ↓ Taylorordnung	2	4	8	16	32	64
3	0	2	6	9	11	12
6	6	8	10	11	12	13

Kleins Software (Anzahl korrekte Ziffern binär)

Systemordnung → ↓ Taylorordnung	2	4	8	16	32	64
3	0	2	6	9	12	13
6	6	8	10	12	13	14

Neue Software (Anzahl korrekte Ziffern binär)

In den untersuchten Beispielfällen konnte jeweils eine leicht verbesserte Genauigkeit der Ergebniseinschließungen festgestellt werden. Dies entspricht der Tatsache, dass in der neuen Software verbesserte Softwarekomponenten (C-XSC, Gleichungssystem-Löser, Taylorarithmetik) gegenüber dem Altsystem zum Einsatz kommen.

Nun wird die Rechenzeit am gleichen Beispiel betrachtet. Für beide Software-Versionen wurde die folgende serielle Umgebung verwendet:

- PC Pentium M760 2.0GHz, 1024MB Hauptspeicher, SUSE Linux 10.0, g++-4.0.2, C-XSC 2.1, Pascal-XSC 3.6.2

Systemordnung → ↓ Taylorordnung	2	4	8	16	32	64
3	0.1	0.3	0.8	6.5	31.0	445
6	0.1	0.8	5.0	38	270	2394

Klein (Zeit in sec.)

Systemordnung → ↓ Taylorordnung	2	4	8	16	32	64
3	0.1	0.2	0.4	3.4	11.4	206
6	0.1	0.3	2.6	17.2	135	1074

Neue Software (Zeit in sec.)

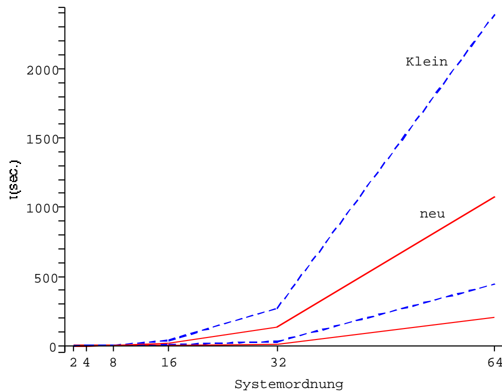

Abbildung 6.12: Rechenzeit des alten und neuen seriellen Integralgleichungslösers bei gleicher Hardware

Die Zeiten sind auch in Abbildung 6.12 dargestellt.

Vergleicht man statt der Rechenzeiten in der obigen Vergleichsumgebung die Zeiten unter Einbezug der neuen parallelen Software einerseits und der Original-Testumgebung[4] aus [72] andererseits, so ergibt sich ein beeindruckendes Bild, das neben den Beschleunigungsmöglichkeiten durch Parallelisierung, die schon früher in diesem Kapitel untersucht wurden, auch die Entwicklung der Hardware seit 1990 deutlich werden lässt.

Taylorord.	Systemord.	Klein	Neue Software
4	4	956	0.1
6	4	2417	0.1
4	8	6120	0.1
6	8	18240	0.2

Alte und neue Software (parallele Version, 32 Prozessoren) in Original-Testumgebung (Zeit in sec.)

[4] ATARI Mega ST 4 (8MHz Prozessor, 4MB Hauptspeicher)

Selbst die in Bezug auf die Rechenezeit größten Parameterwerte beim Test der alten Software auf der alten Testumgebung (Ergebnisse aus [72]) erfordern bei Berechnung mit der neuen parallelen Software weniger als eine Sekunde an Rechenzeit bei paralleler Berechnung auf 32 Prozessoren.

Die neue Software ermöglicht somit zusammen mit aktueller Hardware die Wahl höherer Parameterwerte und damit die Berechnung wesentlich engerer Einschließungen oder die erstmalige Einschließung der Lösung von Integralgleichungen, die im Rahmen von [72] nicht gelöst werden konnten. Letzteres betrifft etwa Integralgleichungen mit Quotienten in der Kernfunktion, so wie es im obigen Beispiel in Abschnitt 6.3.6 der Fall ist.

Nachfolgend sind drei weitere Beispiele zusammen mit den jeweils erzielten maximalen Einschließungsgenauigkeiten aufgeführt. Es zeigt sich, dass auch die Einschließungen der Lösungen von Integralgleichungen, die bereits durch die Software von Klein gelöst werden, durch die neue Software weiter verbessert werden können.

Kern	R. Seite	D	Klein	Neu
$3s\cosh(t)\sinh(t)$	$\sinh(s) - s\ \sinh^3(1)$	$[0,1]\times[0,1]$	32	37
$\frac{1}{\sqrt{2}} s\sin(s\sqrt{t+3})$	$\frac{1}{20}(s+\exp(s+3))$	$[0,1]\times[0,1]$	29	42
$\frac{1}{2\sqrt{7}} s\exp(st)$	$2-\exp(-s-3)$	$[0,1]\times[0,1]$	19	42

Alte und neue Software (Anzahl korrekte Ziffern binär)

6.3.8 Vergleich mit der Software nach Obermaier aus Abschnitt 5.7

Abschließend wird der neue parallele Integralgleichungslöser mit der Software nach Obermaier verglichen, die in den Abschnitten 3.3 bzw. 5.7 beschrieben wird.

Auch hier werden Genauigkeit und Rechenzeit als Messgrößen betrachtet. Da die Rechenzeit hier nicht in Bezug auf gewählte Parameterwerte

ausgedrückt werden kann, wird sie für jeweils feste erreichte Ergebnisgenauigkeiten bestimmt.

Die Rechenzeit zum Erreichen einer vorgegebenen Genauigkeit der Lösungseinschließung hängt beim Vergleich unterschiedlicher Verfahren und unterschiedlicher Arithmetiken (Taylorarithmetik und Steigungsarithmetik) stark von der jeweiligen Problemstellung, d.h. von den vorliegenden Funktionen in der Integralgleichung, ab. Im Hinblick auf den parallelen Integralgleichungslöser dieser Arbeit werden deshalb zwei unterschiedliche Fälle betrachtet:

- Fälle, in denen die Software gute Einschließungen liefert
- Fälle, in denen die Software schlechtere Einschließungen oder nur für hohe Parameterwerte Einschließungen liefert und bei denen der Einsatz eines anderen Verfahrens oder einer anderen Arithmetik von Nutzen sein kann

Der erste Fall liefert, am Beispiel

$$y(s) - 3\int_0^1 s\cosh(t)\sinh(t)y(t)\,dt = \sinh(s) - s\ \sinh^3(1)$$

betrachtet, folgende Zeiten:

Genauigkeit (dez./bin.)	Neuer paralleler Integralgl.-Löser	Par. Löser nach Obermaier
6 / $\approx$ 20	2.0	18
7 / $\approx$ 23	2.4	65
8 / $\approx$ 26	-	246
9 / $\approx$ 29	17.4	3497
10 / $\approx$ 33	114	13265

(16 Prozessoren, Zeit in sec.)

Hier genügten mittlere Parameterwerte, um mit dem neuen parallelen Integralgleichungslöser gute Einschließungen der Lösung zu erzielen, so dass die Rechenzeiten bei Verwendung von 16 Prozessoren vergleichsweise gering ausfielen. Die Software nach Obermaier dagegen benötigte z.T. mehr als das Hundertfache der Rechenzeit bei der gleichen Zahl von Prozessoren und derselben Testumgebung gemäß Abschnitt 6.1, um die gleiche

Einschließungsgenauigkeit zu erzielen. In den Rechenzeiten zeigen sich die Nachteile des aufwändigeren Berechnungsverfahrens von Obermaier, das in den meisten Fällen keine vergleichbar effiziente Lösung linearer Integralgleichungen erlaubt.

Für das Beispiel der Integralgleichung mit Quotientenfunktion aus Abschnitt 6.3.6

$$y(s) - \int_{-\frac{1}{2}}^{\frac{1}{2}} \frac{\rho\alpha^2}{2[(t-s)^2+\alpha^2]^{\frac{3}{2}}}\, y(t)\, dt = 1$$

ergibt sich dagegen für die Parameter $\rho = 0.9, \alpha = 1.0$ ein anderes Bild. Hier wurden folgende sehr unterschiedliche Rechenzeiten gemessen[5]:

Genauigkeit (dez./bin.)	Neuer paralleler Integralgl.-Löser	Par. Löser nach Obermaier
2 / $\approx$ 6	0.3	$<$ 0.5
3 / $\approx$ 10	12.8	$<$ 0.5
4 / $\approx$ 13	759	0.7
5 / $\approx$ 16	-	1.5
6 / $\approx$ 20	-	-
7 / $\approx$ 23	-	55
8 / $\approx$ 26	-	-
9 / $\approx$ 29	-	5561

(16 Prozessoren, Zeit in sec.)

Hier konnten mit dem neuen parallelen Integralgleichungslöser im Rahmen der möglichen Parameterwerte nur Ergebnisgenauigkeiten von bis zu 13 Stellen (binär; $\approx$ 4 Stellen dezimal) erreicht werden, wobei bereits stärker ansteigende Rechenzeiten zu verzeichnen waren. Die Software nach Obermaier dagegen erreicht hier wie im ersten Beispiel mit geringen Rechenzeiten von $<$ 1 min. eine Genauigkeit von 7 Stellen (dezimal; $\approx$ 23 Stellen binär), erst zum Erreichen höherer Genauigkeit steigt der Rechenaufwand stark an. Die Werte sind dabei mit denen aus dem

[5] Während der Test für die nicht vorliegenden Werte der ersten Spalte für hohe Genauigkeiten nicht erfolgreich verlief, wurden in der zweiten Spalte Werte ausgelassen, da sich bei der nächsthöheren Parameterkombination bereits eine höhere Genauigkeit ergab.

ersten Beispiel vergleichbar, d.h. auch hier ist eine besonders hohe Einschließungsgenauigkeit nur mit hohem Aufwand zu erreichen.

Es zeigt sich, dass die Software nach Obermaier als Software zur Lösung nichtlinearer Integralgleichungen bei der Lösung von linearen Integralgleichungen nur dann effizient auf lineare Integralgleichungen anwendbar ist, wenn besondere Problemstellungen vorliegen, die die hier vorgestellte neue Software zur Lösung linearer Integralgleichungen nicht oder nicht in angemessener Zeit bzw. im Rahmen der möglichen Parameterwerte löst. Gleichzeitig ist aber, auch durch die Verwendung der Steigungsarithmetik anstelle der Taylorarithmetik, die Anwendung auf einen größeren Kreis von Funktionen möglich, so dass Einsatzmöglichkeiten besonders in speziellen Fällen linearer Integralgleichungen bestehen.

Im Hinblick auf lineare Fredholmsche Integralgleichungen zweiter Art kann also festgestellt werden, dass der neue parallele verifizierte Löser den bisher einzigen bekannten dem Autor zur Verfügung stehenden parallelen verifizierten Löser, der die lineare Fredholmsche Integralgleichung zweiter Art als Spezialfall der nichtlinearen Fredholmschen Integralgleichung zweiter Art löst, in vielen Fällen in Bezug auf die Rechenzeit deutlich unterbietet und somit für viele Beispiele ermöglicht, genauere Ergebniseinschließungen als bisher möglich zu erzielen.

Bei den vorstehenden Betrachtungen wurde eine feste Prozessorzahl zu Grunde gelegt. Daher folgt noch eine kurze Betrachtung der Parallelisierung der Software unter den Bedingungen der Testumgebung wie in Abschnitt 6.1 beschrieben.

Aufgrund der speziellen Struktur der Software, die aus Einzelprogrammen mit steuernder GUI besteht (siehe auch Abschnitt 5.7.2), konnte die Software nur im Direktbetrieb und nicht im Batchbetrieb getestet werden. Daher beschränkt sich der Test der Parallelisierung auf eine Maximalzahl von 16 Prozessoren. Hierfür sind in der nachstehenden Tabelle Rechenzeiten für unterschiedlich hoch gewählte Parameterwerte angegeben. Dabei zeigten die Werte für beide oben angegebenen Beispiele ähnliche Entwicklungen, wie bereits aus den vorstehenden Tabellen deutlich wird. Für die nachfolgende Tabelle wurden Werte für das erste Beispiel ausgewählt.

Anstatt der Parameterwerte wird jeweils die erreichte Genauigkeit angegeben[6].

Prozessorzahl → ↓ Genauigkeit (dez.)	1	4	8	16
6	230	56	29	18
8	-	830	417	246
8	-	3248	1625	923
9 (*)	-	17364	7474	3497

(Zeit in sec.)

Die Läufe wurden mit dem lokalen Master-Slave-Verfahren, die Werte aus der mit (*) markierten Zeile der Tabelle mit dem klassischen Master-Slave-Verfahren gemäß Abschnitt 4.4 (S. 112) durchgeführt. Bis zu 8 Prozessoren ist eine lineare Beschleunigung zu erkennen, darüber wird die Beschleunigung etwas kleiner. Bei den Zeiten der Messreihe (*), die zunächst eine superlineare Beschleunigung zu zeigen scheinen, tritt ein besonders zu nennender Effekt auf: Bei Verwendung des klassischen Master-Slave-Verfahrens fungiert in der Software ein Prozess *ausschließlich* als Master, der keine Arbeitsaufträge selbst bearbeitet, so dass hier die Ressourcen nicht optimal genutzt werden. Somit steht jeweils ein Prozess weniger als angegeben zur tatsächlichen Bearbeitung zur Verfügung. Unter Berücksichtigung dieses Effekts ergibt sich umgerechnet auch hier eine etwa lineare Beschleunigung.

[6] Die dritte Messreihe ergab trotz höher gewählter Parameterwerte noch keine höhere dezimale Genauigkeit gegenüber der zweiten Messreihe.

7 Zusammenfassung und Weiterentwicklung

Abschließend werden noch einmal die Ergebnisse dieser Arbeit zusammengefasst und einige Möglichkeiten für weitere Untersuchungen und Weiterentwicklung der Software angegeben.

7.1 Zusammenfassung: Verfahren, Softwarekomponenten, Ergebnisse

In dieser Arbeit wurden zwei Zielsetzungen verfolgt:

- Konzeption und Implementierung einer Software zur effizienten und genauen parallelen verifizierten Lösung linearer Fredholmscher Integralgleichungen zweiter Art, aufbauend auf dem Verfahren von Klein [72]; Analyse, Test und Vergleich der Ergebnisse.
- Erstellung neuer und erweiterter Softwarekomponenten zur Erweiterung des Einsatzspektrums und Anwendungsportfolios der C++-Klassenbibliothek C-XSC, insbesondere zur einfachen Benutzung von C-XSC-Datentypen mit MPI in parallelen Umgebungen und zur Integration in existierende und neue Software sowie Export in andere Softwareumgebungen.

Zum Erreichen insbesondere des ersten Ziels wurden in Kapitel 2 zunächst mathematische Grundlagen für die vorgestellten Verfahren angegeben. Für die Taylorarithmetik wurde dann durch die Entwicklung neuer Rekursionsformeln die Anwendung der automatischen Differentiation auf zuvor nicht betrachtete Funktionen ermöglicht.

In Kapitel 3 wurden danach die zu Grunde liegenden Verfahren zur verifizierten Lösung linearer Fredholmscher Integralgleichungen und Integralgleichungssysteme zweiter Art dargestellt.

In Kapitel 4 wurden neue parallele Verfahren zu den beschriebenen seriellen Verfahren und Ihren Teilverfahren entwickelt.

In Kapitel 5 wurden zahlreiche neue oder erweiterte Softwarekomponenten vorgestellt, die zum Erreichen beider genannter Ziele konzipiert und implementiert wurden. Dabei handelt es sich um

- modifizerte Komponenten zur Anwendung bestehender Software auf neuen Plattformen bzw. in neuen Umgebungen
- vollständiges Redesign von Altsystemen
- Erweiterungen von Komponenten um neue Funktionalitäten sowie
- vollständig neu konzipierte Komponenten.

Insbesondere wurden Interfaces zur Anbindung an unterschiedliche Softwaresysteme geschaffen.

Im Einzelnen wurden verwirklicht:

- Erweiterte ein- und zweidimensionale Intervall-Taylorarithmetik in C++/C-XSC mit neuen mathematischen Funktionen und erweiterter Funktionalität
- Neues Kommunikationspaket in C++/C-XSC/MPI zur Benutzung von MPI-Funktionen mit den Datentypen der Klassenbibliothek C-XSC sowie mit den Datentypen der obigen Taylorarithmetik und STL-Vektoren
- Neu konzipierter serieller verifizierter Integralgleichungslöser für lineare Fredholmsche Integralgleichungen zweiter Art in C++/C-XSC nach dem Verfahren von Klein
- Neu konzipierter serieller verifizierter Systemlöser für lineare Fredholmsche Integralgleichungen zweiter Art in C++/C-XSC nach dem Verfahren von Klein

- Neu konzipierter paralleler verifizierter Systemlöser für lineare Fredholmsche Integralgleichungen zweiter Art in C++/C-XSC nach dem Verfahren von Klein
- Neuer paralleler verifizierter linearer Intervall-Gleichungssystem-Löser nach Rump in C++/C-XSC
- Modifizierter paralleler verifizierter Integralgleichungslöser nach Obermaier [106] in C++/Java zur Anwendung in der parallelen Umgebung *ALiCEnext*
- Modifizierter paralleler verifizierter globaler Optimierer in C++/C-XSC nach Wiethoff [131] zur weiteren Demonstration des Einsatzes des MPI-Kommunikationspakets für C-XSC-Datentypen
- Redesign des Maple-Pakets *intpakX* zur Intervallrechnung und Visualisierung der Ergebnisse von Intervall-Berechnungen in Maple
- Neues integriertes Interface zum Export der Lösung von Integralgleichungen mit dem Einzel- oder Systemlöser und deren Visualisierung in Maple mit dem Paket *intpakX*

Insbesondere stehen über die Anwendungen für Integralgleichungen (und Optimierung) hinaus

- die erweiterte Intervall-Taylorarithmetik
- das neue MPI-Kommunikationspaket
- der neue parallele verifizierte lineare Gleichungssystem-Löser
- das Maple-Paket *intpakX*

zur unabhängigen Verwendung zur Verfügung. Das Maple-Paket *intpakX*, der parallele lineare Gleichungssystem-Löser und insbesondere das Kommunikationspaket finden bereits in weiteren laufenden oder schon abgeschlossenen Arbeiten (Bachelor- und Masterarbeiten sowie Dissertationen, z.B. [13] [74] [88] [116] [121]) und sonstigen Publikationen (etwa [78]) Verwendung.

In Kapitel 6 wurde die erstellte Software, insbesondere der neue parallele verifizierte Integralgleichungslöser, getestet und die Ergebnisse dargestellt und erläutert.

Die Anwendung des neuen Kommunikationspakets zeigt zunächst, dass ein erfolgreicher und effizienter Einsatz im Rahmen der entstandenen Anwendungen möglich war und gleichzeitig ein Weg geschaffen wurde, in einfacher Weise MPI-Programme zu erstellen, in denen Kommunikation mit C-XSC-Objekten erfolgt.

Mit dem parallelen verifizierten Integralgleichungslöser für lineare Fredholmsche Integralgleichungen zweiter Art steht nach erfolgreichen Untersuchungen zur Parallelisierung und zum Verhalten von Lösungsqualität und Rechenzeit eine effiziente Software zur Verfügung, mit der lineare Fredholmsche Integralgleichungen zweiter Art durch den Übergang zu Integralgleichungssystemen mit erheblich höherer Einschließungsgenauigkeit bei dennoch kürzerer Rechenzeit als bisher möglich parallel gelöst werden können. Dabei konnte für verschiedene Funktionen die Genauigkeit der Einschließung durch die Wahl höherer Werte insbesondere der Systemdimension im Systemverfahren erheblich gesteigert werden. Für genügend große Parameterwerte bei der Lösung der betrachteten Probleme konnte eine annähernd lineare Beschleunigung der parallelen Software erreicht werden.

7.2 Weiterentwicklung

Die Ergebnisse dieser Arbeit eröffnen eine Anzahl von Perspektiven für die Weiterentwicklung. Exemplarisch werden im Folgenden einige Entwicklungsmöglichkeiten genannt.

Einsatz von Arithmetiken

Die Software zur parallelen verifizierten Lösung von linearen Integralgleichungen und Integralgleichungssystemen verwendet die Taylorarithmetik zur Darstellung von Funktionen. Am Vergleich mit der Software nach Obermaier ist die unterschiedliche Einsatzfähigkeit bei bestimmten Funktionen zu erkennen. Dies kann auch auf die unterschiedliche Arithmetik zurückgeführt werden. Es kann somit weiter untersucht werden, ob der Einsatz anderer Arithmetiken in dem vorgestellten verifizierten Integralgleichungslöser die Einsatzmöglichkeiten von Software und Verfahren verbessert. Da das Verfahren allerdings in zentralen Bestandteilen auf

die Taylorarithmetik ausgerichtet ist, muss zunächst die Anwendbarkeit geprüft werden und das Verfahren ggf. modifiziert werden. Zudem muss die entsprechende Arithmetik in C-XSC implementiert und als Modul zur Verfügung gestellt werden.

Einsatz des linearen Integralgleichungslösers als Teilverfahren

Der neue verifizierte lineare Integralgleichungslöser kann statt als Einzelanwendung auch als Komponente in anderen Verfahren eingesetzt werden. Das Verfahren von Obermaier etwa beinhaltet die Lösung linearer Integralgleichungen als Teilverfahren, bei dem untersucht werden kann, inwieweit sich die vorgestellte Software als Komponente einsetzen lässt. Die Erstellung einer solchen Software unter Einsatz des neuen verifizierten linearen Integralgleichungslösers stellt eine weitere Softwareentwicklungsaufgabe dar.

Parallelisierung der Komponenten

Für die neue parallele Software können zusätzliche Parallelisierungsmöglichkeiten untersucht werden:

- Können Iterationsphase und Aufbau des Gleichungssystems noch stärker parallelisiert werden, indem etwa eine aktive Verteilung der Teilprobleme durch den Masterprozess erfolgt? Wie oft ergibt sich, abhängig von der möglichen Performance und jeweils aktuellen Auslastung der parallelen Umgebung, bei der praktischen Anwendung eine relevante Beschleunigung beim Einsatz derartiger Strategien?

- Kann die Software so umgestaltet werden, dass sämtliche Daten verteilt vorliegen, so dass der Speicheraufwand jedes einzelnen Prozesses sinkt? Welche maximalen Parameterwerte können so erreicht werden und in welchen Fällen können so bessere Ergebnisse bei der Einschließungsgenauigkeit erreicht werden? Wie entwickelt sich in diesem Fall die Rechenzeit? Diese Fragestellung gilt insbesondere für den parallelen verifizierten linearen Gleichungssystem-Löser.

Bibliotheken

Die eingesetzten Bibliotheken, insbesondere C-XSC, werden ständig weiterentwickelt. Im Hinblick auf das Kommunikationspaket zur Benutzung von MPI-Funktionen mit C-XSC-Datentypen besteht eine mögliche Weiterentwicklung in der Entwicklung von Template-Matrix- und Vektor-Klassen. Die Existenz derartiger Klassen kann die Entwicklung solcher neuen Komponenten vereinfachen, da eine geringere Anzahl von Einzelfällen und Spezialisierungen betrachtet werden muss.

Gleichzeitig kann durch Erstellung modifizierter Versionen der in dieser Arbeit vorgestellten Software der Einsatz verschiedener Intervall-Bibliotheken untersucht werden. Hierbei ist allerdings zu beachten, dass auch der Funktionsumfang solcher Bibliotheken variiert und die in C-XSC vorhandenen Matrix-und Vektor-Datentypen sowie das exakte Skalarprodukt in anderen Bibliotheken nicht notwendigerweise vorhanden sind.

Weitere Entwicklungsmöglichkeiten

Umgekehrt können mit den neu zur Verfügung stehenden Komponenten auch neue Probleme gelöst werden, die unabhängig von den hier präsentierten Anwendungen sind. Der parallele verifizierte Gleichungssystem-Löser kann in anderen Anwendungen, die die verifizierte Lösung linearer Gleichungssysteme beinhalten, eingesetzt werden, das MPI-Kommunikationspaket für C-XSC-Datentypen spielt bei der Entwicklung paralleler Verfahren mit C-XSC eine wichtige Rolle, und durch die Weiterentwicklung der Anbindung von Maple an C-XSC können weitere Visualisierungsmöglichkeiten für Software geschaffen werden, die in C++/C-XSC implementiert wurde.

Anhang A: Installation und Anwendung der seriellen Software

Die Installation und Anwendung der seriellen Software erfolgt in wenigen einfachen Schritten:

- Entpacken des vorhandenen Archivs und Kopieren in ein geeignetes Verzeichnis im Dateisystem
- Anpassung des Makefiles und Einfügen der gewünschten Funktionen in die Beispielanwendung
- Übersetzen des Quellcodes
- Ausführung

Da sich das Vorgehen für den seriellen Einzel- und Systemlöser nur in unwesentlichen Details unterscheidet, beschränkt sich die nachfolgende Erläuterung auf die Angaben zum Systemlöser.

Umfang

Folgende Komponenten mit den nachfolgend angegebenen Dateien sind im Archiv enthalten:

- Taylorarithmetik:
  `itaylor.cpp`
  `itaylor.hpp`
  `dim2taylor.cpp`
  `dim2taylor.hpp`

- Linearer Gleichungssystem-Löser:
  `ilss.cpp`
  `ilss.hpp`
  `lss_aprx.cpp`
  `lss_aprx.hpp`
- Integralgleichungslöser:
  `System-FIGL-Loeser.cpp`
  `System-FIGL-Loeser.hpp` und
  `System-FIGL-Loeser-Bsp.cpp`
  als Anwendung, die das Hauptprogramm enthält
- Zusätzliche Operationen für Vektoren:
  `vectoroperations.cpp`
  `vectoroperations.hpp`

Folgende Komponenten werden zusätzlich benötigt:

- C-XSC Version 2.1 oder höher
- `g++` Version 3.3.1 oder höher

Die Software ist für die Verwendung mit dem Betriebssystem Linux/Unix gestaltet (Testumgebung: SUSE Linux Version 10.0 für die seriellen Programme).

Anpassungen im Makefile

Im Folgenden ist der erste Teil eines Beispiel-Makefiles angegeben.

```
#= Benutzereinstellungen. ============================

# C-XSC-Installationsverzeichnis
# z.B. /usr/local/cxsc
PREFIX=/opt/cxsc

#= Einstellungen fuer Compiler und Bibliotheken ======

CXX=g++                            # Compiler
CXXOPTS=-Wall -Winline             # Optionale Flags
CXXINC=-I$(PREFIX)/include         # Include-Pfad
CXXLIB=-L$(PREFIX)/lib             # Library-Pfad
```

```
CXXRPATH=-Wl,-R$(PREFIX)/lib     # dynamisches Linken
LIBRARIES=-lcxsc -lm             # Libraries

#= Programm ========================================
PROGRAM=System-FIGL-Loeser-Bsp # Programmname
```

Hier muss als Variable `PREFIX` das Verzeichnis angegeben werden, in dem C-XSC installiert ist. Die weiteren Angaben können unverändert bleiben. Wenn jedoch die Benutzung mit einem eigenen Anwendungsprogramm vorgesehen ist, das den Löser hinzubindet, muss neben der Einbindung der Headerdatei `System-FIGL-Loeser.hpp` in der Anwendung der Programmname in der Variable `PROGRAM` durch den Dateinamen der neuen Anwendung ersetzt werden.

Einfügen der gewünschten Funktionen in die Eingabedatei

Die Funktionen und Definitionsbereiche, durch die die Integralgleichung definiert ist, müssen in der Beispielanwendung oder der jeweils gewünschten Anwendung in C++/C-XSC-Syntax angegeben werden oder wahlweise aus einer separaten Datei eingebunden werden. Hier ein Beispiel für die folgende Integralgleichung:

$$y(s) - 3\int_0^1 s\cosh(t)\sinh(t)y(t)\,dt = \sinh(s) - s\ \sinh^3(1)$$

```
dim2taylor K(dim2taylor_vector& x)
{
  dim2taylor s=x[1];
  dim2taylor t=x[2];
  dim2taylor erg;

  erg=interval(3.0)*s*cosh(t)*sinh(t);          //(*)

  return erg;
}

itaylor ReSeite(itaylor& s)
{
 itaylor erg;
```

```
 erg=sinh(s)
     -s*pow(sinh(interval(1.0)),interval(3.0)); //(*)

 return erg;
}

const interval KDefS=interval(0.0,1.0);
const interval KDefT=interval(0.0,1.0);
const interval ReSeiteDef=interval(0.0,1.0);
```

Im gezeigten Code muss nur jeweils der Funktionsausdruck in der Zeile `(*)` für Kern `K` und rechte Seite `ReSeite` durch den gewünschten neuen Ausdruck ersetzt werden. Es empfiehlt sich, Konstanten wie im Beispiel explizit in den gewünschten Datentyp umzuwandeln.

Das Programm kann dann mit Hilfe des bekannten Werkzeuges `make` übersetzt und danach ausgeführt werden.

Anhang B: Installation und Anwendung der parallelen Software auf ALiCEnext

Für die Benutzung der Software in parallelen Umgebungen sind mehr Schritte erforderlich als zur Benutzung der seriellen Software.

Parallelrechner bestehen aus sehr unterschiedlicher Hardware und sind meist hochgradig individuell konfiguriert. Diese Beschreibung gilt daher nur für die Benutzung in der parallelen Testumgebung *ALiCEnext*, für andere Umgebungen ergeben sich mit großer Wahrscheinlichkeit Änderungen, die in der jeweiligen Umgebung begründet sind und hier nicht berücksichtigt werden können.

Die Installation und Anwendung der parallelen Software erfolgt in folgenden Schritten:

- Entpacken des vorhandenen Archivs und Kopieren in ein geeignetes Verzeichnis im Dateisystem
- Anpassung des Makefiles und Einfügen der gewünschten Funktionen in die Eingabedatei `functions.in`
- Übersetzen des Quellcodes
- Anlegen eines Batch-Skriptes zur Übergabe an das Batchsystem für die Ausführung
- Einstellen des Batchskriptes in eine geeignete *Queue* des Batchsystems

Der Umgang mit den in den Berechnungen zu verwendenden Funktionen und die Übersetzung des Quellcodes wird bereits bei der Beschreibung der Installation und Benutzung der seriellen Software beschrieben. Das Makefile enthält in der parallelen Version zusätzliche Angaben zu MPI: Include- und Library-Pfad und Übersetzungsoptionen in den Variablen `MPIINCPATH` und `MPILIBOPTS`, die im Regelfall aber nicht modifiziert werden müssen.

Umfang

Folgende Komponenten mit den nachfolgend angegebenen Dateien sind im Archiv enthalten:

- Taylorarithmetik:
  `itaylor.cpp`
  `itaylor.hpp`
  `dim2taylor.cpp`
  `dim2taylor.hpp`
- MPI-Kommunikationspaket für C-XSC-Datentypen, Taylorarithmetik und STL-Vektoren:
  `cxsc_mpicomm_templ.cpp`
  `cxsc_mpicomm_templ.hpp`
  `taylor_mpicomm.cpp`
  `taylor_mpicomm.hpp`
  `vector_mpicomm.cpp`
  `vector_mpicomm.hpp`
- Linearer Gleichungssystem-Löser:
  `ilss_par.cpp`
  `ilss_par.hpp`
  `lss_aprx_par.cpp`
  `lss_aprx_par.hpp`
  `matmul_par.cpp`
  `matmul_par.hpp`
- Integralgleichungslöser:
  `System-FIGL-Loeser-par.cpp`
  `System-FIGL-Loeser-par.hpp` und

`System-FIGL-Loeser-par-Bsp.cpp`
als Anwendung, die das Hauptprogramm enthält

- Zusätzliche Operationen für Vektoren:
  `vectoroperations.cpp`
  `vectoroperations.hpp`
- Eingabedatei für Funktionen:
  `functions.in`

Folgende Komponenten werden zusätzlich benötigt:

- C-XSC Version 2.1 oder höher
- `g++` Version 3.3.1 oder höher

Die Software ist für die Verwendung mit dem Betriebssystem Linux/Unix gestaltet (Testumgebung: SUSE Linux Version 10.0 für die seriellen Programme).

Batch-Verarbeitung

Zur Ausführung auf dem Parallelrechner *ALiCEnext* ist ein Batchskript anzulegen, da die Ausführung von Programmen in der Regel nur im Batchbetrieb möglich ist. Informationen hierzu finden sich auch unter [4].

Ein Beispielskript ist nachfolgend angegeben:

```
#!/bin/bash

#####################################################
# PBS-Optionen
#####################################################

# Anzahl zu verwendender Knoten (1 Proz. pro Knoten)
#PBS -l nodes=16

# PBS erstellt eine Datei, in der die Namen der
# tatsaechlich zur Verfuegung gestellten
# Knoten gespeichert werden. Der Name dieser
# Datei wird in der Umgebungsvariable
# $PBS_NODEFILE gespeichert.
# Anzahl zur Verfuegung gestellter Prozessoren
```

```
# zaehlen:
NPROCS=`wc -l < $PBS_NODEFILE`

#####################################################
# Verzeichnisse
#####################################################

TARGETDIR="/home/meinname"  # ein eigenes Verzeichnis
DATADIR=  "/data/meinname"  # lokales Verzeichnis
                            # auf dem jew. Knoten
                            # (auf ALiCEnext immer
                            # /data/<loginname> )
RESULTDIR=$TARGETDIR"/data-"`date +%d%m%y_%H%M%S`
                            # Moegliches Verzeichnis
                            # fuer Ausgabedateien
                            # mit akt. Datum und Zeit
mkdir $RESULTDIR            # anlegen

#####################################################
# Programmname
#####################################################

TARGETPROGRAM="System-FIGL-Loeser-par-Bsp"

#####################################################
# Programmaufruf
#####################################################

cd $TARGETDIR

./$TARGETPROGRAM -np $NPROCS 3 03 8 008 \
  $DATADIR $RESULTDIR

# Aufrufparameter:
#  -np $NPROCS : Prozessorzahl (entstpricht der
#                tatsaechlich von PBS zur Verfuegung
#                gestellten Prozessorzahl, s.o.)
#  3           : Taylorordnung
#  003         : String zur Angabe der Taylorordnung
#                im Ausgabedateinamen
#  8           : Systemordnung
```

```
#   008          : String zur Angabe der Systemordnung
#                  im Ausgabedateinamen
#   $DATADIR     : Oben definiertes Verzeichnis fuer
#                  Dateiausgaben auf dem
#                  jeweiligen Knoten
#   $RESULTDIR   : Oben definiertes Verzeichnis, in das
#                  die Ausgabedateien am Ende
#                  verschoben werden
```

Die einzelnen Parameter, insbesondere für den Programmaufruf, sind oben als Kommentare erklärt.

Notwendig für die Bearbeitung durch das Batchsystem sind nur

- die Angabe der angeforderten Zahl an Prozessoren
- der Aufruf des Programms

Es ist jedoch zu beachten, dass während der Ausführung auf den einzelnen Knoten keine Dateioperationen im *Home*-Verzeichnis des Benutzers stattfinden sollten, da dieses über das Netzwerk eingebunden wird und so bei größeren Ein- und Ausgaben aus und in Dateien eine starke Netzwerkbelastung erzeugt wird, da meist zahlreiche Programme von verschiedenen Benutzern gleichzeitig ausgeführt werden.

Die lokale Verwaltung der Ausgabedateien erfolgt im Anwendungsprogramm und ist in den zu Verfügung stehenden Anwendungen der obigen Empfehlung entsprechend geregelt.

Das Batchskript kann nunmehr an das Batchsystem zur Ausführung übergeben werden mit dem Kommandozeilenaufruf

```
qsub -q <queue> <batchskript>
```

Hierbei muss `<queue>` durch den Namen einer Queue ersetzt werden, `<batchskript>` durch den Namen des Batchskriptes. Es stehen z.B. die Queues `short`, `medium` und `large` zur Verfügung, die sich in der maximal zur Verfügung stehenden Rechenzeit unterscheiden. Aktuelle Informationen sind [4] zu entnehmen oder bei der Administration zu erfahren.

Literaturverzeichnis

[1] Aguirre-Ramirez, G.; Shashanis, O.: *A numerical solution for integral equations.* International Journal for Numerical Methods in Engineering, Vol. 15, Issue 10, pp. 1575-1579, 1980.

[2] Alefeld, G.; Herzberger, J.: *Einführung in die Intervallrechnung.* Bibliographisches Institut (Reihe Informatik, Nr. 12), Mannheim / Wien / Zürich, 1974.

[3] Alefeld, G.; Herzberger, J.: *An Introduction to Interval Computations.* Academic Press, New York, 1983.

[4] Informationen über ALiCEnext:
`http://www.alicenext.uni-wuppertal.de.`

[5] ANSI and IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*, Std 754-1985, ANSI/IEEE, New York, 1985.

[6] Appell, J.; Väth, M.: *Elemente der Funktionalanalysis.* Vieweg, Wiesbaden, 2005.

[7] Atkinson, K.E.: *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind.* Society for Industrial and Applied Mathematics, Philadelphia, 1976.

[8] Bode, A.: *Klassifikation paralleler Architekturen.* In: [129], S.11-40.

[9] Bohlender, G.; Lüderitz Kolberg, M.; Claudio, D. M.: *Modifications to Expression Evaluation in C-XSC.* Preprint BUW-WRSWT-2005/5, Wissenschaftliches Rechnen / Softwaretechnologie, Bergische Universität Wuppertal, 2005.

[10] Blomquist, F.; Hofschuster, W.; Krämer, W.: *Real and Complex Taylor Arithmetic in C-XSC.* Preprint BUW-WRSWT-2005/4, Wissenschaftliches Rechnen / Softwaretechnologie, Bergische Universität Wuppertal, 2005.

[11] Bräuer, M.C.: *Berechnungsmethoden für Ableitungen und Steigungen und deren Realisierung in C-XSC.* Diplomarbeit, Universität Karlsruhe, 1999.

[12] Bräuer, M.; Hofschuster, W.; Krämer, W.: *Steigungsarithmetiken in C-XSC.* Preprint BUGHW-WRSWT-2001/03, Wissenschaftliches Rechnen / Softwaretechnologie, Bergische Universität Wuppertal, 2001.
`http://www.math.uni-wuppertal.de/wrswt/literatur.html`.

[13] Braun, S.: *Visualisierung der exakten Lösungsmengen von linearen Intervallgleichungssystemen in Maple.* Bachelor-Thesis, Bergische Universität Wuppertal, 2006.

[14] Bronstein, I.N.; Semendjajew, A.; Musiol, G.; Mühlig, H.: *Taschenbuch der Mathematik.* Harry Deutsch, Thun/Frankfurt a.M., 1997.

[15] Brönnimann, H.; Melquiond, G.; Pion, S.: *A Proposal to Add Interval Arithmetic to the C++ Standard Library.* Draft (revision 2), November 2006.
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2137.pdf`.

[16] *Informationen zu CentOS*.
`http://www.centos.org`.

[17] Chambers, Ll.G.: *Integral Equations.* International Textbook Company, London, 1976.

[18] Chan, A.; Gropp, W.; Lusk, E.: *Users Guide for MPE: Extensions for MPI Programs*. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, 1998.
`http://www.mcs.anl.gov/mpi/mpich1/docs/mpeman.pdf`.

[19] Chan, A.; Gropp, W.; Lusk, E.: *Scalable Log Files for Parallel Program Trace Data(DRAFT)*. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, 2000.

[20] Chan, A.; Ashton, D.; Gropp, W.; Lusk, R.: *Jumpshot-4 Users Guide.* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, 2005. ftp://ftp.mcs.anl.gov/pub/mpi/slog2/ js4-usersguide.pdf.

[21] Chiriaev, D.; Walster, G.W.: *Interval Arithmetic Specification.* Draft, 1998. http://www.mscs.mu.edu/˜globsol/Papers/spec.ps

[22] *C-XSC Download:* http://www.math.uni-wuppertal.de/wrswt/xsc/cxsc_new.html.

[23] Devaney, J. E.; Michel, M.; Peeters, J.; Baland, E.: *AutoMap: A Software Tool for the Automatic Creation of MPI Data Structures From User Code*. Technical report, NIST, April 1997.

[24] Devaney, J. E.; Michel, M.; Peeters, J.; Vrielink, K.: *AutoLink: An MPI C Library For Sending and Receiving Dynamic Data Structures.* Technical report, NIST, April 1997.

[25] Dobner, H.-J.: *Numerische Methoden zur verifizierten Lösung von Integralgleichungen.* Habilitationsschrift, Universität Karlsruhe, 1991.

[26] Dongarra, J.J.; Luszczek, P.; Petitet, A.: *The LINPACK Benchmark: Past, Present, and Future.* Concurrency and Computation: Practice and Experience 15(9):803-820, August 2003. http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf.

[27] Dongarra, J.J.: *Performance of Various Computers Using Standard Linear Equations Software (Linpack Benchmark Report)* University of Tennessee Computer Science Technical Report, CS-89-85, 2006. http://www.netlib.org/benchmark/performance.ps.

[28] Engl, H.W.: *Integralgleichungen*. Springer, Wien/New York, 1997.

[29] Fischer, G.: *Lineare Algebra.* 9. Auflage, Vieweg, Braunschweig, 1986.

[30] Fischer, H.-C.: *Schnelle automatische Differentiation, Einschließungsmethoden und Anwendungen.* Dissertation, Universität Karlsruhe, 1990.

[31] Flynn, M.J.: *Very High Speed computing Systems.* Proc. IEEE, Vol. 54, pp.1901-1909, 1966.

[32] Forster, O.: *Analysis 1.* Vieweg, Braunschweig/Wiesbaden, 1983.

[33] Forster, O.: *Analysis 2.* Vieweg, Braunschweig/Wiesbaden, 1984.

[34] Frommer, A.: *Lösung linearer Gleichungssysteme auf Parallelrechnern.* Vieweg, Braunschweig, 1990.

[35] Geulig, I., Krämer, W.: *Intervallrechnung in Maple - Die Erweiterung intpakX zum Paket intpak der Share-Library.* Preprint 99/2, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1999.

[36] Gienger, A.: *Zur Lösungsverifikation bei Fredholmschen Integralgleichungen zweiter Art.* Dissertation, Universität Karlsruhe, 1997.

[37] *GNU gcc/g++.* `http://gcc.gnu.org`.

[38] Golub, G.; Van Loan, C.F.: *Matrix Computations.* The John Hopkins University Press, Baltimore, 1989.

[39] Goujon, D.; Michel, M.; Peeters, J.; Devaney, J.: *AutoMap and AutoLink: Tools for Communicating Complex and Dynamic Datastructures Using MPI.* Lecture Notes in Computer Science, Volume 1362, pp 98-109, Springer, 1998.

[40] Griewank, A.: *On Automatic Differentiation.* In: Iri, M.; Tanabe, K. (Ed.): *Mathematical Programming: Recent Developments and Applications*, pp. 83-108, Kluwer Academic Publishers, 1989.

[41] Griewank, A.; Corliss, G. (Eds.): *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Proceedings of Workshop on Automatic Differentiation at Breckenridge, SIAM, Philadelphia, 1991.

[42] Griewank, A.: *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, 2000.

[43] Grimmer, M.: *Maple Research PowerTool Interval Arithmetic.* 2002. `http://www.maplesoft.com/applications/app_center_browse.aspx?CID=13&SCID=155`

[44] Grimmer, M.: *Interval Arithmetic in Maple with intpakX.* In: PAMM - Proceedings in Applied Mathematics and Mechanics, Vol. 2, Nr. 1, p. 442-443, Wiley-InterScience, 2003.

[45] Grimmer, M.; Petras, K.; Revol, N.: *Multiple Precision Interval Packages: Comparing Different Approaches.* In: Numerical Software with Result Verification, Lecture Notes in Computer Science, Volume 2991/2004, pp. 64 - 90, Springer, Heidelberg, 2004.

[46] Grimmer, M.: *An MPI Extension for the Use of C-XSC in Parallel Environments.* Preprint BUW-WRSWT-2005/3, Universität Wuppertal, 2005. `http://www.math.uni-wuppertal.de/wrswt/literatur/lit_wrswt.html`.

[47] Gropp, W.; Lusk, E.; Skjellum, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, Massachusetts, 1999.

[48] Gropp, W.; Lusk, E.; Thakur, R.: *Using Mpi-2: Advanced Features of the Message Passing Interface.* MIT Press, Cambridge, Massachusetts, 2000.

[49] Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A.: *A high-performance, portable implementation of the MPI Message-Passing Interface standard.* Parallel Computing, 22(6):789828, 1996.

[50] Gropp, W.; Lusk, E.: *Installation and User's Guide to MPICH, a portable implementation of MPI.* Mathematics and Conputer Science Division, Argonne National Laboratory, Argonne. `http://www-unix.mcs.anl.gov/mpi/mpich1/docs/mpichman-chp4.pdf`.

[51] Hackbusch, W.: *Integralgleichungen: Theorie und Numerik.* Teubner, Stuttgart, 1989.

[52] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *Numerical Toolbox for Verified Computing I: Basic Numerical Problems.* (Vol. II siehe [76], C++-Version siehe [53]) Springer, Berlin / Heidelberg / New York, 1993.

[53] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *C++ Toolbox for Verified Computing: Basic Numerical Problems.* Springer, Berlin / Heidelberg / New York, 1995.

[54] Hausdorff, F.: *Mengenlehre.* De Gruyter, Leipzig, 1972.

[55] Heuser, H.: *Funktionalanalysis*. Teubner, Stuttgart, 1975.

[56] Heuser, H.: *Funktionalanalysis*. 2. Auflage, Teubner, Stuttgart, 1986.

[57] Ibraev, S.: *A new parallel method for verified global optimization.* Dissertation, Universität Wuppertal, 2001.

[58] Hillson, R.; Iglewski, M.: *C++2MPI: A Software Tool for Automatically Generating MPI Datatypes from C++ Classes*. In: IEEE Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC 2000), 27-30 August 2000, Trois-Rivieres, Quebec, pp. 13-17.

[59] Hofschuster, W.; Krämer, W.: *FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format*, Preprint 98/7 des Instituts für Wissenschaftliches Rechnen und Mathematische Modellbildung (IWRMM), Universität Karlsruhe, 1998.

[60] Hofschuster, W.; Krämer, W.; Wedner, S.; Wiethoff, A.: *C-XSC 2.0 – A C++ Class Library for Extended Scientific Computing.* Preprint BUGHW-WRSWT-2001/1, Wissenschaftliches Rechnen / Softwaretechnologie, Bergische Universität Wuppertal, 2001. `http://www.math.uni-wuppertal.de/wrswt/preprints/prep_01_1.pdf`

[61] Hofschuster, W.; Krämer, W.: *C-XSC 2.0 – A C++ Class Library for Extended Scientific Computing.* In: *Numerical Software with Result Verification*, R. Alt, A. Frommer, B. Kearfott, W. Luther (eds), Springer Lecture Notes in Computer Science 2991, pp. 15–35, 2004.

[62] Hölbig, C.; Krämer, W.: *Selfverifying Solvers for Dense Systems of Linear Equations Realized in C-XSC*. Preprint BUGHW-WRSWT-2003/1, Wissenschaftliches Rechnen / Softwaretechnologie, Bergische Universität Wuppertal, 2003.

[63] Huber, W.: *Paralleles Rechnen.* Oldenbourg, Wien, 1997.

[64] ISO/IEC: *ISO/IEC 14882:2003(E), Programming Languages – C++*. ISO/IEC, 2003.

[65] Kaplan, D.; Stevens, R.: *Processing Graph Method 2.0 Semantics*. Naval Research Laboratory, September 1995.

[66] Kaucher, E.; Miranker, W. L.: *Self-Validating Numerics for Function Space Problems.* Academic Press, New York, 1984.

[67] Kelch, R.: *Ein adaptives Verfahren zur numerischen Quadratur mit automatischer Ergebnisverifikation.* Dissertation, Universität Karlsruhe, 1989.

[68] Kienitz, A.: *Untersuchungen zum Einsatz von Taylormodellen bei der verifizierten Lösung von Gleichungssystemen.* Diplomarbeit, Rheinisch-Westfälische Technische Hochschule Aachen, 2003.

[69] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: *PASCAL–XSC — Sprachbeschreibung mit Beispielen.* Springer, Berlin/Heidelberg/New York, 1991.

[70] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: *PASCAL–XSC — Language Reference with Examples.* Springer, Berlin/Heidelberg/New York, 1992.

[71] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: *C–XSC, A C++ Class Library for Extended Scientific Computing.* Springer, Berlin/Heidelberg/New York, 1993.

[72] Klein, W.: *Zur Einschließung der Lösung von linearen und nichtlinearen Fredholmschen Integralgleichungssystemen zweiter Art.* Dissertation, Universität Karlsruhe, 1990.

[73] Klein, W.: *Enclosure Methods for Linear and Nonlinear Systems of Fredholm Integral Equations of the Second Kind.* In: Adams, E., Kulisch, U.: *Scientific computing with automatic result verification*. Academic Press, Boston 1993.

[74] Kolelis, K.: *Verifizierte parallele Berechnung der Lösungsmenge von parameterabhängigen Gleichungssystemen und deren Visualisierung.* Master-Thesis, Bergische Universität Wuppertal. In Arbeit, vorauss. Fertigstellung 2007.

[75] Krawczyk, R.: *Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken.* Habilitationsschrift, Universität Karlsruhe, 1969.

[76] Krämer, W.; Kulisch, U.; Lohner, R.: *Numerical Toolbox for Verified Computing II.* (Vol. I siehe [52]) Draft, Universität Karlsruhe, 1996. `http://www.uni-karlsruhe.de/~Rudolf.Lohner/papers/tb2.ps.gz`

[77] Krämer, W.; Blomquist, F.: *Algorithms with Guaranteed Error Bounds for the Error Function and the Complementary Error Function.* Preprint BUGHW-WRSWT-2000/2, Bergische Universität Wuppertal, 2000.

[78] Krämer, W.: *Introduction to the Maple Power Tool intpakX.* Preprint BUW-WRSWT-2006/9, Bergische Universität Wuppertal, 2006.

[79] Kuhlins, S.; Schader, M.: *Die C++-Standardbibliothek. Einführung und Nachschlagewerk* Springer, Berlin, 2005.

[80] Kulisch, U.: *Grundlagen des Numerischen Rechnens — Mathematische Begründung der Rechnerarithmetik.* Reihe Informatik, Band 19, Bibliographisches Institut, Mannheim/Wien/Zürich, 1976.

[81] Kulisch, U.; Miranker, W. L.: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[82] Kulisch, U.; Miranker, W. L. (Eds.): *A New Approach to Scientific Computation.* Proceedings of Symposium held at IBM Research Center, Yorktown Heights, N. Y., 1982. Academic Press, New York, 1983.

[83] Kulisch, U. (Ed.): *PASCAL–SC: A PASCAL extension for scientific computation*, Information Manual and Floppy Disks, Version ATARI ST. B. G. Teubner, Stuttgart, 1987.

[84] Kulisch, U. (Ed.): *Wissenschaftliches Rechnen mit Ergebnisverifikation — Eine Einführung.* Ausgearbeitet von S. Geörg, R. Hammer und D. Ratz. Vol. 58. Akademie Verlag, Berlin, und Vieweg Verlagsgesellschaft, Wiesbaden, 1989.

[85] Lerch, M.; Tischler, G.; Wolff von Gudenberg, J.; Hofschuster, W.; Krämer, W.: *The Interval Library filib++ 2.0 - Design, Features and Sample Programs.* Preprint BUGHW-WRSWT-2001/4, Wissenschaftliches Rechnen / Softwaretechnologie, Bergische Universität Wuppertal, 2001.

[86] Lerch, M.; Tischler, G.; Wolff von Gudenberg, J.; Hofschuster, W.; Krämer, W.: *FILIB++, a fast interval library supporting containment computations.* ACM Transactions on Mathematical Software, Vol. 32 No. 2, S. 299-324, 2006.

[87] Lohner, R.: *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen.* Dissertation, Universität Karlsruhe, 1988.

[88] Lüderitz Kolberg, M.: *(Titel noch nicht bekannt).* Dissertation, Pontifica Universidade Catolica de Rio Grande do Sul, Porto Alegre, Brasilien. Noch kein Veröffentlichungstermin.

[89] Informationen zum Computer-Algebra-System *Maple*: `http://www.maplesoft.com`.

[90] Meise, R.; Vogt, D.: *Einführung in die Funktionalanalysis.* Vieweg, Braunschweig/Wiebaden, 1992.

[91] Message Passing Interface Forum. `http://www.mpi-forum.org`.

[92] Message Passing Interface Forum: *MPI: A Message Passing Interface Standard.* University of Tennessee, Knoxville, Tennessee, 1993-1995.

[93] Message Passing Interface Forum: *MPI-2: Extensions to the Message Passing Interface.* University of Tennessee, Knoxville, Tennessee, 1995-1997.

[94] Michel, M.; Devaney, J.: *A Generalized Approach for Transferring Data-Types with Arbitrary Communication Libraries.* In: Proceedings of the Workshop on Multimedia Network Systems (MMNS 2000) at the 7th International Conference on Parallel and Distributed Systems, Iwate, Japan, July 4-7, 2000.

[95] Michlin, S.G.: *Vorlesungen über lineare Integralgleichungen.* VEB Deutscher Verlag der Wissenschaften, Berlin, 1962.

[96] Moore, R. E.: *Interval Analysis.* Prentice Hall Inc., Englewood Cliffs, N. J., 1966.

[97] Moore, R. E.: *Intervallanalyse.* R. Oldenbourg, München, 1969.

[98] Moore, R. E.: *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, Pennsylvania, 1979.

[99] *MPICH–A Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne. `http://www-unix.mcs.anl.gov/mpi/mpich1`.

[100] Myers, N.: *A new and useful technique: „traits“.* C++ Report, 7(5), S. 32-35, Juni 1995.

[101] Neaga, M.: *PASCAL–SC — Eine PASCAL-Erweiterung für das wissenschaftliche Rechnen.* In [84], S. 69-84, 1989.

[102] Neumaier, A.: *Interval methods for systems of equations.* Cambridge University Press, Cambridge, 1990.

[103] Nyström, E.: *Über die praktische Auflösung von Integralgleichungen mit Anwendungen auf Randwertaufgaben der Potentialtheorie.* Acta Math., 54:185-204, 1930.

[104] *Portable Batch System.* `http://www.openpbs.org`.

[105] PGM and the Processing Graph Method Tool Information: `http://www.ait.nrl.navy.mil/pgmt`.

[106] Obermaier, H.: *Computerverifikation von Lösungen nichtlinearer Integralgleichungen.* Dissertation, Universität Karlsruhe, 2003.

[107] Rall, L.B.: *Computational Solution of Nonlinear Operator Equations.* Robert E. Kreiger Publishing Company, New York, 1979.

[108] Rall, L.B.: *Automatic Differentiation: Techniques and Applications.* Lecture Notes in Computer Science, No. 120, Springer, Berlin, 1981.

[109] Ramm, A.G.: *A Simple Proof of the Fredholm Alternative and a Characterization of the Fredholm Operators.* American Mathematical Monthly, Vol. 108, p. 855, 2001.

[110] Ratschek, H.: *Die Subdistributivität in der Intervallarithmetik.* Z. Angew. Math. Mech. 51, 189-192, 1971.

[111] Ratschek, H.; Rokne, J.: *Computer Methods for the Range of Functions.* Ellis Horwood Limited, Chichester, 1984.

[112] Ratz, D.: *Automatic Slope Computation and its Application in Nonsmooth Global Optimization.* Shaker, Aachen, 1998.

[113] Rump, S. M.: *Kleine Fehlerschranken bei Matrixproblemen.* Dissertation, Universität Karlsruhe, 1980.

[114] Sachs, L.: *Angewandte Statistik.* Springer, Berlin/Heidelberg, 1997.

[115] Schaback, R., Werner, H.: *Numerische Mathematik.* 4. Auflage, Springer, Berlin/Heidelberg/New York, 1993.

[116] Schmenkel, M.: *Eine Test-Suite für C-XSC in parallelen Umgebungen.* Bachelor-Thesis, Bergische Universität Wuppertal, 2005.

[117] Scott, L.R.; Clark, T.; Bagheri, B.: *Scientific Parallel Computing.* Princeton University Press, Princeton, 2005.

[118] Sedgewick, R.: *Algorithmen.* Deutsche Ausgabe, Pearson Education Deutschland, München, 2002.

[119] Sparrow, E.M.: *Application of variational methods to radiation heat-transfer calculations.* Journal of heat transfer, Vol. 82, pp. 375-380, 1960.

[120] Spaniol, O.: *Die Distributivität in der Intervallarithmetik.* Computing 5, 6-16, 1970.

[121] Steinberg, A.: *Genaue, parallele Berechnung und Visualisierung von iterierten Funktionensystemen.* Master-Thesis, Bergische Universität Wuppertal. In Arbeit, vorauss. Fertigstellung 2007.

[122] Stoer, J.: *Numerische Mathematik 1.* 7. Auflage, Springer, Berlin/Heidelberg/New York, 1994

[123] Stroustrup, B.: *Die C++ Programmiersprache.* 4. Auflage (Deutsche Übersetzung), Addison-Wesley, München 2000.

[124] Tapamo Kahou, J.H.: *Some new Acceleration Mechanisms in Verified Global Optimization.* Dissertation, Universität Wuppertal, 2005.

[125] Top500-Liste.
`http://www.top500.org`.

[126] Torque (Tera-scale Open-source Resource and Queue manager): http://www.clusterresources.com/products/torque.

[127] Walter, W.: *Analysis I.* Springer, Heidelberg, 1985, 1990.

[128] Walter, W.: *Analysis II.* Springer, Heidelberg, 1990.

[129] Waldschmidt, K. (Hrsg.): *Parallelrechner. Architekturen – Systeme – Werkzeuge.* Teubner, Stuttgart, 1995.

[130] Werner, D.: *Funktionalanalysis.* Springer, Berlin/Heidelberg, 1997.

[131] Wiethoff, A.: *Verifizierte globale Optimierung auf Parallelrechnern.* Dissertation, Universität Karlsruhe, 1997.

[132] *XSC-Sprachen: Informationen und Download.* Wissenschaftliches Rechnen / Softwaretechnologie, Universität Wuppertal, 2000-2006. http://www.math.uni-wuppertal.de/wrswt/literatur/xsc-sprachen.html.

[133] Young, R.C.: *The Algebra of many-valued quantities.* Math. Ann. 104, 260-290, 1931.

[134] Zaanen, A.C: *Normalisable transformations in Hilbert space and systems of linear integral equations.* Acta Math., 83:197-248, 1950.

[135] Zima, H.P.; Chapman, B.M.: *Automatische Parallelisierung sequentieller Programme.* In: [129], S.563-592.

[136] Zimmer, M.: *Implementierung eines Präcompilers für Skalarproduktausdrücke in CXSC.* Bachelor-Thesis, Bergische Universität Wuppertal, 2005.

Hinweis: Zugriff auf die angegebenen WWW-Links: Stand 1.1.2007.